2013 5th International Workshop on Near Field Communication (NFC)

Applying Relay Attacks to Google Wallet

Michael Roland, Josef Langer
NFC Research Lab Hagenberg
University of Applied Sciences Upper Austria
{michael.roland, josef.langer}@fh-hagenberg.at

Josef Scharinger
Department of Computational Perception
Johannes Kepler University Linz
josef.scharinger@jku.at

***Abstract*—The recent emergence of Near Field Communication (NFC) enabled smart phones resulted in an increasing interest in NFC security. Several new attack scenarios, using NFC devices either as attack platform or as device under attack, have been discovered. One of them is the software-based relay attack. In this paper we evaluate the feasibility of the software-based relay attack in an existing mobile contactless payment system. We give an in-depth analysis of Google Wallet's credit card payment functionality. We describe our prototypical relay system that we used to successfully mount the software-based relay attack on Google Wallet. We discuss the practicability and threat potential of the attack and provide several possible workarounds. Finally, we analyze Google's approach to solving the issue of software-based relay attacks in their recent releases of Google Wallet.**

I. Introduction

The steadily increasing number of smartphone models equipped with support for Near Field Communication (NFC) has lead to several research activities on smartphone and NFC security. Security critical and sensitive applications like payment have gained a particular interest. Researchers have performed all kinds of security evaluations of programming interfaces (APIs), application concepts and actual implementations of payment systems. Examples are the forensic analysis and reverse engineering of Google Wallet [2]–[4], the analysis of secure element APIs in Android [5]–[7] and discovery of vulnerabilities [8] in Google Wallet.

A. Relay Attack

A well-known issue with contactless payment cards and secure element-enabled devices is the relay attack. First evaluated in the context of contactless smartcards by Hancke [9] and Kfir and Wool [10], the practicability of relay attacks has greatly improved due to the availability of NFC-enabled mobile phones. Francis et al. [11] showed that it is possible to relay NFC signals over Bluetooth using two mobile phones. They [12] further revealed that, with the introduction of software card emulation in some smart phones, it is even possible to relay contactless credit card transactions and electronic passport transactions between two phones. Roland [13] further analyzed the security implications of software card emulation.

A relay attack can be seen as a simple range extension of the contactless communication channel (see Fig. 1). Thus, an attack requires three components:

1) a reader device (also called *mole* [9]) in close proximity to the card under attack,
2) a card emulator device (also called *proxy* [9]) that is used to communicate with the actual reader, and
3) a fast communication channel between these two devices.

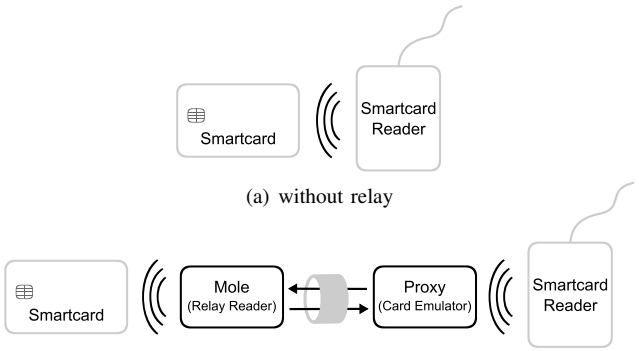

Figure 1. Communication between a smartcard and a reader. (Source: [1])

The attack is performed by bringing the mole in proximity to the card under attack. At the same time, the card emulator is brought into proximity of a reader device (POS terminal, access control reader...) Every command that the card emulator receives from the actual reader is forwarded to the mole. The mole, in turn, forwards the command to the card under attack. The card's response is then received by the mole and sent all the way back through the card emulator to the actual reader.

This type of attack cannot be prevented by application-level cryptography [9], [14]. The problem is that the relay attack is a simple range extension of the contactless interface, so neither the mole nor the card emulator needs to "understand" the actual communication. They simply proxy any bits of data they receive. As a consequence, current EMV credit card payment protocols using Mag-Stripe mode as well as EMV ("Chip & PIN") mode can be relayed.

As existing cryptographic protocols on the application layer cannot prevent relay attacks, several alternative methods have been identified to prevent or hinder relay attacks [9], [10], [14]:

1) The card's radio frequency interface can be shielded with a Faraday cage (e.g. aluminum foil) when not in use.
2) The card could contain additional circuitry for physical activation and deactivation.
3) Additional passwords or PIN codes could be used for two-factor authentication.
4) Distance bounding protocols can be used on fast channels to determine the actual distance between card and reader.

Other measures – like measurement of command delays to

This paper is a revised version of the technical report "Applying recent secure element relay attack scenarios to the real world: Google Wallet Relay Attack" arXiv:1209.0875 [1].

DOI 10.1109/NFC.2013.6482441

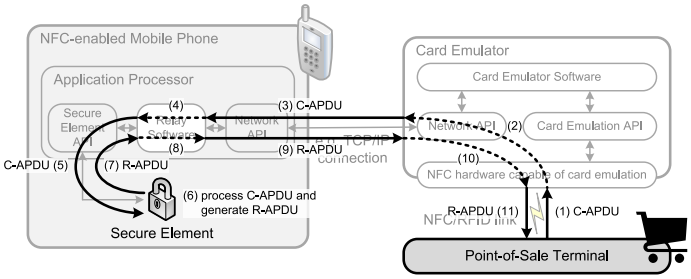

Figure 2. Relay scenario: Relay software – installed on the victim's phone – relays commands (C-APDU) and responses (R-APDU) between the secure element and the card emulator across a wireless network. (Source: [1])

detect additional delays induced by relay channels – have been identified as not useful. For instance, Hancke et al. [14] conclude that the timing constraints of ISO/IEC 14443 are too loose to provide adequate protection against relay attacks.

While initial approaches to relay attacks [9], [10] focused on forwarding physical layer protocols (bit transfer level), recent approaches [7], [11], [12], [15] skip the lower layers and directly transfer application layer protocols (APDUs, application protocol data units). This relaxes timing requirements and greatly improves achievable relay distances.

B. The Next Generation: Software-based Relay Attack

The threat potential of relay attacks was mitigated by the fact that all relay scenarios required close physical proximity to the device under attack. However, recent research [7], [15] follows a different approach. Instead of accessing a device's secure element through the external (contactless) interface, it is accessed from the device's application processor through the internal interface. While the original relay attack required mole hardware in physical proximity of the device under attack, pure software (malware) on an attacked device's application processor replaces the physical mole.

The complete relay system, as suggested in [7] and verified in [15], and the flow of relayed smartcard commands (APDUs) are shown in Fig. 2. The system consists of four parts:

1) a mobile phone (under control of its legitimate user),
2) a relay software (under control of the attacker),
3) a card emulator (under control of the attacker), and
4) a reader device (e.g. at a point-of-sale terminal).

The relay software is installed on the victim's mobile phone. This application is assumed to have the privileges necessary for access to the secure element and for communicating over a network. These privileges can be either explicitly granted to the application or acquired by means of privilege escalation. The relay application waits for APDU commands on a network socket and forwards these APDUs to the secure element. The responses are then sent back through the network socket.

The card emulator is a device that is capable of emulating a contactless smartcard in software. The emulator has RFID/NFC hardware that acts as a contactless smartcard when put in front of a smartcard reader. The emulator software forwards the APDU commands (and responses) between a network socket and the emulator's RFID/NFC hardware.

Command APDUs (C-APDUs) received from the reader device are routed through the card emulator and over a wireless network (cellular, WiFi, Bluetooth...) to the victims device. There, the relay app forwards the C-APDUs to the secure element. The corresponding responses (R-APDUs) generated by the secure element are routed all the way back (through the relay app, the wireless network and the card emulator) to the reader device.

C. Access to the Secure Element

A critical requirement for the software-based relay attack is that the relay software can exchange APDUs with the secure element. Roland et al. [7] analyzed various schemes for access control to the secure element. They concluded that, even though some of these schemes provide sophisticated access control capabilities, all of them have one significant flaw: They all rely on the mobile device's operating system (executed on the application processor) to perform access control enforcement. Thus, in all cases, the secure element (secure component) blindly trusts the application processor's (insecure component's) access control decisions. Therefore, once an application passes the security checks performed by the operating system on the application processor, it can exchange arbitrary APDUs with the secure element. Considering the current trend in privilege escalation exploits for various mobile device platforms (cf. [15], [16]), we assume that an arbitrary application can use exploits to bypass restrictions and security checks performed by the operating system on most platforms that are currently in the field.

II. Google Wallet

Google Wallet is a container for payment cards, gift cards, reward cards and special offers. It consists of an Android app with a user interface and JavaCard applets on the secure element. The user interface is used to protect the wallet with a PIN code, to manage the payment, gift and reward cards, to select the currently active card, to find specific offers and to view the transaction history. The secure element is used to store sensitive information of the payment, gift and reward cards and to interact with existing POS reader infrastructures. The analysis and attack described in this paper have been performed with version 1.1-R52v7 of the Google Wallet app and the secure element applets installed in February 2012.

For our analysis, we added debug output to Android's secure element API (`com.android.nfc_extras`). As this is a separate library (.jar file), we were able to modify its source and re-compile the library. We added log messages that reveal the name of the interface class, the name of the method, the method parameters and the return values. We then replaced the library's .jar file on the device with our new one. With this debug output we were able to monitor access to the various methods of the API. The debug log also allowed us to trace all APDUs exchanged between Google Wallet and the secure element.

Upon first start, the Google Wallet app initializes the secure element and installs a PIN code that is necessary for using the app's user interface. During initialization several applets are installed and personalized on the secure element using

GlobalPlatform card management. Specifically, a secure channel based on the secure channel protocol SCP02 is established between the secure element and a remote server which performs the card management through this authenticated and partly encrypted channel.

A. Static Structure

After successful initialization, several applets can be found on the secure element. The following is a list of applet instance identifiers (AIDs) that have been identified to be used by either Google Wallet during normal operation or by POS terminals during payment transactions:

1) A000000476 2010
2) A000000476 3030
3) 325041592E5359532E4444463031
4) A000000004 1010
5) A000000004 1010 AA54303200FF01FFFF

AID 1 is the Google Wallet on-card component, which is used by Google Wallet to manage the payment cards on the secure element. AID 2 is the Google MIFARE access applet, which is used to manage the MIFARE Classic 4K memory of the secure element. Both applet instances can only be accessed from the application processor. Selecting them though the RF interface results in the error code 6999, denoting that the selection of those applets failed.

The last three applet instances are related to credit card payment transactions. AID 3 is the directory definition file of the EMV Proximity Payment System Environment as mandated by [17]. It contains a list of all activated credit card applications. It can be selected from the application processor and through the RF interface. While Google Wallet is in locked state an empty list is returned, otherwise the list contains AID 4 and 5.

AID 4 is the AID for a regular MasterCard credit card. We assume that the other credit card AID (AID 5) denotes a MasterCard co-branded as Google Wallet but we have no confirmation for this[1]. Both credit card applet instances contain equal data structures (including the same primary account number, PAN). The credit card applets are only selectable while Google Wallet is in unlocked state, otherwise the error code 6999 is returned. Access to both applet instances is possible from the application processor and through the RF interface.

B. Interacting with Google Wallet's On-Card Component

Several commands used by the Google Wallet app to interact with its on-card component have been identified:

- 00 A4 0400 07 **A0000004762010** 00 is used to select the on-card component.
- 80 E2 00**AA** 00 is used to unlock the wallet and allow access to the credit card application after the Google Wallet app has successfully verified the PIN. The PIN itself is *not* verified by the on-card component.
- 80 E2 00**55** 00 is used to lock the wallet.
- 80 CA 00A5 00 returns a list containing the two credit card instances.
- 80 F0 **01**00 12 4F 10 **A0000000041010AA54303200FF01FFFF** 00 is used when the Google Prepaid card is disabled through the Google Wallet app.
- 80 F0 **02**00 12 4F 10 **A0000000041010AA54303200FF01FFFF** 00 is used when the Google Prepaid card is enabled through the Google Wallet app.

C. Google Prepaid Card: A MasterCard PayPass Card

The credit card applet is based on the *EMV Contactless Specifications for Payment Systems* and is a MasterCard PayPass card. It supports Mag-Stripe mode with dynamic CVC3 (card verification code) and requires online transactions. Full EMV mode is not supported. Also there is no cardholder verification. Even though, PIN-based cardholder verification would significantly complicate relay attacks or would even render them impossible if the attacker is unable to "guess" the PIN.

A typical Mag-Stripe mode transaction with the Google Prepaid card consists of the following command sequence (cf. [1] for a detailed analysis of a Mag-Stripe transaction):

1) POS → Card: 00 A4 0400 0E **325041592E5359532E4444463031** 00: Select Proximity Payment System Environment (PPSE).
2) Card → POS: 6F 3A 84 0E **325041592E5359532E4444463031** A5 28 BF0C 25 61 15 4F 10 **A0000000041010AA54303200FF01FFFF** 87 01 **01** 61 0C 4F 07 **A0000000041010** 87 01 **02**: Respond with a list of supported EMV payment applications and their priority indicators.
3) POS → Card: 00 A4 0400 10 **A0000000041010AA54303200FF01FFFF** 00: Select MasterCard Google Prepaid card.
4) Card → POS: 6F 20 84 10 **A0000000041010AA54303200FF01FFFF** A5 0C 50 0A **4D617374657243617264**: Respond with application details (here: application label "MasterCard").
5) POS → Card: 80 A8 0000 02 8300 00: Get processing options.
6) Card → POS: 77 0A 82 02 **0000** 94 04 **08010100**: Respond with the application interchange profile (Mag-Stripe only, online transactions only, no cardholder verification...) and the location of the Mag-Stripe data file (elementary file 1, one record starting at record 1).
7) POS → Card: 00 B2 **010C** 00: Read record 1 of record data file 1 (Mag-Stripe data).
8) Card → POS: 70 6A 9F6C 02 **0001** 9F62 06 0000000 00038 9F63 06 0000000003C6 56 29 42**3534333 0xxxxxxxx30xxxx37xxxxxxxx**5E**202F**5E**313731 31313031**303031303030303030303030 9F64 01 04 9F65 02 0038 9F66 02 03C6 9F6B 13 **5430 xxxx0xx7xxxx**D**1711101**0010000000000F 9F67 01 04: Respond with the Mag-Stripe version, track 1 and track 2 information.
9) POS → Card: 80 2A 8E80 04 **nnnnnnnn** 00: Compute cryptographic checksum for a given unpredictable number **nnnnnnnn**.
10) Card → POS: 77 0F 9F61 02 **zzzz** 9F60 02 **yyyy** 9F36 02 **xxxx**: Respond with the application transaction

[1] An installation with version 1.5-R79-v5 of the Google Wallet app and version 1.6 of the on-card component installed in September 2012 reports the second credit card AID as A0000000041010AA539648FFFF00FFFF.

counter (xxxx) and dynamically generated CVC3 for track 1 (yyyy) and track 2 (zzzz).

Most of the data exchanged in a Mag-Stripe transaction is static for all transactions (e.g. the Mag-Stripe data). *Compute cryptographic checksum* (9 and 10) is the only APDU command/response pair that contains dynamically generated data that differs for each transaction: the unpredictable number generated by the POS, and the transaction counter and CVC3 codes generated by the card. Each *compute cryptographic checksum* command that is sent to the card must be preceded by a fresh *get processing options* (5 and 6) command. Thus, the minimum sequence for generating a dynamic CVC3 is

1) *Select MasterCard Google Prepaid card* (3 and 4),
2) *Get processing options* (5 and 6),
3) *Compute cryptographic checksum* (9 and 10).

III. Google Wallet Relay Attack

We applied the software-based relay attack scenario to Google Wallet to verify its applicability. *Google Wallet* has been chosen for several reasons:

- Google Wallet already has a huge user base[2].
- It complies with EMV payment standards and can be used with any point-of-sale terminal that supports MasterCard PayPass.
- Due to Android being an open source system, it was fairly easy to explore the Android NFC stack. Also, we could easily implement monitoring of Google Wallet's interaction with the secure element.
- Google Wallet is well-known for being used on rooted devices which means that the operating system's security measures are already weakened/bypassed on those devices.

A. The Relay App

The relay app, a purely Java-based Android app, is a simple TCP client that maintains a persistent TCP connection to a remote server (the card emulator). When the card emulator requests access to the secure element, a connection is established through Android's hidden[3] secure element API (class `NfcExecutionEnvironment` in `com.android.nfc_extras`). The app first selects the Google Wallet on-card component and sends the unlock command. This imitates the behavior of the Google Wallet app upon successful PIN entry by the user. The relay app then listens for command APDUs on its network interface and forwards them to the secure element. The response APDUs from the secure element are transmitted back to the card emulator. When the transaction is complete, the Google Wallet on-card component is selected again and the lock command is used to lock the wallet.

For this test scenario, the relay app has been manually granted the permissions necessary to access the secure element. On Android 2.3.7, this was done by loosening the restrictions for access to the secure element API in a customized firmware. On Android 4.0.3, the relay app's signature was added to the secure element permissions file (`/system/etc/nfcee_access.xml`). Root access to the device was necessary in all cases. Instead of manually granting the permissions, privilege escalation exploits could be integrated into future versions of the app to automate this process. For easy integration of future exploits, a privilege escalation framework (cf. [18]) could be embedded into the app.

For the purpose of our tests, the relay app has a foreground component that needs to be started manually and provides configuration options like the IP address of the card emulation server. Moreover, app requires user confirmation to establish the connection to the card emulator. However, for an actual attack, the app could be started automatically on device boot-up and run completely in the background.

[2]Google Play Store listed more than 500,000 installations in early 2012. As of September 2012 Google Wallet has already over 1,000,000 installations.

[3]I.e. not included in the public Android API documentation.

Figure 3. Card emulator made from a notebook and an ACS ACR 122U NFC reader. (Source: [1])

B. The Card Emulator

The card emulator (Fig. 3) has been built from an ACS ACR 122U NFC reader and a notebook computer (running a card emulation server application). The ACR 122U supports software card emulation mode and is available for less than EUR 50 (including taxes and shipping) from touchatag.com. Several examples on how to use this device in card emulation mode can be found on the web.

The card emulation software (written in Python) contains a TCP server that listens for incoming connections from the relay app. Once a TCP connection has been established, the emulation server puts the ACR 122U into card emulation mode and waits for commands from a POS terminal. When the card emulator detects activation by a POS terminal (or any other smartcard reader), it requests access to the secure element through the relay app. Then, all received command APDUs are forwarded through the network interface to the relay app and all response APDUs received from the relay app are returned to the POS terminal. When the ACR 122U leaves the range of the POS terminal (RF field is no longer detected), the connection to the secure element is closed.

C. Test with a real Point-of-Sale (POS) Terminal

We successfully tested the Google Wallet relay attack by paying at a POS terminal. The POS terminal used for our test was a Hypercom Artema Hybrid with a ViVOtech ViVOpay 5000 contactless reader. For ethical reasons we used our own credit card terminal instead of a POS installation in

the field. However, our POS terminal is identical to those used in recent roll-outs at Schlecker and Zielpunkt in Austria. Videos of the successful relay attack are available on YouTube: http://www.youtube.com/watch?v=hx5nbkDy6tc and http://www.youtube.com/watch?v=_R2JVPJzufg.

IV. Discussion

An NFC reader device (available for less than EUR 50), a notebook computer and some average programming skills are all that was necessary to mount this attack. However, we admit that, while using the ACR 122U together with a notebook computer worked in our controlled environment, this setup will certainly raise suspicions when used to pay in a store.

An alternative approach would be to use another mobile phone as card emulator. Francis et al. [12] showed that a credit card can be emulated using a BlackBerry 9900 in software card emulation mode. Other NFC-enabled BlackBerry devices, like the BlackBerry 9380, can be used as well. Changes introduced to the CyanogenMod 9.1 aftermarket firmware for Android enable software card emulation on Android devices (cf. [13]). A mobile phone has several advantages:

- accepted form factor for mobile contactless transactions,
- same network interfaces as the device under attack, and
- a BlackBerry 9380 is available for less than EUR 300.

A. Getting the Relay App on Devices

To roll out the relay app to user's devices, it could be integrated into any existing app downloaded from Google Play Store. The infected app could then be re-published on Google Play Store under similar (or even identical) publisher information and with the same app name as its original. The publisher account that is necessary to re-publish the app costs EUR 20.

For many users it would be difficult to distinguish the original app from the malware, as these apps would only differ in the number of installs and user comments. To specifically target users of rooted devices, an app that already requires root permissions could be used as a base for code injection.

Google started to combat this approach with their recent updates to the Google Play Developer Program Policy.

B. Transaction Limits

In Austria, PIN-less contactless transactions are usually limited to EUR 25. However, experience reports on the Internet suggest that Google Wallet can be used for transactions of at least up to USD 100 (approx. EUR 75). An attacker would typically not attack a single Google Wallet device, but instead distribute transactions on many devices infected with the relay app. Thus, an attacker could build a "bot network" of Google Wallets. This method has the advantage that each wallet would be charged less, which might cover the attack for a longer period. Also, the attacker could use the "bot network" to select a device with a good (i.e. stable and fast) network connection.

C. Improving the Attack

The analysis [15] of communication delays induced by a relay attack reveals that the relay channel adds a significant portion of the overall command/response delay. This results in a noticeable slow-down of relayed transactions in comparison to direct transactions. One possibility to improve the speed of relayed transactions is to cache all static transaction data and only transmit dynamically generated data during the transaction. Thus, only the dynamic fields of the *get processing options* command/response pair need to be exchanged. All other data can be retrieved from the Google Wallet device prior to the attack. This reduces the number of bytes exchanged over the relay channel during a transaction from 296 bytes to 10 bytes.

D. Possible Workarounds

We identified several possible workarounds. Each of them has its advantages and disadvantages.

1) Timeouts of POS Terminals: An easy, but potentially unreliable, measure to prevent relay attacks would be the enforcement of short timeouts (e.g. the benchmark targets specified by the EMV specifications) for payment transactions on the POS terminals. Transactions taking longer than this timeout should be interrupted or discarded. While this measure will prevent most long-distance relay scenarios, relays over shorter distances and fast communication channels might not be rejected. Also, such tight timeouts will prevent cloud-based EMV applications (cf. [13] and YES-wallet, http://www.yes-wallet.com/).

2) Google Wallet PIN Verification:

> "*A PIN and the ability to remotely disable Google Wallet make it very safe.*" [19]

In version 1.1-R52v7 of the Google Wallet app, the PIN that protects the wallet is only verified within the mobile phone app. Simple lock and unlock commands are used to control the state of the on-card component instead of on-card PIN verification. This, once more, delegates access control for a secure component (Google Wallet on-card component and credit card applets) to a potentially insecure component (application processor). The on-card component does not verify this PIN.

PIN verification could be handled by the on-card component on the secure element. After all, PIN verification is a core component of smartcards anyways. In that case, the attacker would need to know the wallet's PIN in order to conduct a successful attack.

Another approach would be to require PIN entry and online PIN verification at the point-of-sale for any transaction amount. However, this is impracticable or even impossible at certain points-of-sale.

3) Disabling Internal Mode Communication for Payment Applets: Modern secure elements (like those embedded into Google's Nexus devices) provide instruments to distinguish between external communication (RF contactless interface) and internal communication (application processor) from within a JavaCard applet. Rules for interface based access can be applied on a per-applet basis and even on a per-APDU basis. These capabilities could be used to disable internal mode communication for all payment applets and consequently disable their vulnerability for software-based relay attacks.

The disadvantage of this workaround is that the secure element cannot be used for future on-device secure payment applications (e.g. EMV-based authorization of payment transactions in the mobile phone's web browser). Such applications

would, however, be one of the key benefits of having a secure element inside a mobile phone.

V. Reporting and Industry Response

We reported our findings and proposed workarounds to Google (and some of their Google Wallet partners) in April 2012. Google quickly acknowledged the problem and confirmed that they could reproduce the attack. Our tests in June 2012 revealed that new installations of Google Wallet (i.e. secure element applets provisioned in June) were no longer vulnerable to our relay attack setup. Further testing in September 2012 showed that users of older versions of Google Wallet are now required to update to the latest version. This forces existing users to receive the necessary fixes of the secure element applets. Therefore, we assume that Google Wallet users are no longer vulnerable to the relay attack scenario described in this paper.

VI. Analysis of the Relay-immune Google Wallet

Version 1.6 of the Google Wallet on-card component (installed with version 1.5-R79-v5 of Google Wallet in September 2012) is no longer vulnerable to the software-based relay attack setup described in this paper. The relay attack is inhibited by the fact that the select command fails for both MasterCard credit card applet instances (`A000000004 1010` and `A00000 0004 1010 AA539648FFFF00FFFF`) with the error code `6999`. Thus, access to the credit card applet from the application processor has been disabled as we suggested (cf. section IV-D3). The Proximity Payment System Environment can still be selected though both internal and external mode.

The on-card component can still only be selected through internal mode. It now returns its version number upon selection and some commands for interaction with it have slightly changed their parameters. The commands for switching between locked and unlocked state of the wallet are still the same. As a result, it is still possible to unlock Google Wallet and the credit card contained in it without PIN verification. Consequently, a malicious application could enable the credit card on the RF contactless interface even though Google Wallet is protected by a PIN that is not known to the malicious application.

VII. Conclusion

In this paper we examined the feasibility of the software-based relay attack based on the mobile contactless payment application Google Wallet. We analyzed the communication between the Google Wallet app and the secure element, as well as the interaction between a point-of-sale credit card terminal and the Google Wallet device. Then, we used this information to create a prototype relay setup. With this setup we could confirm that Google Wallet was indeed vulnerable to the software-based relay attack. On the one hand, the credit card applets in the secure element were not sufficiently protected from access through apps on the application processor. On the other hand, the PIN protection of Google Wallet can be bypassed as the on-card component does not verify the PIN itself but instead can be controlled by simple lock and unlock commands. Google responded to our finding by fixing the vulnerability to software-based relay attacks. However, bypassing the PIN is still possible with the current version of the wallet.

Acknowledgment

This work is part of the project “4EMOBILITY” within the EU programme “Regionale Wettbewerbsfähigkeit OÖ 2007–2013 (Regio 13)” funded by the European regional development fund (ERDF) and the Province of Upper Austria (Land Oberösterreich).

References

[1] M. Roland, “Applying recent secure element relay attack scenarios to the real world: Google Wallet Relay Attack,” Technical Report, arXiv:1209.0875 [cs.CR], Sep. 2012, http://arxiv.org/abs/1209.0875.

[2] A. Hoog, “Forensic security analysis of Google Wallet,” *viaForensics Mobile Security Blog*, Dec. 2011, https://viaforensics.com/mobile-security/forensics-security-analysis-google-wallet.html.

[3] C. Benninger, “A Brave New Wallet – First look at decompiling Google Wallet,” *Intrepidus Group Insight*, Sep. 2011, http://intrepidusgroup.com/insight/2011/09/a-brave-new-wallet-first-look-at-decompiling-google-wallet/.

[4] N. Elenkov, “Exploring Google Wallet using the secure element interface,” *Android Explorations*, Aug. 2012, http://nelenkov.blogspot.com/2012/08/exploring-google-wallet-using-secure.html.

[5] N. Elenkov, “Accessing the embedded secure element in Android 4.x,” *Android Explorations*, Aug. 2012, http://nelenkov.blogspot.com/2012/08/accessing-embedded-secure-element-in.html.

[6] N. Elenkov, “Android secure element execution environment,” *Android Explorations*, Aug. 2012, http://nelenkov.blogspot.com/2012/08/android-secure-element-execution.html.

[7] M. Roland, J. Langer, and J. Scharinger, “Practical Attack Scenarios on Secure Element-enabled Mobile Devices,” in *Proceedings of the Fourth International Workshop on Near Field Communication (NFC 2012)*, Helsinki, Finland, Mar. 2012, pp. 19–24.

[8] J. Rubin, “Google Wallet Security: PIN Exposure Vulnerability,” *zveloBLOG*, Feb. 2012, https://zvelo.com/blog/entry/google-wallet-security-pin-exposure-vulnerability.

[9] G. P. Hancke, “A Practical Relay Attack on ISO 14443 Proximity Cards,” Jan. 2005, http://www.rfidblog.org.uk/hancke-rfidrelay.pdf.

[10] Z. Kfir and A. Wool, “Picking Virtual Pockets using Relay Attacks on Contactless Smartcard,” in *Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks (SECURECOMM’05)*, Sep. 2005, pp. 47–58.

[11] L. Francis, G. P. Hancke, K. E. Mayes, and K. Markantonakis, “Practical NFC Peer-to-Peer Relay Attack Using Mobile Phones,” in *Radio Frequency Identification: Security and Privacy Issues*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6370/2010, pp. 35–49.

[12] L. Francis, G. P. Hancke, K. E. Mayes, and K. Markantonakis, “Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones,” Cryptology ePrint Archive, Report 2011/618, 2011, http://eprint.iacr.org/2011/618.

[13] M. Roland, “Software Card Emulation in NFC-enabled Mobile Phones: Great Advantage or Security Nightmare?” in *4th International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*, Newcastle, UK, Jun. 2012, http://www.medien.ifi.lmu.de/iwssi2012/papers/iwssi-spmu2012-roland.pdf.

[14] G. P. Hancke, K. E. Mayes, and K. Markantonakis, “Confidence in smart token proximity: Relay attacks revisited,” *Computers & Security*, vol. 28, no. 7, pp. 615–627, 2009.

[15] M. Roland, J. Langer, and J. Scharinger, “Relay Attacks on Secure Element-enabled Mobile Devices: Virtual Pickpocketing Revisited,” in *Information Security and Privacy Research*, ser. IFIP AICT. Springer Boston, Jun. 2012, vol. 376/2012, pp. 1–12.

[16] J. Rubin, “Google Wallet Security: About That Rooted Device Requirement...” *zveloBLOG*, Feb. 2012, https://zvelo.com/blog/entry/google-wallet-security-about-that-rooted-device-requirement.

[17] *EMV Contactless Specifications for Payment Systems – Book B: Entry Point Specification*, EMVCo Spec., Version 2.1, Mar. 2011.

[18] S. Höbarth and R. Mayrhofer, “A framework for on-device privilege escalation exploit execution on Android,” in *3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*, San Francisco, CA, USA, Jun. 2011.

[19] “Google Wallet – How it works – In-store,” http://www.google.com/wallet/how-it-works/in-store.html, Sep. 2012.