Attacking Android's Intent Processing and First Steps towards Protecting it

Peter Schartner · Stefan Bürger

Universität Klagenfurt
System Security | IT Services
{peter.schartner | stefan.buerger}@aau.at

Technical Report TR-syssec-12-01

Abstract

Many operating systems, including the Windows- and Android-family, are based on message processing. These messages (called events or intents respectively) are sent from the operating system to applications or vice versa or between applications. Both, Windows, and Android provide entry points (so called hooks) for additional message processing software. Since there is no check, if these additional event processing methods are malicious or not, this opens the door for well known attack scenarios like password-sniffers. But even worse, the new message processor may drop system messages or insert new (forged) messages into the event queue. In this paper we will describe, how additional message processing routines can be used to attack systems secured by mTANs (mobile TANs – TANs sent to the user's phone via the short message service – SMS) like web-banking and mobile signatures on PCs and smartphones. After describing the attack principle, we will discuss potential countermeasures and open problems.

Keywords: Attacking Android intents, smartphones, intent manipulation, intercepting mTANs, intent protection.

1 Introduction

Starting with 1.1.2011 most German and Austrian financial institutes have replaced printed TAN-lists with other security systems. Most popular mechanisms to authorize financial transactions are so called SMS-TANs (or mobile TANs – mTANs) and digital signatures. Both systems use two-factor-authentication: knowledge of a password or PIN, and possession of a mobile phone or smartcard. In this article, we will focus on mTANs or similar systems which use smartphones to send authorization data to the user.

Figure 2 shows the operating principle of financial transactions which are authorized by use of mTANs. First, as with most other e-banking processes, the user starts his web browser, opens the URL of his bank within a HTTP-session and enters login and password. This data is sent to the e-banking server, where it is verified. In the next step, the user enters his transaction, which is again sent to the e-banking server. To complete his transaction, the user clicks on a button labeled like "Send mTAN". Now, the e-banking server generates a random mTAN (most commonly 4 to 8 alphanumeric characters) and sends it via SMS to the mobile phone of the user. The user receives this short message, extracts the mTAN and types the mTAN into the according text-field of the e-banking webpage. If the verification of the mTAN at the e-banking server is successful, the transaction is completed. Otherwise it is canceled. In both cases, the user is informed accordingly via the e-banking webpage.

The advantage of this system over conventional TAN-lists is the second factor within the authentication process: possession of the mobile phone. Now, even if the attacker has full control over the PC and gets hold of login and password he still fails to complete the authorization process, because he has no control over the mobile phone of the user. In order to complete the authorization, he additionally needs either physical access to the mobile phone (which can easily be detected or prevented) or control over the mobile phone. This paper discusses ways how to achieve the latter requirement in two situations:

1. e-banking by use of a PC and a mobile phone
2. e-banking by use of a smartphone or tablet PC (or equivalent hardware) with integrated GSM/UMTS communications.

The second scenario is obviously the more dangerous and from the researchers point of view the more interesting one: here the attacker has to get control over only one device. Additionally, this device is most commonly not as secure as a PC or laptop is.

2 Related Work

In the context of PCs, some attacks, based on manipulating the message processing system, have been described. The so called "shatter attack", was described in a paper of Chris Paget [1], [2] in August 2002. After inserting an additional message handler (see figure 1 for the operating principle), the `WM_SETTEXT` message is used to copy malicious code into a text-field of an application, which is hence transferred into its memory. After inserting the malicious code, the `WM_TIMER` message was used to jump to the address of the malicious code to get it running. Countermeasures ("Shatter-proofing Windows") against this and other message-based attacks have been discussed in 2005 by Close, Karp, and Stiegler [3]. Until now, none of them has been integrated into current operating systems.

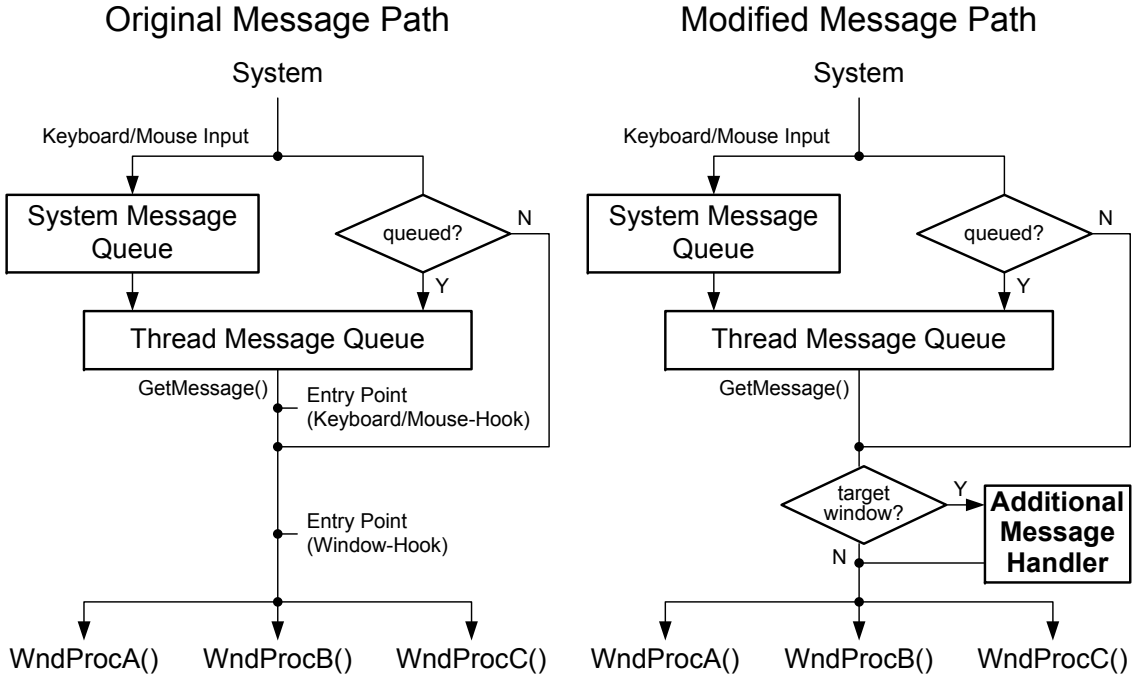

Figure 1: Original (left) and modified message path (right)

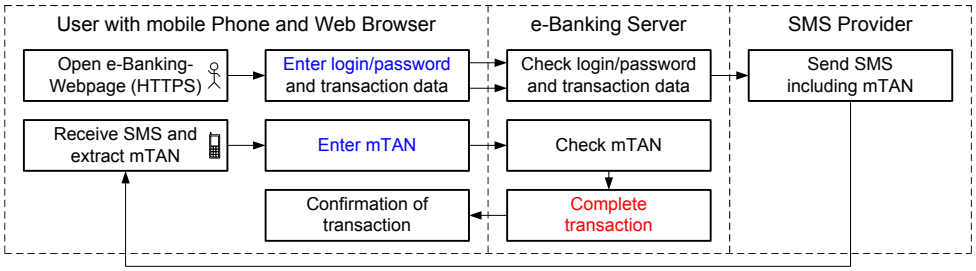

Figure 2: Authorization of transactions by use of mTANs

3 Attack Method for Smartphones

If a smartphone is used for e-banking, the web browser and the SMS management run on the same device. So the attacker's goal is to place a program (trojan) on the smartphone which provides a

1. password-sniffer to get hold of the e-banking login and password and
2. manipulates the short message processing in a way, that the user will not notice mTANs for transactions he never initiated.

Note that this attack scenario also works for GSM/UMTS-enabled tablet-PCs or laptops. Since smartphones most often are less protected against malware than tablet-PCs or laptops, we will focus on smartphones running the Android operating system [4] in the remainder of this section.

3.1 Android's Intent Passing

The Android programming environment is based on a kind of message passing system, or in Android's terminology, an intent passing (and processing) system [5]. An Android application is built from three kinds of core components. All three – activities, services and broadcast receivers – are activated through intents (see figure 3). Intent passing is used to implement late run-time binding between components in the same or different applications. Additionally, like applications installed by the user, native Android applications also use intents to launch activities, services and to respond on broadcast events.

The intent object itself is a simple passive data structure holding information about the intent [5]. We can imagine intents as a message object holding a destination component address and some data to be processed. The Android API provides three methods which accept intents as input and use the given information to start activities (`startActivitiy(Intent)`), start services (`startService(Intent)`), and broadcast messages (`sendBroadcast(Intent)`) (see [6] for details). Roughly spoken, there are two kinds of intents:

1. The ones holding information about an operation to be performed, and
2. in case of broadcasts, a description of something that has happened and is being announced.

Intent processing is divided into two groups. Explicit intents are an efficient way for sending application-internal messages, such as an activity starting a subordinate service or launching a sister activity. Explicit intents only work within and between your own applications. In all other cross-application contexts, component names would generally not be known and therefore implicit intents are available to solve this problem. For this kind of intents no target is needed and Android itself has to find the best component (or components) to handle the intent

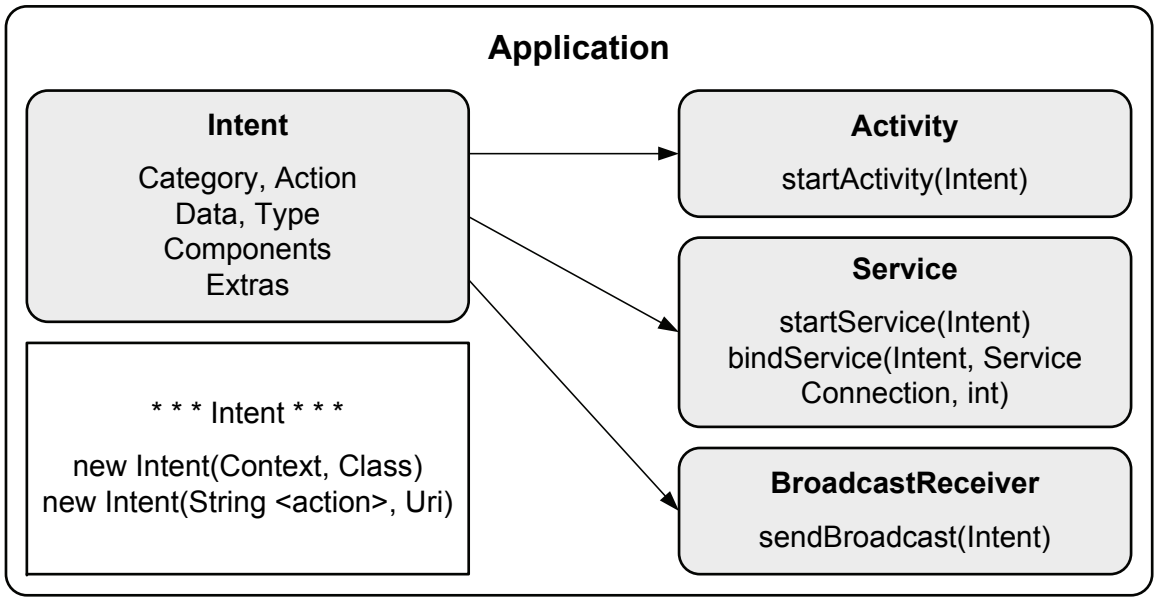

Figure 3: Android's intent processing

[5]. To find out, which component to choose from, android uses implicit names called action strings. For example if the VIEW action string is sent within an intent, the android system will automatically direct the intent to the user's preferred image viewing application. Note that this inter-component communication (ICC) is very similar to inter-process communication in Unix-based systems.

In the same way it's possible, to define action strings for your own application. At first the developer has to define a specific action string in the application manifest. And secondly he has to create an intent filter, to find out what activity or service fits best to perform on a specific action string. The intent filters make a decision according to different categories. The most dangerous and at the same time the most interesting category for our research is the `DEFAULT` category. This category allows us to replace one or more of Androids native applications such as the SMS application. That means, a self-developed application can be used to respond on an implicit intent. Afterwards the intent can either be discarded or forwarded to the next matching activity, service or broadcast receiver (see [7] for details). If an intent is used to send an broadcast message there are two ways other components are informed. Either `sendBroadcast(intent)` is used and the android system decides by checking the intent filters of all broadcast receivers which fits best. Or, if the order in which the broadcast receiver receives the intent is important, `sendOrderedBroadcast(intent)` can be used. In the second case a priority value must be set in the application manifest. This value, set within the `<intent-filter>` tag, is an integer where a higher number stands for a higher priority.

3.2 Android's Security System

The android security system is built on the base of a standard Linux system with some additional features. Data and applications are protected through a two-level security mechanism. One directly at the system level and the second one at the inter-component communication (ICC) level, built on the guarantees provided by the underlying Linux system. In almost all cases android is able to prevent the system itself of higher damage because each application runs as an independent process. The whole ICC-communication is controlled by the android middleware by reasoning about labels assigned to applications and components. This so called

access permission label is a simple text string, set in the application manifest. To control the security of an application, developers have to assign a collection of permission labels (see [6]). While "installing" an application, the user is asked to grant all requested permissions.

```
<uses-permission android:name=
  "android.permission.RECEIVE_SMS"/>
```

When an intent is sent by a component, the reference monitor looks at the permission labels assigned to the called application. The intent will be processed if and only if the target's permission label is in the collection of the calling component ICC establishment. That directly means that a developer has to know the permissions of each component he wants to use.

Normally, when creating your own components and associated permission labels, nobody can get access to your components because of unknown permission labels. But in some cases this security level isn't high enough. For this purpose a developer can set his component private and ensure that no one other can use it. So, in order to increase the security of self-developed applications and components, the private attribute should always be used.

As long as permission labels are unknown android and its security system is very secure at the application level. But, whenever a permission label is known publicly, such as the android built-in permission labels, and users aren't carefully enough to decide which permissions should be granted and which shouldn't, android has significant security risks at the user level. Nevertheless it's not worth to lose all the benefits the android system brings to the mobile world by changing the system architecture or philosophy. Rather, it gets most important to inform the users that risk and security is up to them.

3.3 Modified Message Processing

The idea behind our attack is the interception of intents and the replacement of all native Android activities and services used to perform an e-banking transaction. If we can achieve this, we are able to do an e-banking transaction entirely without any user input. One thing we have to consider before talking about this manipulation, is Android's permission system. The one and only possibility to get access to the intent system is to install a custom application such as a general e-banking program. During the installation process Android asks the user to grand permissions to use and replace activities, services and to listen to broadcast events. And the interesting thing here is, all Android permission checks are done at installation. On later execution the user is never prompted again to reevaluate those permissions (see [7] for details).

```
<uses-permission android:name=
  "android.permission.RECEIVE_SMS"/>
```

After this permission is granted, we have to register our application as listener for incoming SMS messages. The registration is done in the application manifest file by defining a broadcast receiver listening on the `SMS_RECEIVED` event. If additionally the category is set to `DEFAULT`,our newly installed application is the first one to be informed, if a new SMS message arrives.

```
<receiver android:name="mySMSReceiver">
  <intent-filter>
    <action android:name=
      "android.provider.Telephony.
      SMS_RECEIVED"/>
    <category android:name=
      "android.intent.category.DEFAULT"/>
  </intent-filter>
</receiver>
```

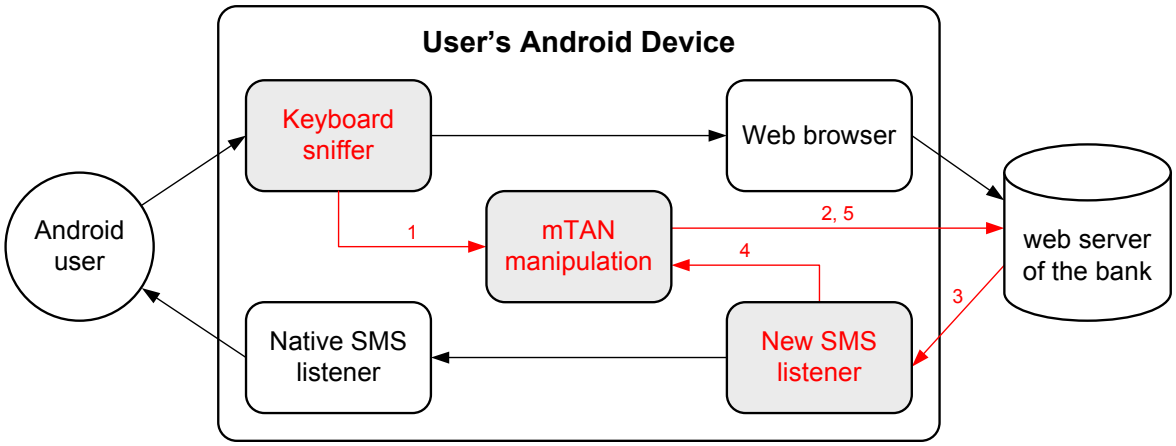

Figure 4: Attacking e-banking – scenario 1

But this is only the first step to manipulate an e-banking transaction. The second thing to do is to get access to the users account data. To achieve this, we will employ a keyboard sniffer, which filters all intents [8] and waits for the user to type the banks URL followed by login and password. Note that the keyboard sniffer is not limited to physical keyboards. In case of virtual keyboards, we can detect the left-click and analyze the screenshot of the surrounding area of the "cursor" (in case of a touch-screen the position of the fingertip), which will reveal the number, the user just clicked (or tipped on) on.

Just by using this little amount of technical knowledge, we can consider the following straight forward attack scenario. The user opens up the web browser and visits his e-banking portal. The installed key sniffer is activated by entering the banks URL and catches the entered login data (step 1 in figure 4). After initiating a transaction on the bank's online portal, a confirmation short message, holding the mTAN is sent to the customer. The new SMS listener receives the mTAN SMS, and delivers the intent to the native SMS application without any modifications. Hence, the user isn't able to notice this initial step because the user triggered transaction is performed as expected. But from now on our attacking tool works completely autonomous (see figure 4). After opening the banks URL and providing login and password (step 2), the attack tool initiates a transaction. By intercepting the mTAN short message (steps 3 and 4), the transaction can easily be completed in step 5. In contrast to the user triggered transaction, the mTAN short message is now deleted after the transaction has been completed.

Obviously, opening an e-banking session on the victims phone is quite time consuming and hence very risky for the attacker. So we have to improve our attack scenario in order to reduce the risk of the attacker being detected. The better way of attacking from the attackers point of view, is to forward login and password to the attacker's device (step 1 in figure 5). After receiving login and password, the attacker starts the e-banking session, provides login and password and initiates a transaction. The bank server responds with an mTAN short message which is intercepted by the new SMS listender and – like login and password – forwarded from the victim's device to the attacker's device (steps 3,4 and 5). Finally the mTAN is used to complete the malicious transaction (step 6). Using this attack method, the victim has very little chances to detect the malicious bank transaction.

Anyhow, the Android system itself is a very secure platform, because there is no way getting around permissions granted by the user. But if the user makes one mistake or is tricked by a wrong application description, everything becomes possible. Therefore you have to see Android

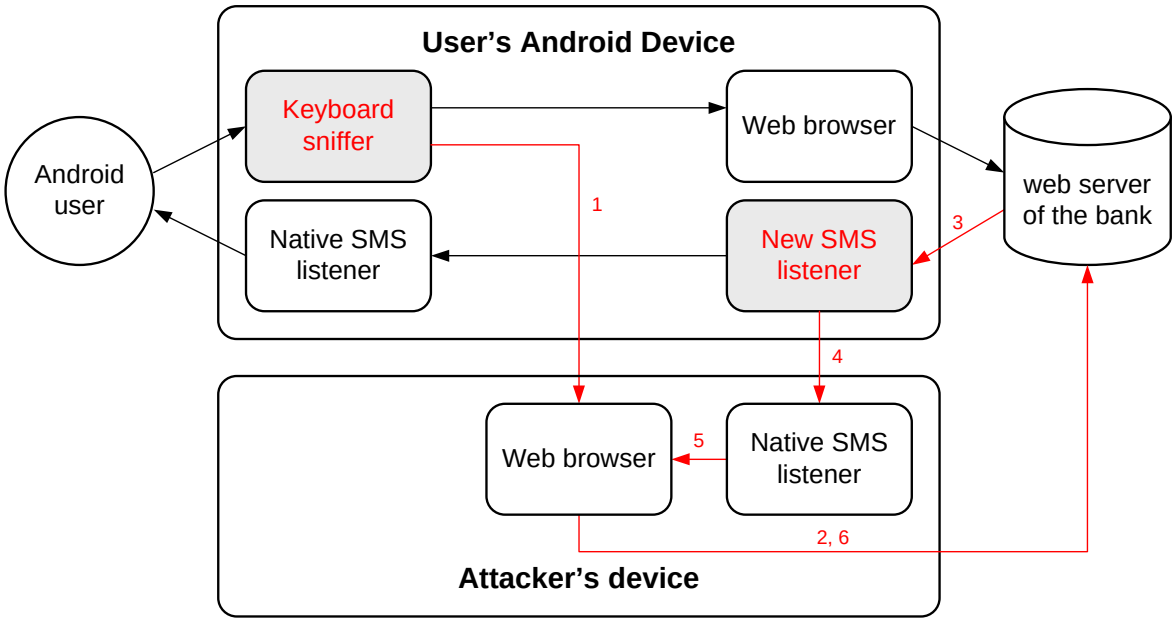

Figure 5: Attacking e-banking – scenario 2

like every other computer operating system where the user has complete control and the full responsibility for every installed program. In any case, Android phones are not a bit similar to old cell phones with minimal functionality and maximal security. Hence cell phone users have to rethink how to act and what could happen when using such a powerful Android device (see [7] for details).

If the victim uses a PC to run the web browser, the attacker has to get control over the PC and the users mobile phone. Assuming that the attacker has control over the victims PC or managed to get the e-banking login and password by means of classical attacks like phishing, he only needs to get hold of the mTAN short message which authorizes a specific transaction. In this case, the attack tool for the smartphone is reduced to the manipulation of the SMS management.

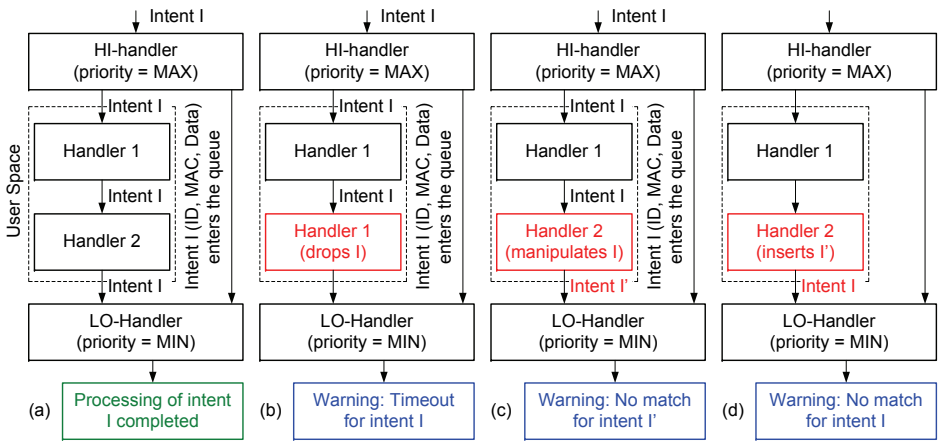

Figure 6: Surveillance of the Event Processing Path

3.4 Open Problems

Due to their high risk potential, keyboard intent handlers cause a warning message each time they are inserted into the system's intent processing. Contrary to the displayed requested permissions when installing the application, this warning should raise red flags and hence keyboard sniffers should not be undetectable. In order to disguise our attack we need to find a workaround. Attack methods for virtual PC keyboards might help: here, the attacker traces the mouse movement and in case of a click he takes a (local) screenshot.

Our actual research shows no possibility to capture screen at the current position of the finger(s) while typing on the soft keyboard. The main problem here is that there is no API provided to do this on a not rooted android smartphone. This on one hand means the standard user is provided with a very secure operating system without having to be afraid of identity theft. Unfortunately on the other hand (from the point of view of this paper) we have to think of other attack scenarios. The most feasible way should be a homemade application, for example a malicious e-banking application or an alternative web browser. In this case we get full access to soft keyboard text inputs when using a self-made view component. Here, a special method from the view class (`onKeyPreIme(int keyCode, KeyEvent event)`) can be used to catch all key events before the android IME (input method) consumes them. At the moment it is not clear, if permissions have to be granted to catch the keyboard input in a way like this. Nevertheless it is worth mentioning, that the return value of the view object's `onKey...()`-Methods (see [9] for details) is either `true` if the method handled the event, or `false` if the event should be handled by the next receiver. This strongly suggests that there is a way to stealthily control the keyboard events.

Another open problem is related to the manipulation of outgoing short messages. This process seems not to be based on intents. Intents are only used to inform the system (by a broadcast message) that a short message has been sent. Obviously, replacing the SMS application will help, but our aim is to manipulate the behavior of the native SMS application: specific messages, e.g. related to eBanking, should be blocked or forwarded to the attackers phone.

4 Countermeasures for PCs and Smartphones

4.1 Current Situation

At the time, the operating systems of the Windows-family and currently available anti-virus- and anti-malware-software do not detect our prototype as being potentially malicious. As a first step, there should at least be a warning for the user, if some program intends to modify the message processing paths of the system.

The Android operating system displays a (warning) message, containing information on the components (like contacts, WLAN or GPS) being used, when "installing" a new applet. Hence the user can (or more precisely has to) decide, if he grants access or not. But in case of installing a trojan with some obvious benefit for the user (like in our application scenario support for SMS-management or SMS-SPAM-filtering), it may be quite reasonable for the user that the applet is going to use his contacts, and messaging capabilities. So simply displaying the requested access rights is not enough to protect the user from malware.

4.2 Warnings and automated Blocking of unwanted Actions

Instead of simply displaying the requested access rights, anti-malware-software could, depending on the selected protection level, completely block the installation process of specific programs.

Besides access rights, intent handlers have to be assigned a priority. This priority is used for ordered insertion of the handler into the intent processing queue: an intent handler with a specific priority is installed before all handlers with lower priority (also including the systems intent handlers). These priorities can be used to improve the quality of information, which is displayed to the user. Now, the displayed requested resources and access rights can be sorted by their risk-level and their priority, so that the most "dangerous" ones appear at the beginning of the list. Think of a list with dozens of entries; most commonly (like it is the case with disclaimers and other information displayed during installation) the users will not read to the end.

4.3 Detecting manipulated Messages/Intents or manipulated Processing Paths

Concerning the priority of androids intent handlers, we came up with the idea of inserting one special handler with the highest possible priority (HI-handler) and one with the lowest possible priority (LO-handler – see figure 6 left). The first idea was to attach some sort of cryptographic checksum to all incoming intents. This raises two problems: first we need a shared secret with the second handler and second, the second handler has no chance to detect dropped intents. So we decided to inform the second handler about the arrival of a certain intent. Based on this information, the second handler can now check if

1. all intents have been processed (i.e. have run through) the entire event processing queue,
2. no intents have been manipulated on their way through the event processing queue and
3. no additional intents have been inserted into the processing queue (from handlers residing inside the processing queue).

By this, we can detect deleted (see warning (b) in figure 6), manipulated intents (see warning (c) in figure 6) and inserted intents (see warning (d) in figure 6). Unfortunately, we can not detect intents inserted by malicious programs, because these intents would enter the processing queue at the top, where the high-priority-handler is located. Nevertheless, logging the events and processing the logged data by use of anomaly detection, could help to identify malicious programs.

Obviously, both handlers sketched above should either be integrated in the android OS (i.e. outside the space for user handlers) or both should have priority levels not accessible to ordinary user programs. Otherwise, the attacker could insert his handler before the system's first handler and hence again compromise the system.

5 Conclusion

In this paper we presented attack methods for authorization processes based on SMS-TANs. The presented attack scenarios work on all systems based on mTANs: Besides e-banking, examples include so called mobile signatures (here, a server digitally signs documents on the users behalf; the authorization is again secured by means of mTANs [10]) and single-sign-on-systems (e.g. ProSoft's SMS Passcode [11]).

At the time of writing, we are working on a proof-of-concept implementation of the attack scenarios and the countermeasures described above for Android smartphones. Additionally, current research includes refinement of the proposed countermeasures and developing further mechanisms to automatically detect malicious intent handlers.

References

[1] C. Paget (alias Foon), "Exploiting design flaws in the Win32 API for privilege escalation. Or... Shatter Attacks – How to break Windows." archived version on http://web.archive.org/web/20060904080018/http://security.tombom.co.uk/shatter.html, August 2002.

[2] ——, "Shatter attacks - more techniques, more detail, more juicy goodness." archived version on http://web.archive.org/web/20060830211709/security.tombom.co.uk/moreshatter.html, Mai 2003.

[3] T. Close, A. Karp, and M. Stiegler, "Shatter-proofing Windows," archived on http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Close/tylerclose_whitepaper_US05.pdf, 2005, (Whitepaper at Black Hat USA 2005).

[4] Android, "Android Operating System," http://www.android.com, 2011.

[5] ——, "Android Developers: Intents and Intent Filters," http://developer.android.com/guide/topics/intents/intents-filters.html, 2011.

[6] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security & Privacy Magazine*, vol. 7, pp. 50–57, 2009.

[7] R. Meier, *Professional Android 2 Application Development*. John Wiley & Sons Ltd, 2010.

[8] Android, "Android Developers: Handling UI Events," http://developer.android.com/guide/topics/ui/ui-events.html, 2011.

[9] ——, "Android Developers: public class View," http://developer.android.com/reference/android/view/View.html, 2011.

[10] A-Trust, "Mobile Signatur, Handy Signatur," http://www.handy-signatur.at, 2011.

[11] ProSoft, "SMS Passcode," http://www.prosoft.de/produkte/sms-passcode, 2011.