Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Reverse engineering Internet banking

Proyecto Final de Carrera

Ingeniería Técnica de Informática de Sistemas

Autor: Eduardo Pablo Novella Lorente

Director: Ismael Ripoll

04-09-2013

Reverse engineering Internet Banking

Eduardo Pablo Novella Lorente

Institute for Computing and Information Science
Digital Security Group

Radboud University Nijmegen
The Netherlands

A thesis supervised by:

Erik Poll and Joeri de Ruiter

2013 June

Abstract

This paper presents a security analysis of the online banking system of one of the most important banks of the Netherlands. New security devices have been designed in order to authorize transactions in a handy and user-friendly way. In doing so, it has been attempted to strengthen the online authentication system using hand-held smartcard readers connected by a USB-cable to a PC. These USB-connected smartcard readers are devices with a small display and numeric keyboard with two additional keys in order to accept or deny operations. The customers of Internet banking can perform any operation in a "secure way" with such devices.

In this document we will discuss different USB-connected smartcard readers from several Dutch banks. For *ABN-AMRO* we will discuss the smartcard reader called the *e.dentifier2* made by *Gemalto*, which we will principally focus on it in more detail, and for *ING* the reader *DigiPass 850*, made by *VASCO*.

We will focus on reverse engineering the ABN-AMRO's readers. We will verify that last attack [1] on those devices is not working in the new version of the *e.dentifier2* and we will also reverse engineer some additional functionalities,which were not considered in earlier research about ABN-AMRO e.dentifier2 reverse engineering [2], and currently do not seem to be used in ABN-AMRO's internet banking website. These additional functionalities [6] could be used for authentications and transactions in Internet banking in the future.

Acknowledgements

To my family and best friends ... Specially for some who will not stay with us anymore.

To the best teachers I ever had. For those who tried to motivate students in order to have fun with computers and education. I will name some of them but not all : José Verdoy, Jon Ander Gómez Adriá, Germán Moltó, Alicía Roca, Ismaell Ripoll and Juan Vicente Capella Hernández.

To my supervisors Erik Poll and Joeri de Ruiter for their support and for giving me a chance to get to know a bit about bank smartcards.

To my English teacher Karin Kessels for giving some corrections in this document.

And finally for my Faculties : Universidad Politécnica de Valencia and Radboud Universiteit Nijmegen.

Contents

Chapter 1

Introduction

1.1 Smartcard readers for Internet banking

Some important banks in the Netherlands are using new security devices in order to strengthen the authentication process with their customers. As usual, any bankcard always requires a PIN code as the essential authentication method in any transaction with the bank. However, banks are establishing new security policies, which try to offer more security in the challenges generated mainly by 3 factors: cardholders interaction, bankcard and own smartcard reader of the bank. In these cases the cardholder can log into the bank, sign payments, send transactions, generate secure codes and so on. Banks are attempting to give cardholders more insight into detail of transactions and involve them more in them.

Secondly, we will discuss two versions of ABN-AMRO USB-connected smartcard reader. We will talk about an older version and a newer one of the e.dentifier2. With the older version of e.dentifier2, a new security feature was presented as its main aim: "What You Sign Is What You See"(WYSWYS). This new signing method lets users can understand what they are signing with their devices in a friendly way. However, this was totally proven false in the paper *"Design to Fail: A USB-Connected Reader for Online Banking"* [1], where a design flaw in the protocol was exposed, where an infected PC clearly could give the go-ahead without waiting for the user to press $'OK'$. This flaw could allow the computer to choose transaction details and then carry out a bank transfer without confirmation by the end user. After this failure was discovered, readers could not be patched and ABN-AMRO got in touch with their supplier in order to develop a new version of e.dentifier2.

Thirdly, we will talk in detail about DigiPass 850, an ING USB-connected smartcard reader made by the vendor VASCO. This smartcard reader seems to be a new version of USB-unconnected DigiPass 800 but now this model can work with USB connection and without. We know that DigiPass 850 was showcased with the Belgian e-ID around 2004 and we guess that it is used for inside payments. According to the vendor this device "has multiple authentication methods as one-time passwords (OTP) and PKI infrastructure" and it is used with its own software called VACMAN.

Core objectives of this paper will be to investigate if they fixed the problem with the new version of e.dentifier2 and continue the reverse engineering with readers previously mentioned. Mostly we will focus on the e.dentifier2 where some functions of its plugin are still of unknown use and they will be investigated from a low-level overview in some detail. These functions are known in EMV-CAP specifications [6] as *Mode1* and *Mode2* and they are used for authentication in transactions although they are secret. Nowadays we do not see these functions interact with the plugin, but they could be activated in the future. We will analyze different protocols used in communications between e.dentifier2, host PC and smartcard with different tools discussed further.

Chapter 2

Background

In order to make the following chapters easier to follow, we will give some background on the EMV and EMV-CAP standards and some details about challenges-responses or one-time passwords. Moreover we will take a look how the USB protocol works. Also we will have a look at step-by-step several operations like login and send transactions using the USB-connected and unconnected mode of the e.dentifier2. This chapter is also offering important information on the main goal of the e.dentifier2, "What You Sign Is What You See", which is trying to avoid fraud in e-banking.

2.1 "Sign What You See"(SWYS)

A challenge-response authentication protocol is a security protocol that verifies an identity with a response to a challenge. This challenge is issued by a sender, which usually consists of a totally unpredictable random number, also known as nonce, and from this nonce the receiver must answer with a nonce and something else that shows who he really is. There are lots of different types of challenge-response authentication processes in the real world, but specifically in Internet banking, those nonces are random numbers of 8 digits with ABN-AMRO. If our bank wants us to authenticate, it will send a challenge to us and we will copy this number in the smartcard reader in addition to a PIN code and our smartcard reader will generate a response to this challenge; therefore this number is a proof that we are the cardholders and of course we have the correct PIN code.

The digital signature is an operation where a transaction is authorized or confirmed; the way to do this is by using these challenges-responses. Digital signatures can be done by cardholders, authorizing payments, or by smartcards giving responses towards the e.dentifier2 in order to verify the right way in the last operation computed by the smartcard. This data is signed using asymmetric cryptography, but they also use symmetric cryptography in order to generate *MAC (Message Authentication Code)*. Those codes are evidence to verify the source of messages. Smartcards and banks know those symmetric keys (usually *3DES* keys). The main goal is that the cardholder can control the signing of transactions in a handy way. For that a new feature was included in the e.dentifier2, called by the vendors: *"What You Sign Is What You See" (WYSIWYS)*, also called *"Sign What You See"(SWYS)*. This technique tries to prevent Man-in-the-Browser attacks. Those attacks infect web browsers in order to evilly modify transactions in the host. Due to those attacks, USB-connected smartcard readers can be more secure because the smartcard reader displays transaction details that the cardholder can understand and decide if he wants to accept or deny it, so the cardholder does not just see meaningless random numbers. So apparently it would be more difficult to exploit these sorts of attacks with this new USB-connected design. This new system tries to defeat Man-in-the-Browser attacks definitively.

2.2 The USB Protocol

In this section we will discuss background information about the USB protocol quickly in order to better understand some later sections. First of all, we can say that USB devices are classified as a hub or a function. Hubs provide additional attachment points, whereas functions provide capabilities to the system. In a USB system we can highlight 3 main parts:

- *USB Host*. There only exists one in the system. A USB host has a root hub which offers more ports. A cardholder will be able to connect the e.dentifier2 to any free port of the USB host (PC of the cardholder).
- *USB interconnect*. This provides a connection from USB device(s) to USB Host.
- *USB devices* (hub or function). A hub simply offers additional ports, and a function offers capabilities to the system. The e.dentifier2 will be a USB device

connected to this hub in order to provide e-banking services to customers of ABN-AMRO.

A USB communication is created between these parts creating a data flow. In this communication we can also highlight some parts such as:

- *Endpoints.* They are unique addressable points in devices. This endpoints have different characteristics like frecuency, latency, bandwith, number identifier, transfer type and so on. They can be In/Out depending on its direction. "In" is from the USB device (e.dentifier2) to the USB Host (PC). "Out" is from USB host to the USB device.

- *Pipes.* These are associating endpoints with the USB host. They are the way to communicate USB host and endpoints.

Briefly, we can have a look at the four basic transfer types:

- *Control.* These lossless transmissions are used for the configurations.

- *Bulk data* . Sequential lossless transmissions are used to transfer a large amount of data.

- *Interrupt data* . Transmissions need priority over others. Typically, a mouse or pointers.

- *Isochronous data* . Transmissions for streaming data. This can have loss but also have priority on the USB bus.

The e.dentifier2 is a USB device and a USB host can be the cardholder's computer. The e.dentifier2 will be able to create communications over USB-cable thanks to endpoints, where it will be sent data over pipes in both directions (In/Out). This means e.dentifier2 to PC and vice versa. Mostly in the cases of the USB traffic, which has been eavesdropped in this research, it was bulk data.

2.3 EMV

EMV stands for Europay, MasterCard and Visa. EMV is the standard used for banking smartcards. In this standard is defined in detail the interaction at the physical, electrical, data and application levels between banking smartcards and terminals (eg. ATMs and point-of-sale terminals).

2.4 EMV-CAP

The Chip Authentication Program (CAP) is a specification for using EMV banking smartcards for Internet banking. It started as an initiative of MasterCard, and later for Visa. This specification defines a handheld device, also known as CAP reader or EMV-CAP reader, with a smartcard slot and a numeric keypad capable of displaying between 8 and 12 characters.

In EMV-CAP, when the user signs an operation from a EMV-CAP reader, a couple of cryptograms are created as proof of authorization by the smartcard. Those cryptograms are called *Application Cryptograms* (AC) and we can distinguish the following:

- *ARQC (Authorization Request Cryptogram).* This is the response from smartcard against the challenge sent by the e.dentifier2.
- *AAC (Application Authentication Cryptogram).* This second cryptogram is a step more in order to correctly complete the transaction.

Although EMV is not public it has frequently been reverse engineered [5], specifically the EMV-CAP standard used in the e.dentifier2 is still unknown. Some internal functions are creating a kind of "black box" with data, which are totally "mangled" in the output.

2.5 e.dentifier2

In this section we will discuss ABN-AMRO's reader, also known as the e.dentifier2, which is a hand-held smartcard reader that has a small display and numeric keyboard with two additional keys in order to accept or deny operations. This device can work in USB-connected or unconnected mode. However, we will just focus on USB-connected mode in this paper.

It seems that several versions have been released. For finding out the information which version is running in the device we just keep the key ≪ 5 ≫ held in the keypad and insert any smartcard. We think that there are only two versions of e.dentifier2 but we do not know how many versions are running nowadays. Specifically in this paper we are working with the two following versions:

- *e.dentifier2 F/W 01.02 H/W C Dec 19* **2007** *18:39:42* . This will be called old version or old e.dentifier2 in this document.
- *e.dentifier2 F/W 01.05 H/W C Feb 07* **2012** *14:54:39* . This will be called new version or the improved e.dentifier2 in this document.

The manufacturers of the smartcard reader of ABN-AMRO claim the e.dentifier2 is "the most secure sign-what-you-see end-user device ever seen" [1]. In this document we will discuss this.

2.5.1 Connected mode

In this section we deal with two operations: login and signing transaction. Any transaction can be done if the cardholder is previously logged in. In a login, the e.dentifier2 will reply with a response in order to be able to login to the website. In a transaction, a challenge and response will play a role in the operation on both sides of the communication. Those operations will be outlined in the following point. They were already explained in detail in other research [2].

2.5.1.1 Login

The cardholder wants to log into the Internet banking system using e.dentifier2 connected over USB-cable. Hence, these are the main steps:

1. Cardholder plugs his e.dentifier2 into a free USB port in his system.
2. If the driver is not installed then the cardholder installs e.dentifier2 drivers [2] in his system. A plugin is installed in the web browser and other files to recognize the USB-reader in the operativing system.

[1] `http://www.gemalto.com/financial/ebanking/about/case_studies/ABN_AMRO.html`

[2] Drivers available only for Windows and MacOS systems

3. Cardholder inserts bankcard into e.dentifier2.
4. Cardholder visits login ABN-AMROs website.
5. Web browser starts an SSL-TLS session between bank and cardholder PC.
6. Bank website reads the account number from the smartcard using JavaScript program called BECON.js[1]. This is also checking if a card is inserted and if the e.dentifier2 is ready. A couple of functions in this Javascript program will be called: *CheckConnection()* and *CheckCard()* [2] .
7. If everything is right, e.dentifier2 asks the cardholder for PIN code
8. Cardholder types PIN code in the e.dentifier2.
9. E.dentifier2 lets the smartcard verify the PIN code for this account number.
10. If the PIN code is okay, the e.dentifier2 returns a response to the website.
11. The bank verifies if the responses match because it calculates its own response.
12. If the responses match, then the cardholder is logged in.

2.5.1.2 Signing transaction

The cardholder wants to make a transaction in the Internet banking system using e.dentifier2 connected over USB-cable. Hence, these are the main steps:

1. Cardholder is already logged in.
2. Cardholder wants to make a payment in the website and he orders the transaction in a web form.
3. The JavaScript program, BECON.js, calls to *GetResponse(p_sSignData)* where the input is data with details of the transaction by the bank. This data is sent to e.dentifier2 over USB traffic.
4. The e.dentifier2 asks the cardholder for PIN code

[1] `https://www.abnamro.nl/en/logon/generic/scripts/BECON.js`

[2] These functions will be commented further in the next chapter.

5. Cardholder types PIN code in the e.dentifier2.

6. E.dentifier2 lets the smartcard verify the PIN code for this account number.

7. If the PIN code is okay, then the e.dentifier2 shows a text in the display's screen with the amount received by the bank, where the cardholder can be accepted or denied the transaction in the display of the e.dentifier2 pressing *'OK'* or *'C'* , which means Cancel.

8. If the user presses 'OK', the e.dentifier2 sends a cryptogram to the smartcard using a challenge-response with *SignData*, the smartcard also returns a cryptogram that is converted for the e.dentifier2 in a response. This response is sent to the bank as proof of the fact that user want to accept this transaction.

9. The bank can verify if responses match because it calculates its own response.

10. If they are identical, then the payment is carried out.

2.5.2 Unconnected mode

We will also discuss the unconnected mode, but it is not really relevant for this thesis. The cardholder wants to log into the Internet banking system using unconnected e.dentifier2. Hence, these are the main steps:

1. Cardholder visits login ABN-AMROs website.

2. Cardholder inserts card and account number.

3. Web browser starts an SSL-TLS session between bank and cardholder PC.

4. A JavaScript code called BECON.js is checking if card and account number exits and are correct.

5. If everything is right, ABN-AMRO starts a challenge-response, and the website shows a challenge in the web browser.

6. Cardholder inserts bankcard into e.dentifier2 and presses login key.

7. E.dentifier2 asks to the cardholder for PIN code.

8. Cardholder types PIN code in the e.dentifier2.
9. E.dentifier2 internally checks PIN code for this account number.
10. If PIN code is okay, e.dentifier2 shows the response number.
11. Cardholder enters this response in the ABN-AMROs website.
12. Cardholder is logged in.

2.6 DigiPass 850

Other banks like ING Direct have also designed USB-connected readers. The company VASCO was responsible for creating these readers for ING. In this paper we started to research the version DigiPass 850 [1]. Nowadays these readers are not applied yet in the real life for regular ING customers but they seem to be used for business payments [2].

A main aim for ING is also to prevent the PIN code leakage over PC or internet like ABN-AMRO tried when they made its reader. Also it tries to achieve a more user friendly mode, avoiding having to type bank account numbers, challenges and so on. The manufacturer VASCO remarks the use of DIGIPASS: *"a propietary system of two-factor authentication in the cloud"*, as new feature of these devices. In this document we have attempted to find out how USB communications work in these devices and how protocols are working on them.

[1] `http://download.alsoft.cz/vasco/pdf/Digipass_Desk_850.pdf`

[2] `https://start.inginsidebusiness.com/EAILogonWebApp/EAILogon.jsp`

Chapter 3

Tools

In this section we are going to talk about every tool that we used to achieve information about the system. It is very important if we have a big picture about scenario. Therefore let's have a look in the following picture to understand better next discussions:

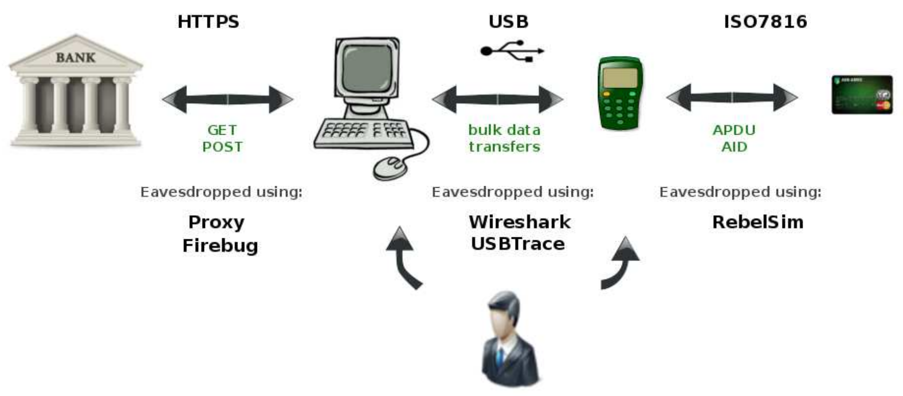

Figure 3.1: Real scenario - Big picture about possible eavesdropping points

3.1 USBTrace

This program was easy, comfortable and quick to use because you could watch raw USB traffic in a single packets. After using this program, USB commands findings were very useful to write our own driver to replay USB bulk data over e.dentifier2. We noticed that all packets were multiple of 8. So communication is always multiple of 8 bytes.

It was used to eavesdrop USB raw traffic from DigiPass 850 and it was observed multiple of 6 bytes. Due to using other system to interact with the smartcard reader, not many USB commands have been retrieved.

This tool was run in a Windows machine.

3.2 Wireshark

To sniff USB communications we also use Wireshark. The Operating System and this tool "convert" the raw USB packets into the network traffic. So it works as a network interface. In addition a special configuration is required if we want to properly intercept USB packets and it is that we need at least *libpcap* 1.0.0 in our machine. It was another overview about USB taffic, with much more information that USBTrace. Fortunately this tool can be launched in console mode (tshark) using filters to automate the research.

For instance, some filter like this was used for capturing USB commands:

```
usb.capdata && usb.endpoint_number==2 && usb.device_address
```

Being "2" the numerical identifier of the endpoint of the e.dentifier2 and the field 'capdata' contains information about the USB traffic.

It was used to find out USB commands in a Linux and Windows machine.

3.3 RebelSim & Realterm

This hardware interface allowed the passive monitoring of data between the e.dentifier2 and the smartcard. This kit is offered like *"Scanner tool of APDU commands from GSM/UMTS Simcard for Analysis"* and it was used with *RealTerm* software [1]. Any smartcard can be exposed to Man-in-the-Middle if we are able to achieve its baud rate. Some previous steps were configured in order to sniff APDU commands. The baud rate was established to 5200 , and the RealTerm output was dumped in hexadecimal format

[1] `http://sourceforge.net/projects/realterm/`

in a text file. [2]

Once we got raw APDU buffer and with help of EMV specifications, we were able to discover the *applet ID* (AID),*SELECT* operations, *GENERATE_AC* and their payloads and traces. They will be discussed in the next chapter.

It was used to find out APDU traffic in a Windows machine.

3.4 JavaCard bankcard

The JavaCard smartcards allow applets (Java Applications) to be run securely on smartcards. These smartcards are based on the Java Card Platform specifications developed by Sun Microsystems and they can be programmed using JavaCard API. When a smartcard is inserted in the e.dentifier2, it will try select certain applet ID (AID). This AID is hard-coded in the device and it is first EMV operation together with get ATR [1]. ABN-AMRO's AID was found out using RealTerm being able to develop our own applet[2] with identical AID. Some functions were disabled as : *PIN verification* or *Application Transaction Counter* (ATC).

We used these kind of smartcards:

- JCOP41 V2.3.1, SmartCard 72 Kbyte EEPROM, Dual Interface [7].

With these smartcards was possible to emulate a real card and it was used to achieve replaying attacks and other functions in the e.dentifier2.

3.5 Firebug

As we know, the e.dentifier2 is connected over USB cable to host PC. This host PC needs a web browser in order to communicate with the Bank website. This web browser is using a JavaScript file when user's PC is visiting logon Bank website. Through those JavaScripts functions, it will be possible to start a communication. This Javascript code will be responsible to verify bank account number with the bank, if the e.dentifier2 is

[1] `http://en.wikipedia.org/wiki/Answer_to_reset`

[2] Proof of concept of an EMV applet simulating an ABN-AMRO card by Joeri de Ruiter [1].

connected, if the connection is correct, version of driver and plugin and so on.

We noticed that a HTML embedded object [1] was working as plugin to communicate with the e.dentifier2, this object was responsible to intercept requests and send them to smartcard reader and it sends the response back.

It was very useful to use a Javascript debugger to find out some USB instructions and the order of operations. It was used *Mozilla Firefox* with the Firebug Add-on installed in it. The JavaScript file that was debugged, **BECON.js**, resides in the Bank website but it is loaded in user's web browser.

Further information about these JavaScript functions will be discussed later.(Chapter 5)

3.6 Own webpage

In order to make reverse engineering of some JavaScript functions was created our own HTML page, which called to JavaScript functions of *BECON.js* to retrieve information. Our own HTML page was combined with a couple of tools mentioned previously. First of all, we loaded our own HTML page to run these JavaScript functions and we activated these tools : *USBTrace* eavesdropping the USB commands with the e.dentifier2 and *RealTerm* sniffing APDU traffic with smartcards. This way makes sure a complete session how the information is being encapsulated with different protocols.

As we explained before in the introduction, this web page was created to find out how *GetMode1Response* and *GetMode2Response* were internally working. These functions are defined in EMV-CAP standard [6] although their specifications are secret, so we think they are executed to do authentication and signing transactions using cryptograms. With this web page we were able to send our crafted input values and investigate further.

Although we were focused on two functions, the rest of functions were also implemented in our own web page. We could observe how every function worked using *Firebug*, *USBTrace* and *RealTerm* at the same time.

Further information in the reverse engineering section.(Chapter 5)

[1] `embed id="PLUGIN_BECON" hidden="true" type="application/BECON-PlugIn" name="PLUGIN_BECON"`

3.7 Operating Systems

A couple of Virtual Machines were used to manage those tools. We did not choose last Operating Systems due to the fact that we do not need them in order to achieve our goals in reverse engineering. Two Operating Systems well-known are *Ubuntu 10.04LTS* and the old *Windows XP sp3*.

3.7.1 Ubuntu 10.04LTS 32 bits

It was also important use a Linux system for using Wireshark in mode USB-sniffing. Python code was run in Linux system because it is more comfortable than Windows for these purposes even because libraries to create our own USB driver: *liusb* and *PyUSB* were not stable yet in Windows. These libraries were uninstalled and installed manually to get a stable environment. For instance, *libusb* and *PyUSB* were updated to the last version avoiding repositories of the Operating System. Moreover *libpcap*, a library to intercept traffic in the Operating System, was updated to *libpcap* 1.3.0 to manage *Wireshark* with USB sniffing support.

3.7.2 Windows XP sp3 32 bits

It was very important to use a Windows system for e.dentifier2's drivers. We know that drivers are only available for MacOS and Windows systems. But thinking of interception of USB traffic with *Wireshark*, I preferred using Windows because MacOS didn't have any reference at *Wireshark* Wiki[1]. Moreover this machine was also used to realize Man-in-the-Middle between e.dentifier2 and smartcard, in this case the software *RealTerm* was used with the following hardware: "RebelSim APDU scanner".Finally a lot of time was spent using Windows so it were installed the libraries *PyUSB* and *libusb-win32* in order to use all in the same machine.

So it was able to get the following actions with this machine:

- Logging and eavesdropping at ABN-AMRO with USB-connected reader with Javacards.
  In this part, USB traffic and their payloads were analyzed with details. And they

[1] `http://wiki.wireshark.org/CaptureSetup/USB`

were important to understand some actions with other commands or APDU-commands sent from the smartcard.

- Replaying USB-traffic with *PyUSB*.
  Thanks to a proof of concept written in Python code [1], it was able to attack the main goal of e.dentifier2 [1] , SWYS (”Sign What You See”). This proof of concept just worked in the old versions of e.dentifier2.

[1]`http://www.cs.ru.nl/~joeri/`

Chapter 4

The improved e.dentifier2

A important question is now treated :

What will it happen with new version of e.dentifier2? How did they fix this big problem?

4.1 Background : Old attack

As we explained before in the introduction, the main goal of the design of SWYS protocol("Sign What You See") in the old e.dentifier2 was principally provide to clients more security in operations. For that, their own reader would be a right solution in order to defeat any kind of Man-in-the-browser or clientside attacks. This protocol was designed wrongly and it was widely reverse engineered [1] finding a critical design error. This flaw was that the e.dentifier2 sent a message to PC informing that the cardholder had pressed 'OK' (Step 7 in the diagram 4.1 SWYS protocol). This is clearly incorrect because an infected PC could press OK itself and inmediately generate a cryptogram (Step 8 in the diagram 4.1 SWYS protocol) accepting the transaction. In this way, the main goal the e.dentifier2 was violated because the cardholder was unable to decide for the transaction. Below we can have a look at SWYS protocol, diagram 4.1, and the critical error is highlighted with bold.

The e.dentifier2 never had to communicate the user action of pressing OK. The e.dentifier2 should have been designed to detect when the cardholder was really pressing OK and in this moment, when cardholder pressed OK, then ask to smartcard for a cryptogram. In this way would achieve main goal of SWYS protocol protecting to the cardholder of a PC infected with malware.

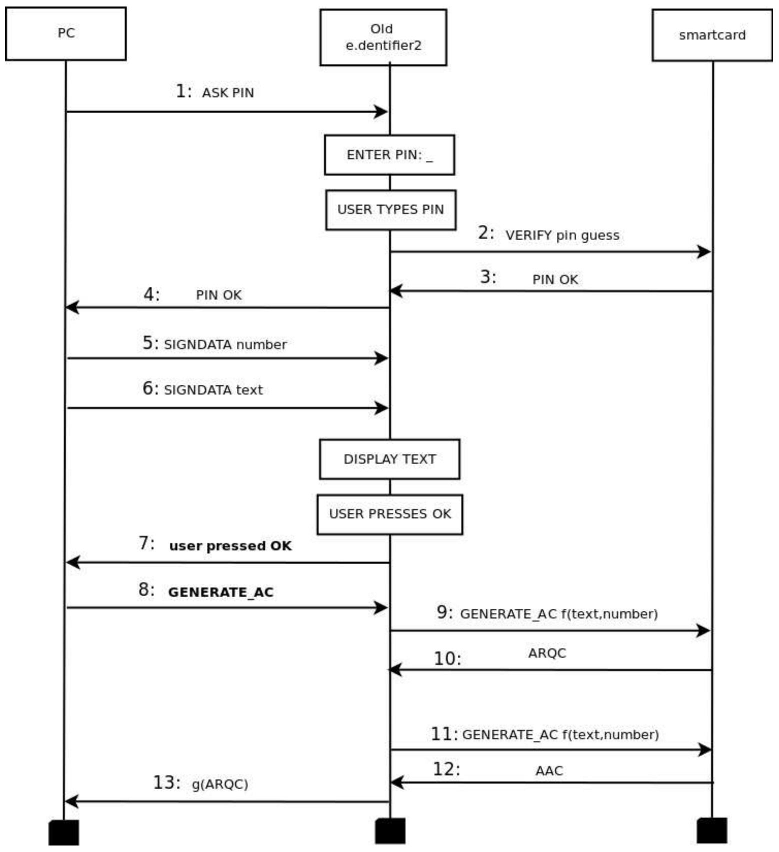

Figure 4.1: SWYS protocol. - Diagram for login and transactions in the old e.dentifier2 [1].

4.2 How was fixed the vulnerability? : New e.dentifier2

Once that ABN-AMRO watched the severity of this problem, their manufacturer designed a new e.dentifier2 fixing the vulnerability. This new e.dentifier2 has been tested against this attack[1] again. Now, the new e.dentifier2 knows if the user really pressed OK or not. Verifying an internal state into the e.dentifier2 which show if it was pressed by the cardholder. This internal state was not checked in the old version of e.dentifier2. A simple error that was not detected in testings being maybe, the most obvious attack in the SWYS protocol. We do not know if such internal state existed or not anytime in the old e.dentifier2 and it was not used or it was simply forgotten.

This new version asks to PC for generating a cryptogram to start a challenge-response but the e.dentifier2 is able to recognize if the cardholder pressed OK himself. Actually, the new version aborts any following operation if this internal state does not show that the cardholder signed the transaction with OK. This means, if we observe this patch in the following messages (Steps: 7 and 8 at diagram 4.2), that the e.dentifier is now able to recognize if such internal state has been activated to go ahead with the transaction, otherwise it will always take the default mode to abort a operation: Cancellation (Caldholder pressed OK). This default mode is also activated for a timeout in the e.dentifier2. This timeout will be activated, for instance, if the cardholder forgets his e.dentifier2 with the displayed text in the screen waiting OK/C for the user.

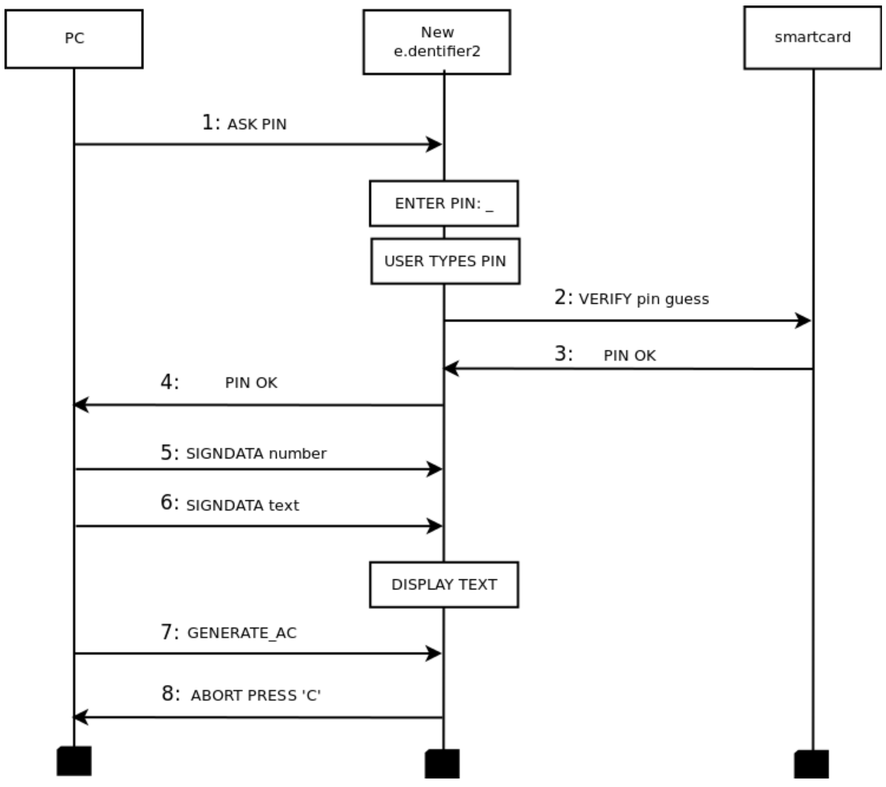

Figure 4.2: Attack to SWYS protocol. - Diagram for the attack on the new e.dentifier.

The following diagram 4.3 could have been a proper design of SWYS protocol in the old e.dentifier to defeat whatever attack of this sort. Unfortunately the new design of patched SWYS protocol was established in the new version after 5 years being vulnerable (As we discussed a couple of versions between 2007-2012). So far, we have not found more attacks in this protocol, but it should be analyzed further. An improved version of SWYS protocol could have been if the e.dentifier2 does not send a message to PC confirming that the cardholder pressed OK. So, we know the e.dentifier2 has an internal state, which shows if OK was pressed by user or not. Therefore, steps 7,8 in Diagram 4.1 or 4.2 should have completely been disabled. The PC does not need to know if the user pressed OK or not yet. Hence, below we offer a possible improved design of SWYS protocol. Unfortunately, this new design could not be deployed because old e.dentifier2 had to keep working despite vulnerability.

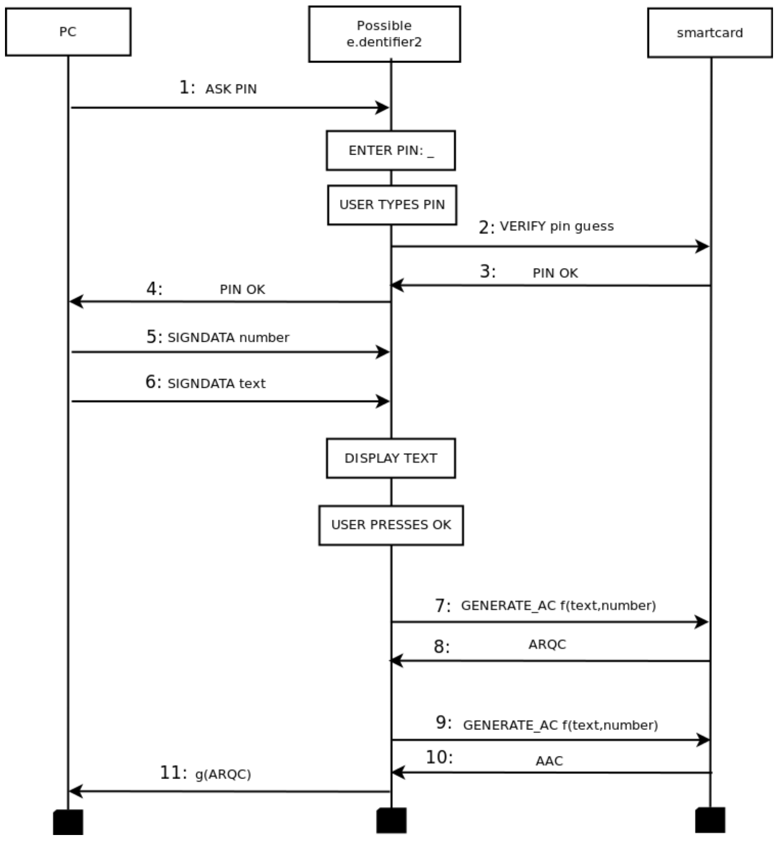

Figure 4.3: How SWYS should have been designed. - Diagram of SWYS improved protocol.

Chapter 5

Reverse engineering of additional functionality

In this section we will discuss protocols with a low level overview. We will sum up all USB instructions that we found on the both e.dentifiers. But also we will observe possible traces of USB traffic depending of the kind of EMV card, cardholder's decision to accept or deny transactions, changing input values in transactions in order to investigate what is happening and so on.

We will focus on two unknown functions and probably not used nowadays, which were not reverse engineered in the others research [1-2]. This functions were found in JavaScript program resident in ABN-AMRO website as we explained earlier. These functions could be used in order to login and sign transactions. Both functions will be named as *GetMode1Response* and *GetMode2Response*. Every function or trace will be discussed below or above of its corresponding table. Some tips will be explained later to understand the following traces of USB traffic.

5.1 Cheat sheet USB commands-responses

To be able to follow better this research, a summary of USB commands and responses can be checked along this document anytime. Some USB traffic has not been reverse engineerd yet, although we think that it has not been relevant to understand core aims of protocols.

Table 5.1: Cheat sheet of USB commands in the e.dentifier2
The following table lists USB commands sent to the smartcard. Let us realize as just first 4 bytes are constructing the USB command. Some functions were not clear enough to give a description.

USBcommand	Description
00 02 6E 6C 00 00 00 00	SET LANGUAGE NL(6E6C)
00 02 65 6E 00 00 00 00	SET LANGUAGE EN(656E)
02 01 00 00 00 00 00 00	GET ATR
02 09 00 00 00 00 00 00	SHOW SHIELD IN DISPLAY
02 0B 00 00 00 00 00 00	Channel is ready?? Detected after functions
01 03 01 02 00 00 00 00	SHOW PICTOGRAM : INSERT CARD
01 03 02 00 00 00 00 00	Card inserted??? Detected in CheckConnection
01 03 03 00 00 00 00 00	??UNKNOWN Detected in CheckCard
01 03 04 00 00 00 00 00	ASK PIN
01 03 05 05 00 00 00 00	SEND SIGNDATA-DATA-login
01 03 05 16 00 00 00 00	SEND SIGNDATA-DATA-transaction
01 03 05 46 00 00 00 00	SEND SIGNDATA-TEXT to display
01 03 06 00 00 00 00 00	GENERATE CRYPTOGRAM
01 03 07 01 00 00 00 00	GETMODE2RESPONSE login
01 03 07 17 00 00 00 00	GETMODE1RESPONSE transaction
01 03 08 15 00 00 00 00	LAST LOGIN MESSAGE ? Guess

Table 5.2: Cheat sheet of USB responses in the e.dentifier2
The following table lists USB responses from smartcard to e.dentifier2. Let us realize as just first 4 bytes are constructing the USB responses. Also we can watch the payload attached to commands, where it is marked with XX can go different values depending of data.

USBresponse	Description
00 01 25 01 00 00 00 00	Communication error II
01 01 01 01 00 00 00 00	Card inserted
01 03 01 01 00 00 00 00	Card inserted
01 03 02 00 00 00 00 00	User pressed OK in ASK PIN
01 03 03 04 00 00 00 00	User pressed OK in Mode1-Mode2Response
01 03 03 XX 00 00 00 00	Successful operation
01 03 05 06 00 00 00 00	Card inserted? Future work
01 03 07 00 00 00 00 00	User presses Cancel in ASK PIN,Mode1-Mode2
01 03 08 01 00 00 00 00	Communication error I
02 81 00 00 00 00 00 00	Unknown
02 81 01 00 00 00 00 00	Unknown

5.2 Traces of USB traffic

In this section we will have a look at USB traffic using the USB-connected e.dentifier2. For those data I have used several bank cards to find out extra information. The following cards were used in order to extract information:

- RaboBank Card (Netherlands). Type: Maestro.
- JavaCard. Type: Emulating Maestro of ABN-AMRO.
- Bankia Card (Spain). Type: VISA Electron.
- NovaGalicia (Spain). Type: MasterCard.

Before the next tables some clarifications about how I have tried to show my research must be explained a bit:

- Out: PC sends to e.dentifier2. It is called USB-command and we observe that they always start with : *02 XX* or *01 03*.
- In: e.dentifier2 sends to PC. It is usually called USB-response in the tables.
- Payload: Data will be attached in the commands. Usually *00 06 ......* and ending with *00 0X* where X can be 2,3,4,5.
- XX : Variable Bytes.
- ?? : It is not sure.
- .... Snipped. Look at payload size to understand that.
- End: Possibly payloads are ending with this line, that means the end of payload. It could be that second byte in the last 8 bytes of a USB response, that could mean the length of payload's data which are remaining. But this assumption is not sure.
- Possible changes of bytes depending of Bank card type and company. Normally in bold, and marked with MasterCard, VISA or others.
- RESPONSE1 or RESPONSE2. We have an if-else condition. Normally when the cardholder presses OK or Cancel.

Table 5.3: SHOW LOGO IN DISPLAY and SET LANGUAGE.
These USB instructions can be individually each other.Although they have been detected for going together mostly of time. `SHOW SHIELD IN DISPLAY` is displaying the logo of ABN-AMRO, in the screen of the e.dentifier2. In this manner, the language can be set to English(EN) or Dutch(NL) without altering USB responses, which are always fixed for both languages if these two USB commands are sent together. Also the USB command `SET LANGUAGE` has been seen with other USB command (`SHOW INSERT PICTOGRAM`) in the JavaScript function *CheckCard()* of *BECON.js*.

This USB traffic has been detected trying to do authentication in the ABN-AMRO's website and in the beginning of any operation as login or transaction.

USBcommand	Description
Out: 02 09 00 00 00 00 00 00	SHOW SHIELDS IN DISPLAY
Out: 00 02 **6E 6C** 00 00 00 00	SET LANGUAGE **NL** = ascii(6E6C)
...... or	
00 02 **65 6E** 00 00 00 00	SET LANGUAGE **EN** = ascii(65 6E)
USBresponse	**Description**
In: 01 03 01 01 00 00 00 00	Response
In: 00 01 01 01 00 00 00 00	Response

- Filling means that sometimes the command fills 'XX' with *0x20* or *0x00*. Depending if it binary data or it is ascii text in order to display in the e.dentifier2.
- Ex: A example of trace or normal values.

Table 5.4: GET ATR (Answer To Reset).
This USB command will ask for the ATR to smartcard and this will answer with its ATR. The USB response will contain the ATR between third and eighth byte in the USB response. The ATR has been marked with XX. Also we found some variations in the last response to end the response in the second byte. It has been highlighted with bold showing different kinds of EMV cards.

This command was detected along many operations interacting with the driver due to its importance. Any operation with the bank requires this command previously.

USBcommand	Description
`Out: 02 01 00 00 00 00 00 00`	`GET ATR`
USBresponse	**Description**
	`24 bytes (4x8) for ATR (X)`
`In: 00 06 XX XX XX XX XX XX`	`3B......`
`In: 00 06 XX XX XX XX XX XX`	
`In: 00 06 XX XX XX XX XX XX`	
`In: 00` **03** `XX XX XX XX XX XX`	`Ending Maestro`
	`00` **02** `90 00 ... Ending VISA?`

Table 5.5: Card inserted?
This USB traffic is not clear yet. For that, we use an interrogation '?' in its name. We have shown both responses depending if the card was inserted (RESPONSE1) or not(RESPONSE2). If the card was not inserted,then we saw that a lack of 8 bytes in the USB response. This lack are the first 8 bytes in the USB RESPONSE1, and the result of that is RESPONSE2.

Another variation that we found was the seventh byte of USB response marked with bold in the tables. This byte can change depending of kind of EMV card which is inserted. Actually, we do not know why this byte is changing sometimes. Below we can watch as this value can change without a known pattern.

USBcommand	Description
`Out: 01 03 02 00 00 00 00 00`	`CARD INSERTED?`
USBresponse	Description
`In: 01 03 05 06 00 00 00 00` In: 00 06 30 31 2E 30 **35** 00	RESPONSE1: 2x8 bytes Card was inserted! VISA,Maestro,Mastercard Highlighted byte could change:**32** (MasterCard)
In: 00 06 30 31 2E 30 **35** 00	RESPONSE2: 8 bytes No card inserted!

Table 5.6: SHOW INSERT CARD PICTOGRAM and SET LANGUAGE
These USB commands are very similar to `SHOW SHIELD IN DISPLAY` and `SET LANGUAGE`. To read further information about `SET LANGUAGE` can be looked up at table 5.3. A language is needed for the e.dentifier2 in order to send messages towards the smartcard. If language bytes are set to *00 00*, the e.dentifier2 will use Dutch language by default.
The difference with this command is the information displayed in the screen of e.dentifier2, this command shows a pictogram asking for a smartcard. Identical response has been observed in this USB traffic. Let us realize that if the smartcard is not inserted, the e.dentifier2 will wait any smartcard to continue. The USB response is only sent after a smartcard has been inserted.

USBcommand	Description
Out: 01 03 01 02 00 00 00 00	SHOW INSERT CARD PICTOGRAM
Out: 00 02 **6E 6C** 00 00 00 00	SET LANGUAGE: NL (6E 6C)
	...... or ....
00 02 **65 6E** 00 00 00 00	SET LANGUAGE: EN (65 6E)
USBresponse	**Description**
In: 01 03 01 01 00 00 00 00	Response Card inserted 1
In: 00 01 01 01 00 00 00 00	Response Card inserted 2

Table 5.7: ??UNKNOWN.
Totally unknown for us. This USB command was detected during a login session with ABN-AMRO's website. A couple of bytes have been observed for their variation. The first of them, is the fourth byte of first 8 bytes of USB response (It is highlighted with bold and value 0B). This byte was observed with close values (0B and 0C). The other bytes highlighted mean the length of data in the octet of USB commands. Below we also observe as different measures with VISA, MasterCard and Maestro. The seventh byte could be length of the payload, but this is not sure.

USBcommand	Description
Out: 01 03 03 00 00 00 00 00	??UNKNOWN
	RESPONSE: 3x8 bytes
In: 01 03 04 **0B** 00 00 00 00	Maestro,VISA,MasterCard can change seventh byte
In: 00 06 XX XX XX XX XX XX	
In: **00 06** XX XX XX XX XX XX	Maestro
00 05 XX XX XX XX XX XX	MasterCard,VISA

Table 5.8: TRANSACTIONS: SEND SIGNDATA-DATA.
Send data to e.dentifier2 in a transaction in order to sign. Signed data of transaction will be attached in the payload (Marked with XX in the table) of the USB command. This USB command was widely reverse engineered at Arjam Blom's thesis [2].

USBcommand	Description
Out: 01 03 05 16 00 00 00 00	SEND SIGNDATA-DATA transaction PAYLOAD: ??x 8bytes Usually 4x8bytes.
Out: 00 06 XX XX XX XX XX XX	Data to send
Out: 00 06 XX XX XX XX XX XX	Data
....	
Out: 00 06 XX XX XX XX XX XX	Data
Out: 00 04 XX XX XX XX XX XX	End of data .Filling with 0x00

Table 5.9: LOGIN: SEND SIGNDATA-DATA.
Send data to e.dentifier2 from PC in login operations. This USB command was widely reverse engineered at Arjam Blom's thesis [2]. Not relevant for the attack [1].

USBcommand	Description
Out: 01 03 05 05 00 00 00 00	SEND SIGNDATA-DATA login

Table 5.10: ASK PIN
Ask for the PIN code of 4 digits to cardholder. The cardholder will enter his PIN code and he will be able to press two options: the key 'OK' (RESPONSE1) or 'C'(RESPONSE2). Both USB responses are showed below.

USBcommand	Description
Out: 01 03 04 00 00 00 00 00	ASK PIN
USBresponse	Description
In: 01 03 02 00 00 00 00 00	RESPONSE1: OK
In: 01 03 07 00 00 00 00 00	RESPONSE2: CANCEL

Table 5.11: SEND SIGNDATA-TEXT
Send text from PC to e.dentifier2 to be signed in it. When a transaction occurs, first it is sent a USB command with data of the transaction(`SEND SIGNDATA-DATA-transaction`), and the next instant is sent a USB command with a text to be displayed in the e.dentifier2 (`SEND SIGNDATA-TEXT`). Now the cardholder can decide if he wants to authorize the transaction, let us remember that now the cardholder is seeing the e.dentifier2's screen and he can understand the message sent from PC with details of transaction. This step can be checked in more details in the Chapter 4, figures 4.1, 4.2 and 4.3. Then the cardholder could press 'OK' or 'C'. From here, the attack to SWYS protocol [1] started pressing OK instead of the cardholder without waiting for a response from e.dentifier2 towards PC. In this moment, the text disappeared in less than one second and e.dentifier2 and smartcard created a cryptogram authorizing the transaction without the cardholder's participation. This cardholder's decision, the main goal of SWYS, is only vulnerable in some millions of old e.dentifier2.

The payload for this USB command is established to 12x8 bytes covering full screen on the e.dentifier2. A possible payload asking for pressing 'OK' can be this: `00 04` **4F 4B** `20 20 74 20`. That means, the key 'OK' is encoded in hexadecimal as `4F 4B`.

USBcommand	Description
Out: 01 03 05 46 00 00 00 00	SEND SIGNDATA-TEXT
	PAYLOAD: 12x8bytes aprox.
Out: 00 06 XX XX XX XX XX XX	Text to be displayed
Out: 00 06 XX XX XX XX XX XX	Text to be displayed
....	
Out: 00 06 XX XX XX XX XX XX	Text. Fill 0x20 (Blank space)
Out: 00 04 XX XX XX XX **74** XX	End of text

Table 5.12: GENERATE_AC
This USB command will attempt to create a cryptogram (Step 7 in the Figures 4.1,4.2 and 4.3). As we explained earlier in the background chapter, a cryptogram (*Application Cryptograms (AC)*) can be an : ARQC and AAC (Check Chapter 2 for further information). This command is sent to e.dentifier2, and this will create both cryptograms asking to the smartcard. Those cryptograms are unpredictable because there is a function `f` which is mangling data and they are proof of confirmation of an operation : transaction or login with the bank.

Two responses have been shown bellow to observe the behavior of both e.dentifiers2 (Old and new version). Let us look at the new e.dentifier2 how presses 'C' itself if it detects that the cardholder have not pressed 'OK' yet (Steps 7-8 at Figure 4.2).

USBcommand	Description
Out: 01 03 06 00 00 00 00 00	GENERATE CRYPTOGRAM
USBresponse	**Description**
	RESPONSE1 with old e.dentifier2 PAYLOAD: ??x8bytes
In: 01 03 **03** XX 00 00 00 00	Attack **WORKS**.The cryptogram will be created
In: 00 06 XX XX XX XX XX XX	cryptogram
....	
In: 00 06 XX XX XX XX XX XX	cryptogram
In: 00 **02** XX XX XX XX XX XX	End of cryptogram: Maestro,VISA **03** MasterCard
	RESPONSE2 with new e.dentifier2 PAYLOAD: 2x8bytes
In: 01 03 **08** 01 00 00 00 00	Attack **FAILS**.The cryptogram won't be created
In: 00 **01** **25** **01** 00 00 00 00	

Table 5.13: Possible traces with GENERATE_AC.
USB traces when the attack [1] works with several EMV Bank cards with old e.dentifier2. Different USB responses have been shown when the attack works, with our emulation of EMV-card ABN-AMRO the attack failed due to some functions that were not implemented on the Javacard.

USBcommand	Description
Out: 01 03 06 00 00 00 00 00	GENERATE CRYPTOGRAM
In: 01 03 08 **03** 00 00 00 00	RESPONSE1: attack fails
In: 00 **03 81 6D** 00 00 00 00	Fake-card ABN
In: 01 03 03 0E 00 00 00 00	RESPONSE2: attack works VISA
	PAYLOAD: 3x8 bytes
In: 00 06 **80 00 0A 02 EC 2B**	Spanish VISA electron (Bankia)
In: 00 06 **08 06 01 0A 03 A4**	
In: 00 02 **A0 00 01 0A 03 A4**	End
In: 00 06 **80 00 0B 89 A5 CB**	Spanish VISA electron (Bankia)
In: 00 06 **6B 06 01 0A 03 A4**	
In: 00 02 **A0 00 01 0A 03 A4**	End
In: 01 03 03 0F 00 00 00 00	RESPONSE3: attack works MasterCard
	PAYLOAD: 3x8 bytes
In: 00 06 **80 00 28 12 EA B7**	Spanish MasterCard (NovaGalicia)
In: 00 06 **FE 01 01 03 A4 20**	
In: 00 03 **06 00 00 03 A4 20**	End with 03
In: 01 03 03 1A 00 00 00 00	RESPONSE4: attack works Maestro
	PAYLOAD: 5x8bytes
In: 00 06 **04 80 01 21 68 35**	Maestro (Rabobank)
In: 00 06 **B5 04 1C 10 A5 00**	
In: 00 06 **03 04 00 00 00 00**	
In: 00 06 **00 00 00 00 00 00**	
In: 00 02 **00 FF 00 00 00 00**	End
In: 00 06 **04 80 01 22 90 8A**	RESPONSE5: attack works Maestro
In: 00 06 **A1 97 1C 10 A5 00**	
In: 00 06 **03 04 00 00 00 00**	
In: 00 06 **00 00 00 00 00 00**	
In: 00 02 **00 FF 00 00 00 00**	End

5.3 Javascript and plugin

A costumer of Internet Banking is able to interact with USB-connected e.dentifier2 through of a web browser and a driver installed previously in his computer. Such driver will be responsible to have a conversation between computer and Banking website using this plugin. This driver will be able to sent USB commands to e.dentifier2 to interact with the smartcard. Such smartcard will converse with e.dentifier2 using APDU commands.

When a costumer visits the Banking website of ABN-AMRO using his web browser, then a HTML embedded object called PLUGIN_BECON will take care of the communication through JavaScript code asking for the device e.dentifier2 every time that costumer performs any operation. This JavaScript program is called BECON.js and many functions were discussed in Arjan Blom's thesis [2]. Some functions were not analyzed in Arjan Blom's thesis [2] due to the fact that they are not used, but however they exist in the JavaScript code. A little background of these functions follows and moreover we can observe a possible output afterwards of JavaScript's function calls.

- **function BECON_CheckConnection()**; Checks if e.dentifier2 and smartcard are ready to start.Moreover it checks if the driver is installed and its version and version of plugin.

  For instance, below we show a successful response (Status=0) when e.dentifier2 and smartcard are ready to create a communication: Let us observe device, driver and plugin versions in the response as well. We have watched both versions of e.dentifier2, the old one with `Device_Version=01.02` and the new one with this version: `01.05`.

  ```
  Status=0
  StatusDescription=Success
  Device_Version=01.05
  Driver_Version=01.24
  BECON_Version=01.04
  ```

- **function BECON_CheckCard()**; Returns bankaccount and card number of the smartcard if device and card are inserted.

For instance, below we show a successful response(Status=0) when e.dentifier2 "ChecksCard".Let us have a look at CardId_PAN where e.dentifier2 returns Bank account number of a JavaCard programmed with this number.

```
Status=0
StatusDescription=Success
CardId_PAN=67030500055981007F
CardId_PSN=01
```

- **function BECON_GetResponse(p_sSignData)**; Returns a signed cryptogram to e.dentifier2 as response. Further details at Arjan Blom's thesis [2].

  In this example, it was inserted `'00000000'` as `p_sSignData` and we obtained the following cryptogram.

```
Status=0
StatusDescription=Success
Response=0480010604C6D7D61C10A50003040000000000000000000000FF0105
```

- **function BECON_DisplayResult(p_iMessageId, p_sLastLogon)**; It will display last logon on e.dentifier2's display. Not relevant for this thesis.
- **function BECON_Cancel()**; Cancels whatever current transaction when caldholder presses cancel button.
- **function BECON_GetObject()**; Returns the object's reference.
- **function BECON_GetLanguage()**; Returns two digits with an identifier `"nl"` for Netherlands or `"en"` for English language).
- **function BECON_GetMode1Response(p_sChallenge, p_iCurrency, p_sAmount)**; Will ask the smartcard to generate a cryptogram using a challenge-response with user's interaction. This function could be used for signing transactions. Further information later.
- **function BECON_GetMode2Response()**; Will ask the smartcard to generate a cryptogram using a challenge-response but without input parameters,it means

no user's interaction. This function could be used for login operations. It will use a challenge-response between smartcard-e.dentifier2 and it will be displayed thanks to the plugin and JavaScript.

Due to secrecy of the protocols, anything is hidden and there is no public documentation. This concept of "Security by obscurity" is very common in Internet Banking and surely quite probably what is the main error mostly of times. The following JavaScript functions, which it will be studied along this section, they are of unknown use. In Arjan Blom's thesis [2], these functions were not investigated. They seem to be unused. Although we have not explicitly seen to the driver calls those functions, we have found them out in detail how they are working.

The following JavaScript functions [6] to analyze:

- function **BECON_GetMode1Response(p_sChallenge, p_iCurrency, p_sAmount)**;
- function **BECON_GetMode2Response()**;

These were tools we used to achieve hidden information:

- *Own HTML webpage* + JavaScript BECON.js in order to launch functions to e.dentifier2.
- *RebelSim* device in order to eavesdrop APDU command-response.
- *USBTrace* in order to intercept USB-commands.

First at all, it was found out how long parameters were to avoid input errors with the plugin. Through of unix command *'strings'* against the driver we could obtain information about the right length. This was important in order to interact with the plugin to recover right data.

Secondly, our own webpage was created in order to call every function of BECON.js and get information about how those functions work.Using the RebelSim device in order to achieve a Man-in-the-middle between smartcard and reader and then were able to discover raw APDUs associated to those functions. Finally RebelSim, USBTrace and our own webpage, in this order exactly,were executed to sniff as much traffic as possible.

Table 5.14: USB commands and responses for Getmode1-Getmode2Response
In the following table we can have a big picture of main USB commands and responses in the e.dentifier2. Both functions share USB responses when the cardholder accepts/denies transactions or do an authentication with bank for login operations.

Below, we can observe USB traffic that we will use later.

USBcommand	Description
Out: 01 03 07 17 00 00 00 00	GetMode1Response + 32 bytes of payload
Out: 01 03 07 01 00 00 00 00	Getmode2Response
USBresponse	**Description**
In: 01 03 07 00 00 00 00 00	User presses Cancel
In: 01 03 03 04 00 00 00 00	User presses OK
In: 01 03 08 01 00 00 00 00	Communication error
In: 00 04 $R_0R_1R_2R_3R_4R_5R_6R_7$ 00 00	Response=$R_0R_1R_2R_3R_4R_5R_6R_7$

5.3.1 GetMode1Response

In this section, we have reverse engineered the JavaScript function *GetMode1Response*. This function is used in USB-unconnected mode, also called as '`5.Securecode`', but it has also been developed for Internet Banking. Nowadays this function seems unused in USB-connected mode. This function is used to authorize payments using challenge-response authentication protocol.

This function has a header such as:

`BECON_GetMode1Response(p_sChallenge, p_iCurrency, p_sAmount)`

To understand better this function, it has been marked with numbers in the diagram to follow with next steps explained below:

1. The web browser sends to cardholder's PC a challenge, currency and amount following this order in this header: `BECON_GetMode1Response(p_sChallenge, p_iCurrency, p_sAmount)`. Web browser will check out input parameters before sending to PC. Some values have to be between those lengths:

    - Amount: 12 digits and it must be among [0000.000.000,00 - 0999.999.999,99]

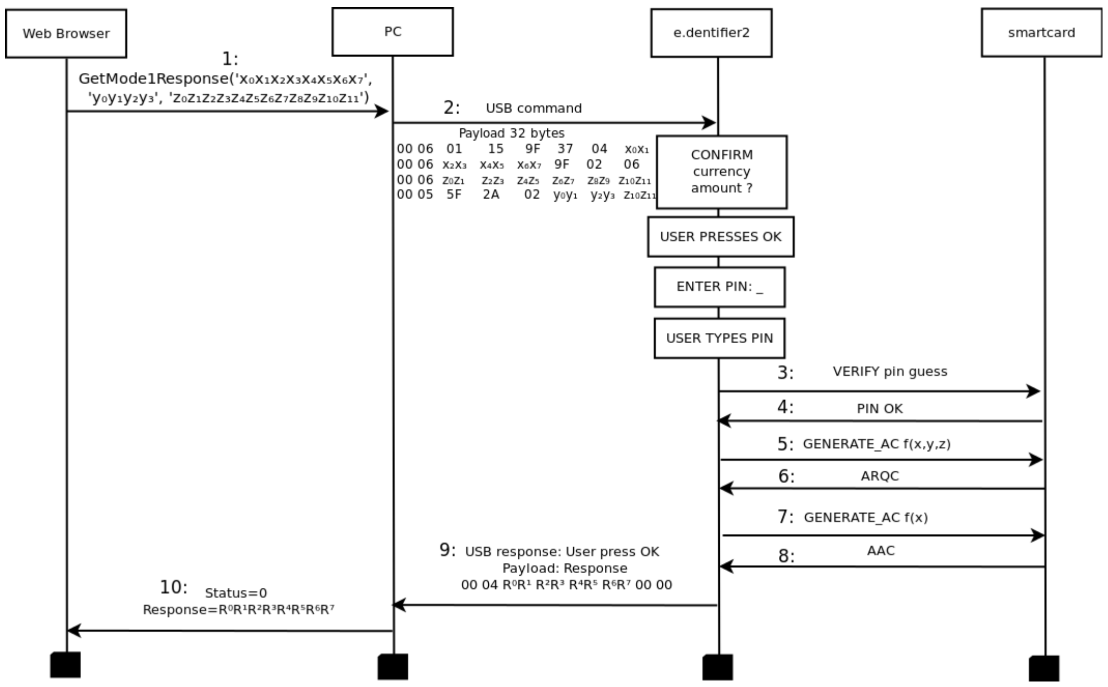
Web Browser
PC
e.dentifier2
smartcard
1: GetMode1Response('x0x1x2x3x4x5x6x7', 'y0y1y2y3', 'z0z1z2z3z4z5z6z7z8z9z10z11')
2: USB command
Payload 32 bytes
00 06 01 15 9F 37 04 x0x1
00 06 x2x3 x4x5 x6x7 9F 02 06
00 06 z0z1 z2z3 z4z5 z6z7 z8z9 z10z11
00 05 5F 2A 02 y0y1 y2y3 z10z11
CONFIRM currency amount ?
USER PRESSES OK
ENTER PIN: _
USER TYPES PIN
3: VERIFY pin guess
4: PIN OK
5: GENERATE_AC f(x,y,z)
6: ARQC
7: GENERATE_AC f(x)
8: AAC
9: USB response: User press OK
Payload: Response
00 04 R0R1 R2R3 R4R5 R6R7 00 00
10: Status=0
Response=R0R1R2R3R4R5R6R7

Figure 5.1: GetMode1Response diagram. - It could be used as signing transaction.

- Currency : 4 digits. It was observed that the currency icon was displayed on the screen with different values depending on the currency code according to the EMV specifications:

  - EMV code: `0978` to use € Euros.
  - EMV code: `0826` to use £ Pounds sterling.
  - EMV code: `0840` to use $ Dollars.

- Challenge: 8 numeric digits.

2. The PC sends a USB command with 32 bytes of payload. Some curious findings were the following:

   - If the currency was not matched with EMV specifications, a value of `0999` [1] was taken by default.
   - The last byte of payload is the same that is in the amount : $z_{10}z_{11}$ (Two last bytes in the step 2 Fig.5.1).

[1] The codes assigned for transactions where no currency is involved

A message will be displayed in the screen with details of the transaction and cardholder will be able to decide if he wants to accept or deny the payment. In this example, we assume that user presses ’OK’.

3. The cardholder enters his PIN code and presses ’OK’. The e.dentifier2 tries to verify if the PIN code is right.
4. If the PIN code matches, smartcard returns ’OK’.
5. The e.dentifier2 starts to generate a cryptogram with input parameters of USB command payload such as: `GENERATE_AC f(x,y,z)`.
6. The smartcard returns a `ARQC` *(Authorization ReQuest Cryptogram).*
7. The e.dentifier2 sends `GENERATE_AC f(x)` in order to achieve a confirmation in the transaction.
8. The smartcard returns an `AAC` *(Application Authentication Cryptogram)* as confirmation towards e.dentifier2.
9. The e.dentifier2 calculates a response with cryptograms appling a bitfilter [3,4] and it returns a USB response that takes a confirmation (User pressed ’OK’) and 4 bytes with 8 numeric digits as response to challenge.
10. PC received this response over USB response, and now PC sends through of driver a response to Web browser. And two parameters explaining if operation was right or something was wrong. A status ’0’ means successful.

Note that in Figure 5.1, 5.3 there is no communication back to PC saying that the user pressed OK. Let us remember in Chapter ”The new e.dentifier2”, that in the attack [1] the e.dentifier2 notified a message back to PC.

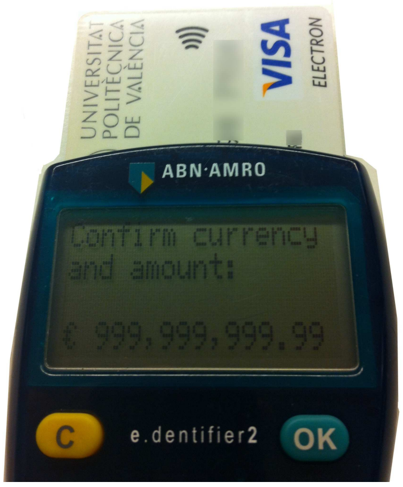
UNIVERSITAT POLITÈCNICA DE VALÈNCIA
VISA ELECTRON
ABN·AMRO
Confirm currency and amount:
€ 999,999,999.99
C
e.dentifier2
OK

Figure 5.2: BECON_GetMode1Response - Challenge with maximal allowed amount sent to e.dentifier2 (Step 2 in the diagram of GetMode1Response, this picture was exactly taken before user pressed OK).

5.3.2 GetMode2Response

In this section we are going to reverse engineer another unused function in the JavaScript code called *BECON.js*. The difference with *GetMode1Response* is that this function has no input parameters, so this function could be used to login with ABN-AMRO's website generating a signed response. Moreover, we can note a difference in USB commands because are always fixed. Therefore its APDU payload is empty (The digits "`8-34-10002` are always fixed in both functions).

However, now the cardholder has not to sign any transaction, he just needs to enter his PIN code and achieve a response in order to get authentication with the bank. Once PIN code is verified, two cryptograms will be generated for the smartcard. Then e.dentifier2 will calculate a response with those cryptograms.

1. The web browser uses JavaScript code to call *GetMode2Response*. This sends it to PC.

2. The PC sends a USB command using e.dentifie2's driver. The e.dentifier2, when receives this USB command, asks for PIN code in the display.

Table 5.15: BECON_GetMode1Response ('99999999', '5555','012345678901')
In the following table we have sent a wrong currency, directly the code `0999` is assigned when currency is not involved. As challenge, we sent 4 bytes with a value of '`99999999` and an amount of 6 bytes '`0123456789`<u>`01`</u>'. We already explained as last byte of amount was repeated at the final of payload (It is undercored in the table bellow as well). We observe in APDU commands, `GENERATE_AC`, as challenge-amount-currency are sent to calculate ARQC in the smartcard. Also we have seen in order to calculate AAC, it is only sent the challenge. We do not know how response is generated, it must be researched with different bit filters. Some bit filters [3,4] have been checked without success. To apply these filters, we have used `ATC` (Application Transaction counter, which has not been shown in this table) and both cryptograms: `ARQC` and `AAC`.

Javascript Input	USB command	APDU command: Generate AC

Challenge=99999999
Currency=5555
Amount=012345678901
	01 03 07 17 00 00 00 00
	00 06 01 15 9F 37 04 **99**
	00 06 **99 99 99** 9F 02 06
	00 06 **01 23 45 67 89 01**
	00 05 5F 2A 02 **55 55** <u>01</u>
		80AE80002B AE (generate ARQC)
		01234567890100000000000000008000
		000000**0999**000000000**99999999**34
		00000000000000000000010002
		80AE00001D AE (generate AAC)
		0000000000000000000005A3380000000
		0**99999999**0000000000000000

Javascript Output	USB response	APDU response: ARQC and AAC

Status=0
StatusDescription=Success
Response=39046235
	User presses OK
	01 03 03 04 00 00 00 00
	Response
	00 04 **39 04 62 35** 00 00
		00C000002B (response)
		9F2608 5BAA4E31F8F56A (ARQC)
		9F1012 **1C10A50003040...00FF**
		00C000002B (response)
		9F2608 4EFD77D8F7438B1 (AAC)
		9F1012 **1C10250003440...00FF**

Table 5.16: BECON_GetMode1Response('66666666', '0978', '033333333333')
In the following table we have sent €uros as Currency using its EMV code (0978). As challenge, we sent 4 bytes with a value of '66666666 and an amount of 6 bytes '0333333333<u>33</u>'. In this example, we can also have a look to last byte of the amount because is repeated at the end of the payload (underscored).

This example is very similar to table 5.15. Read more specifications there.

Javascript Input	USB command	APDU command: Generate AC

Challenge=66666666
Currency=0978
Amount=033333333333
	01 03 07 17 00 00 00 00
	00 06 01 15 9F 37 04 **66**
	00 06 **66 66 66** 9F 02 06
	00 06 **03 33 33 33 33 33**
	00 05 5F 2A 02 **09 78** <u>33</u>
		80AE80002B AE (generate ARQC)
		03333333333300000000000000008000
		000000**0978**00000000**66666666**34
		00000000000000000000010002
		80AE00001D AE (generate AAC)
		000000000000000000005A33800000000
		0**66666666**0000000000000000

Javascript Output	**USB response**	**APDU response: ARQC and AAC**

Status=0
StatusDescription=Success
Response=37511669
	User presses OK
	01 03 03 04 00 00 00 00
	Response
	00 04 **37 51 16 69** 00 00
		00C000002B (response)
		9F2608 **61F59498685E2762** (ARQC)
		9F1012 **1C10A50003040...00FF**
		00C000002B (response)
		9F2608 **8B4BD52036AECD3F** (AAC)
		9F1012 **1C10250003440...00FF**

Table 5.17: Log of USB-communication during GetMode1Response method
Executing **BECON_GetMode1Response**('66666666', '0978', '033333333333')

During this trace the language was not set and it was possible to observe as 2 language's bytes remained empty. We observed that language by default is Dutch in this case. We sent € as currency (0978) and different values to easily detect challenge's and amount's bytes. Moreover you can note the two USB commands previous to GetMode1Response as usual.

USBcommand	Description
Out: 01 03 01 02 00 00 00 00	SHOW PICTOGRAM
Out: 00 02 00 00 00 00 00 00	SET LANGUAGE
Out: 00 02 00 00 00 00 00 00	SET LANGUAGE
USBresponse	Description
In: 01 03 01 01 00 00 00 00	Card's inserted
In: 00 01 01 01 00 00 00 00	Card's inserted
USBcommand	Description
Out: 01 03 07 17 00 00 00 00	GetMode1Response + 32 byte Payload
Out: 00 06 01 15 9F 37 04 **66**	Challenge:1 byte= 66
Out: 00 06 **66 66 66** 9F 02 06	Challenge:3 bytes = 66 66 66
Out: 00 06 **03 33 33 33 33 33**	Amount:6 bytes 03 33 33 33 33 33
Out: 00 05 5F 2A 02 **09 78** 33	Currency €:2 bytes 09 78
USBresponse1 OK	Description
In: 01 03 03 04 00 00 00 00	User presses OK
In: 00 04 **38 73 38 76** 00 00	Response 38 73 38 76
Out: 00 02 00 00 00 00 00 00	GET ATR
In: 00 06 3B 67 00 00 29 20	APDU 3B6700002920-006F78-9000
In: 00 05 00 6F 78 90 00 20	
Out: 00 02 00 00 00 00 00 00	
USBresponse2 CANCEL	Description
In: 01 03 07 00 00 00 00 00	User presses Cancel
In: 02 81 00 00 00 00 00 00	Sometimes is not in trace
In: 02 81 01 00 00 00 00 00	Sometimes is not in trace
Out: 02 01 00 00 00 00 00 00	GET ATR
In: 00 06 3B 67 00 00 29 20	APDU 3B6700002920-006F78-9000
In: 00 05 00 6F 78 90 00 20	
Out: 00 02 00 00 00 00 00 00	

3. The e.dentifier2 lets smartcard verify PIN code.
4. The smartcard returns 'OK' is the PIN code is correct.
5. The e.dentifier2 asks for `ARQC` sending `GENERATE AC` APDU command.
6. The smartcard generates and returns `ARQC`.
7. The e.dentifier2 asks for `AAC` sending `GENERATE AC` APDU command.
8. The smartcard generates and returns `AAC`.
9. The e.dentifier2 replies to PC sending two USB responses, first with 'OK' and then with the response. In this point the e.dentifier2 calculates the response from cryptograms.
10. PC sends the response to web browser using the plugin and driver. And two parameters explaining if operation was right or something was wrong. A status '0' means successful.

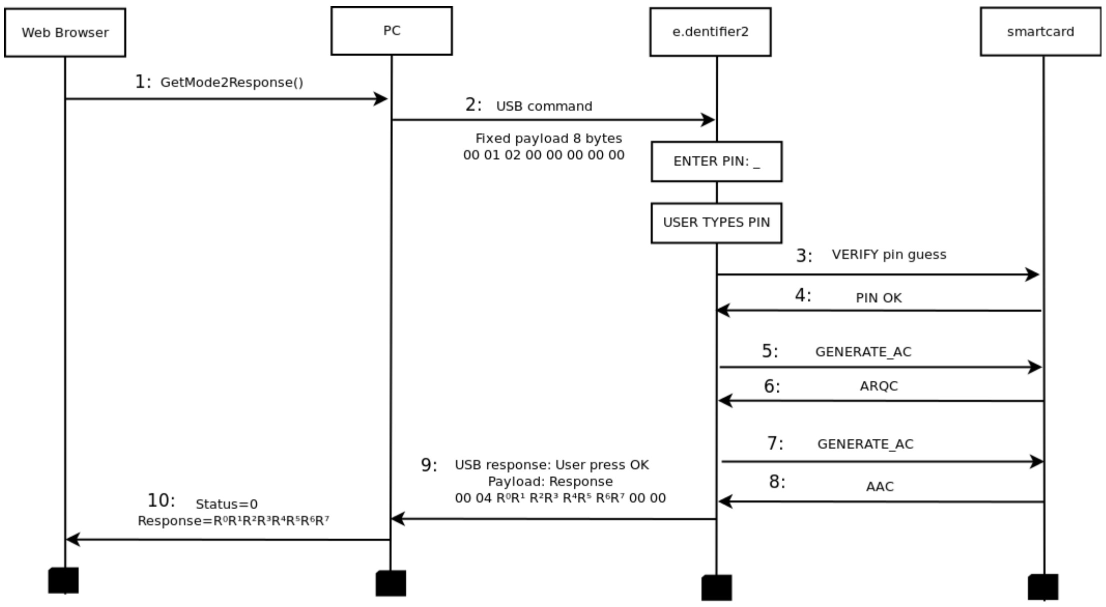

Figure 5.3: GetMode2Response diagram. - It could be used as login.

Table 5.18: BECON_GetMode2Response()
The following table is showing how *GetMode2Response* is working. This function could be used as login in order to generate a signed response.

Javascript Input	USB command	APDU command: Generate AC
		80AE80002B AE (generate ARQC)
		00000000000000000000000000008000
		0000000000000000000000000034
	01 03 07 01 00 00 00 00	00000000000000000000010002
	00 01 02 00 00 00 00 00	80AE00001D AE (generate AAC)
		000000000000000000005A33800000000
		0000000000000000000000000
Javascript Output	**USB response**	**APDU response: ARQC and AAC**
		00C000002B (response)
Status=0	User presses OK	9F2608 **3817E8EC705522AE** (ARQC)
StatusDescription=Success	01 03 03 04 00 00 00 00	9F1012 **1C10A50003040...00FF**
Response=37435415	Response	00C000002B (response)
	00 04 **37 43 54 15** 00 00	9F2608 **1D2409403CE420A5** (AAC)
		9F1012 **1C10250003440...00FF**

Chapter 6

Future work

In this chapter, we will discuss some points which could be not researched given the time. They could be useful for future students who want to research the e.dentifier2 or whatever EMV-CAP smartcard reader. Actually, we encourage to start with *DigiPass 850*, an EMV-CAP smartcard reader of ING Direct. Also it would be interesting to find out if more versions of e.dentifier2 are being used nowadays. A list of possible suggestions :

For the EMV-CAP smartcard reader of ING:

- Detect baud rate for *DigiPass 850*.
- Find out USB traffic. We attached a web link for login. Unfortunately, I did not have an ING account to try get information about it. We observed 6 bytes USB commands in attempts with this weblink and using *USBTrace*. It was attempted to get login in this weblink, but we did not get a special CD with additional information to achieve our web browser interacts with *DigiPass 850*.
- Investigate more versions of VASCO vendor. They could use similar systems and protocols.

For the e.dentifier2:

- Some USB commands are still unknown. They are highlighted with ?? in this thesis.

- Try to figure out how the function `f(x,y,z)` in *GetMode1Response* is returning the response from cryptograms. This can be done tweaking more values as Currency,Amount and Challenge and trying out more bit filters [3,4]. The same for *GetMode2Response* in the response. Some bit filters were attempted but they went wrong to calculating the response.

Chapter 7

Conclusion

In this chapter we will discuss questions so far unanswered in this document. A big question is if ABN-AMRO was able to properly implement SWYS in the new e.dentifier2. This question includes more subquestions than we can talk about in this chapter.

The main goal of the e.dentifier2 was and is: manage a handy and secure way to authorize transactions for customers. This way, also known as SWYS by ABN-AMRO, seems to be safer than the old e.dentifier2. Nevertheless, this does not mean that SWYS is completely reliable. In this document, we have compared both versions of e.dentifier2. The new version is not vulnerable to last attack [1]. New vectors of malware could appear in the new version in the future. In this thesis, we also looked at unused functionalities in the USB-connected mode. The JavaScript function `GetMode1Response(Challenge, Amount,Currency)` does not have the problem of sending a message to the PC when the user has pressed OK. In the function to authorize transactions, `GetMode1Response`, the SWYS protocol appears to have been correctly implemented. This means that the user is asked to authorize a transaction that can be understandable for him. This transaction is shown in the screen of the e.dentifier2 and the user can press OK, however now this message is not "leaked" towards the PC as happened in the `SIGN-DATA-TEXT` (Only old e.dentifier2). The e.dentifier2 waits for the user response in the smartcard reader, after asking for a PIN and immediately after starts to generate a couple of cryptograms with `GENERATE AC` in order to get a response.

Some questions are still without an answer:
Why does this function not have this flaw while the other does? If they are doing more or less same operations.
Why was not it used in USB-connected mode if it was much more secure?
Possibly ABN-AMRO did not pick `GetMode1Response` due to the website being able to give more information on the screen of the smartcard readers using `(SIGN-DATA-TEXT)`.
Why is this function only available in USB-unconnected mode smartcard readers?

These questions are still a mystery and they are out side of our scope. Nevertheless, they are still open further discussion.

References

[1] Arjan Blom, Gerhard de Koning Gans, Erik Poll, Joeri de Ruiter, and Roel Verdult. **Designed to Fail: A USB-Connected Reader for Online Banking**. *17th Nordic Conference in Secure IT. Karlskrona, Sweden.*Octobre 31, 2012.

[2] Arjan Blom. Supervisor: Erik Poll .Master thesis. **ABN-AMRO e-dentifier2 reverse engineering**. *Digital Security Group. Institute for Computing and Information Science. Radboud University Nijmegen*, December 19, 2011.

[3] Michael Schouwenaar. **Dutch EMV-cards and Internet Banking**. *Digital Security Group. Institute for Computing and Information Science Radboud University Nijmegen.*

[4] Saar Drimer, Steven J. Murdoch, and Ross Anderson. **Optimised to Fail: Card Readers for Online Banking**. *Computer Laboratory, University of Cambridge, UK* .

[5] Daniele Bianco, Adam Laurie, Andrea Barisani, Zac Franklen. **Chip and PIN is definitely broken. Credit Card skimming and PIN harvesting in an EMV**. *InversePath and ApertureLabs* 2011.

[6] Wikipedia. **Chip Authentication Program** . *http://en.wikipedia.org/wiki/Chip_Authentication_Program#Operating_principle* .

[7] Erik Poll, Joeri de Ruiter and Lejla Batina. **Website of Hardware Security subject**. *Digital Security Group. Institute for Computing and Information Science. Radboud University Nijmegen. http://www.cs.ru.nl/~erikpoll/hw/index.html* .

[8] Joeri de Ruiter. **Smart cards en EMV**. *Guest lecture for the Bachelor course Security at Utrecht University.* May 11, 2012.