FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Java Card Second-factor Authentication Plugin for KeePass
Student:	Erich Winkler
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

The goal of this thesis is to develop a plugin for the KeePass password manager that allows unlocking a password database using the Java Card technology.

1. Get familiar with the environment for writing Java applets and figure out how to properly upload them to a smart card.
2. Write an applet for KeePass that provides the second factor of authentication.
3. Write a program that provides communication with the smart card.
4. Modify the plugin to communicate with the smart card.
5. Create a Java authentication applet.
6. Make the whole test environment work - communication between KeePass plugin and the smart card (applet), unlocking the database with the card.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 11 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Java Card Second-factor Authentication Plugin for KeePass

Erich Winkler

Department of Computer Systems
Supervisor: Ing. Jiří Dostál, Ph.D.

December 11, 2021

Acknowledgements

Writing this thesis was extremely challenging for me, and I had to develop numerous new skills to fulfill all initial goals successfully. I would like to thank my family for all the support and understanding, and my supervisor Ing. Jiří Dostál, Ph.D., for his time, interest, and patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on December 11, 2021 ....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Winkler, Erich. *Java Card Second-factor Authentication Plugin for KeePass*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`http://site.example/thesis`⟩.

Abstrakt

Tato práce využívá Java Card technologii jako druhý stupeň autorizace pro KeePass. Tento faktor je implementován za použití systému zásuvných modulů, který je součástí KeePassu. Pro zajištění bezpečnosti celého procesu, práce analyzuje možná bezpečnostní rizika a implementuje bezpečnostní opatření, která tyto rizika snižuje. Největšími riziky jsou emulace Java karty, odposlouchávání komunikace a slabý databázový klíč. Jako první bezpečnostní opatření je použit PIN kódu, který ověřuje uživatele před použitím Java karty. Dalším, neméně důležitým opatřením je šifrování komunikace za pomoci RSA šifry. To je úzce spojeno s použitím Secure Channel protokolu, který zajišťuje, že Java karta neposkytne databázový klíč žádné jiné entitě než danému Pluginu. Posledním důležitým krokem bylo použití bezpečného algoritmu, podporovaného Java Card technologii, ke generování databázového klíče. Tato práce je užitečná reference pro kohokoliv, kdo chce zkoumat možnosti vývoje Java Appletů včetně podporovaných bezpečnostních prvků a systému zásuvných modulů pro KeePass.

Klíčová slova KeePass password manager, Smart Cards, Java Card, Java Applet, 2FA, KeePass zásuvný modul, RSA, 3DES, Secure Channel Protokol

Abstract

This thesis deals with Java Card Technology as a second-factor of authentication for KeePass Password Manager. The 2FA is implemented by using the KeePass Plugin System along with the Java Card applet. To ensure maximum security, this work analyses possible security threats and implements security measures that prevent them. The most significant threats are the danger of emulating the Java Card, communication sniffing, and weak database key. All of them are prevented by a combination of the following factors. First of all, protecting the card by PIN code. Second, encrypt the communication by using the RSA cipher. Third, implementing the Secure Channel Protocol that ensures the Java Card does not provide the database key to any other entity, but the KeePass plugin. Last, proper generation of the database key with a secure random algorithm that is supported by the Java Card. This thesis is a good reference for anyone who wants to explore the possibilities of Java Applet development including supported security functions, and the KeePass Plugin System.

Keywords KeePass password manager, Smart Cards, Java Card, Java Applet development, 2FA, KeePass Plugin Development, RSA, 3DES, Secure Channel Protocol

Contents

List of Figures

List of Tables

Introduction

People have been trying to discover the perfect way to authenticate users for decades. However, the world of technology develops extremely quickly and what used to be safe in the past is not necessarily safe anymore. Therefore, the authentication methods need to be modified or combined to reach the required level of security.

There is a perfect example from everyday life. Passwords have been a primary method of authentication in most computer systems for decades. Back in the day, even a relatively simple, easy-to-remember password could be used to provide a satisfying level of protection to your account. The years passed, and now the password needs to be complex, therefore very hard to remember, and with all that exposure to the online world and billions of stolen credentials every year, all your passwords need to be changed frequently.

To keep this authentication method current, password managers were developed. They allow the user to keep their password strong and secure, therefore increasing the level of protection this authentication factor provides. However, studies indicate that the best way to protect your accounts and data is to add a second level of authentication.

KeePass Password manager allows you to create your own authentication plugins. That means it is relatively easy to add another security layer of your choice to protect your password database efficiently.

Naturally, the best protection is provided by methods that authenticate you by your unique physical characteristics, but these methods are usually costly as they require special hardware and technology. It is better to use something many people already have, or it is relatively cheap to buy.

Java Cards are commonly used these days for many purposes. They are relatively cheap to buy, do not require any expensive additional hardware, and support various cryptographic methods. Besides, it allows you to develop your own applet.

With all stated above, KeePass Password Manager with Java Card seems like a perfect fit. Together it allows you to have your password database safely stored in your computer protected by two authentication factors. In summary, the product of this thesis provides a relatively cheap and efficient way of protecting your password to anyone.

Chapter 1

Goal of the Thesis

This thesis aims to develop a plugin for the KeePass password manager that allows unlocking a password database using the Java Card technology. In other words, add the possession factor into the authentication process. In order to make it happen, the following steps need to be followed.

Firstly, get familiar with the environment for writing Java applets and figure out how to properly upload them to a smart card. Create a Java authentication applet for a smart card, which communicates with a plugin.

Secondly, create a plugin that provides a second factor of authentication for KeePass. Modify the plugin to communicate with a smart card.

Lastly, make the whole environment work and secure the communication between the Smart card and the KeePass plugin.

Chapter 2

State-of-the-Art

This chapter describes factors of authentication, including its advantages and drawbacks. It points out the importance of two-factor authentication and explains the goal of this thesis. Furthermore, it presents popular tools and devices used for authentication, especially password managers and smart cards.

2.1 Authentication Factors

An authentication factor is used to verify the identity and authorization of a user. In the past, unique usernames and self-selected passwords were used as the primary method of authentication. With the increasing computing power of the hardware, it turned out that creating more and more complex passwords is not sufficient. Let's take a look a closer look at all types of authentication factors one by one.

2.1.1 Knowledge Factors

Knowledge-based factors require the user to provide some information in order to get access to a secure system. This information can be anything you know, e.g., PIN or password. Multiple major drawbacks come with this type of authentication.

Firstly, an average user uses multiple services where authentication is needed. Unfortunately, they also tend to use the same password for all their logins. [1] This practice presents a significant security risk in case your password gets compromised. There are now more than 15 billion stolen credentials from 100,000 data breaches available to cybercrime actors. Of this number, some 5 billion are said to be unique, with no repeated credential pairs. The number is increasing, and even with proper habits in the online world, no user can be sure that his credential has been compromised. [2]

Secondly, the exponentially increasing computational power of the hardware makes the passwords vulnerable to brute force attacks. To prevent this type of attack, the passwords need to get longer and more complex. That also makes them harder to remember, which results in using well-known patterns as replacing “o” with “0”. Even according to the National Cyber Security Council: “Imposing additional password complexity requirements on such accounts will increase the burden on these users, but may not provide any more protection.” [3]

Thirdly, even if the credentials are not compromised by the data breach, there are multiple ways how to steal passwords directly from the user.

- Keylogging
    - Secretly recording the keys struck on a keyboard.
- Phishing
    - Fraudulent practice in which an attacker masquerades as a reputable entity, trying to trick the victim to click on malicious links or attachments.
- Pharming
    - Producing fake websites and redirecting users to it in order to steal confidential information. [4]

Despite its drawbacks, knowledge factors are a primary method of authentication in the majority of systems. It seems the future is not in replacing this factor. Bill Gates predicted that the password will be replaced by an alternative method 15 years ago. It clearly did not happen. In fact, the surge of online services has risen significantly. Mainly because it is a well-understood concept by the general public. Also, it is often seen as an easily-implemented, low-cost security measure that does not require special hardware. [5] That is why instead of replacing them, solutions to their disadvantages have been found in the form of MFA and password managers.

2.1.2 Possession Factors

Possession factors are commonly described as “something the user has”. They require the user to possess a specific device or specific piece of information to get access to a secure system. It is essential to distinguish between connected and disconnected tokens.

Connected security tokens are physical items that require a physical connection to generate automated authentication data transfer. The most common examples are Smart Cards, USB tokens, or RSA SecureID tokens. Disconnected tokens don’t need or require to physically connect the item to the device. In this case, a one-time code generated by the token is required to get access to the system. It can be a text message with a code or app on your phone. [6]

Regardless of the type of token, the advantages and disadvantages mostly stay the same. Naturally, one of the most significant advantages is that it cannot be easily compromised without physical access to an item, unlike the knowledge-based factors where the password can be compromised remotely or stolen from a weakly secured database. On the other hand, if the token is lost or stolen, it cannot be used to access the system. In a worse case, an attacker gains unauthorized access. That is why the token should be protected by another layer of security, e.g., PIN code.

It is not recommended to use possession factors for SFA. The risk of losing the item is significant. Although, they fit perfectly to a multi-factor authentication concept. It provides a relatively cheap and very efficient way to improve the security of a system.

2.1.3 Inherence Factors

Inherence factors commonly described as “Something you are” have been increasing their popularity significantly in the past few years, especially for everyday use as unlocking a user’s phone using a fingerprint. These factors generally take the form of biometrics, e.g., face recognition, fingerprints, or iris imprint.

For obvious reasons, they are the most unique and reliable authentication factors possible. Unfortunately, also the most difficult to manage. They require special hardware as a fingerprint scanner. That also means that the system or account can be accessed only on devices with hardware that supports that specific authentication factor. This restriction is undoubtedly useful for security but could be inconvenient for many use-cases. [7] On the other hand, the user does not need to remember or carry anything in order to access the system. [8]

2.1.4 2FA Authentication

Two-factor authentication is a security process to better protect users and resources by asking users to verify identity in two ways. It always uses two of the above-described factors. By adding an additional security layer to the authentication process, it provides a higher level of security than SFA.

The idea behind 2FA is simple. It is significantly harder for the attacker to get everything needed to access the account or service. A combination of factors compensates for the majority of each other’s drawbacks. In the majority of implementation, knowledge factors are the first factor of authentication, and either possession or inference factors are the second. In this case, even if the password is compromised, it is still not enough to login into the account. Therefore, it protects against identity theft via stolen passwords.

In addition, 2FA provides reliable protection against various attacks and frauds as keylogging, phishing, or pharming. Even if the user gets tricked by the phishing email or click on a malicious link and give the credentials to the attacker, the second factor is still there to prevent a successful takeover of the account. [9]

To conclude, according to a 2019 security report from Microsoft, there are 300 million fraudulent sign-in attempts to their cloud services every single day. The same report says that providing an extra layer of security blocks over 99.9 percent of account compromise attacks. That is why it is the most efficient method, which has already become very popular and commonly used. [10]

2.2 Password Managers

Password managers are computer programs that allow users to store, generate and manage their passwords. This section will take a look at their functionality and essential features and briefly describe their advantages and disadvantages.

In section 2.1.1 multiple drawbacks of the password method of authentication have been mentioned. Password managers provide a solution to most of them. All user's passwords are securely stored in an encrypted database. That prevents users from storing passwords in plain text or excel tables. The database is usually encrypted using AES-256, which is generally considered secure and impenetrable using brute-force methods. For unlocking the database, you need a master key. That could be, e.g., password or key file.

Furthermore, if the password manager provides an option to store the password on their cloud, the zero-knowledge architecture is used. That enables a provider to encrypt and store user's data with zero knowledge about the data they are storing. This means, even if your provider becomes a victim of a successful attack, your data are not in danger. The user is the only one who can decrypt them, so even if the attacker steals the database file from the provider, he can't read get the passwords or any other sensitive data out of it.

Another typical issue is using weak passwords or reusing passwords for multiple services. Both of them are solved by generating a strong, unique, and complex password for each service. It is no secret that the weakest part of the password type of authentication is the user. Both of the issues above are caused by users. This is why it is convenient to let the password manager manage their passwords for them because it is much more likely for the majority of users to remember one strong password as a master password than follow all the safety practices to keep their accounts safe.

Despite all the advantages, every tool has its drawbacks. And even for password managers, there are guidelines, which prevent users from making a mistake while using them. The most obvious security threat is a weak master password. Choosing a long passphrase for the master password is crucial to protect your password database from being stolen. The passphrase needs to be sufficiently long to protect against attacks while still allowing memorization.

The threat of a weak master password can be significantly decreased by using 2FA. This way, even if your master password is compromised, your database is still protected by the second security layer. [11]

NIST guidance on password managers:

- "*C*hoose a long passphrase for the master password to the password manager and protect it from being stolen. A passphrase can be made sufficiently long to protect against attacks while still allowing memorization."
- "*C*reate unique passwords for all accounts or use the capability of most program managers to generate random, unique, complex passwords for each account."
- "*A*void password managers that allow recovery of the master password. Any compromise of the master password through account recovery tools can compromise the entire password vault."
- "*U*se multi-factor authentication for program manager applications that allow that capability."
- "*U*se the password generator capability in most password managers to generate complex, random text answers to online "security" questions for those sites still using them." [11]

2.2.1 Comparison of Popular Password Managers

As for every other software, every password manager is better or worse for different use-cases. We will try to compare them by the following criteria: Platform compatibility, security, features, and extensibility.

2.2.1.1 1Password Password Manager

1Password app is available for all common platforms like Windows, Mac, iOS, Android, and Linux. That is indubitably great for users who often change their devices. Interesting is that 1Password has its own stand-alone browser extension for Brave, Chrome, Edge, Firefox, and Safari. So even with browsers, they are trying to be compatible with as many as possible. Furthermore, the plugins now support biometric logins, which, as you know from section 2.1.2 is the most reliable factor of authentication.

This password manager supports 2FA, which is important for multiple reasons described in subsection 2.1.4. The database itself is encrypted using 256-bit AES, which makes it resistant to brute force attacks. Another feature is called "Travel Mode". This mode deletes all sensitive data from devices, which is useful particularly while you are traveling. And what every user should appreciate is Password monitoring. It alerts you if your password is weak, vulnerable, breached, or duplicated.

For the purposes of this thesis, one parameter is particularly important. 1Password does not allow users to create their own plugins. This approach means that you cannot add a new feature. On the other hand, it also means that all the plugins are checked by the creators of 1Password, and the company guarantees their security.

2.2.1.2 LastPass Password Manager

LastPass password is very similar to 1Password. It is available for the same platforms and also offers plugins for multiple browsers like Chrome, Firefox, Safari, Edge, and Opera. Therefore, when it comes to compatibility, this password manager is an excellent choice.

LastPass also provides a password generator, which generates a strong, unique password. Although, you cannot customize the parameters of the password generator. That could be an issue if the generated password does not meet the given requirements. The password manager stores all password databases on the cloud using zero-knowledge architecture. All user's data are locally encrypted before they are sent to the cloud. Hence, it is safe to transfer them over the internet.

The biggest drawback of this password manager is that it does not support 2FA. As you can read in section 2.1.4, 2FA is extremely increasing the security of the login process. Moreover, LastPass does not support creating your own plugins, so you cannot add this feature by yourself.

2.2.1.3 KeePass Password Manager

KeePass is a completely open-source-based password manager. It is available for Windows and for other platforms as Linux or macOS using Mono. This means that compatibility is significantly worse than the password managers above have. Auto-fill is provided by a plugin, which is supported by all commonly used browsers.

The most significant difference between KeePass and most other password managers is storing the data locally on your device instead of on clouds. This solution is excellent for people who do not like the idea of not having complete control over their password database. Although, if you need to access your database from multiple computers, there is an option of uploading your database to services like OneDrive or Google Drive.

KeePass creators are very proud of the security strength of their program. It checks itself with every run and alerts if an algorithm fails the test. Also, it provides complete database encryption, including all notes and details. It supports 256-bit AES and Twofish. Also, KeePass has process memory protection, which prevents using the process of dumping memory to disk and revealing your passwords. Moreover, 2FA is supported. It is possible to combine a master password with other ways of authentication as a key file. The key file can also be carried by a physical piece of hardware as a Smart Card. [12]

Most of the features and extensions are provided by plugins. Most of the plugins are available on KeePass official website. If you do not find a suitable plugin for your issue, you can either create your own plugin or modify any of the ones available. That gives you a lot of options on how to improve your password manager but also could be a security threat. However, there are instructions and templates for plugins on the official KeePass website, which makes it easier to implement the new plugin correctly.

2.3 Smart Cards

There is much debate over what is the exact definition of a smart card. But for this thesis, a smart card will be defined as a physical electronic authorization device used to control access to a resource. It is typically a type of chip card that contains an embedded computer chip. The chip is either a memory or microprocessor type that stores and transacts data. There are multiple characteristics that we use to categorize smart cards.

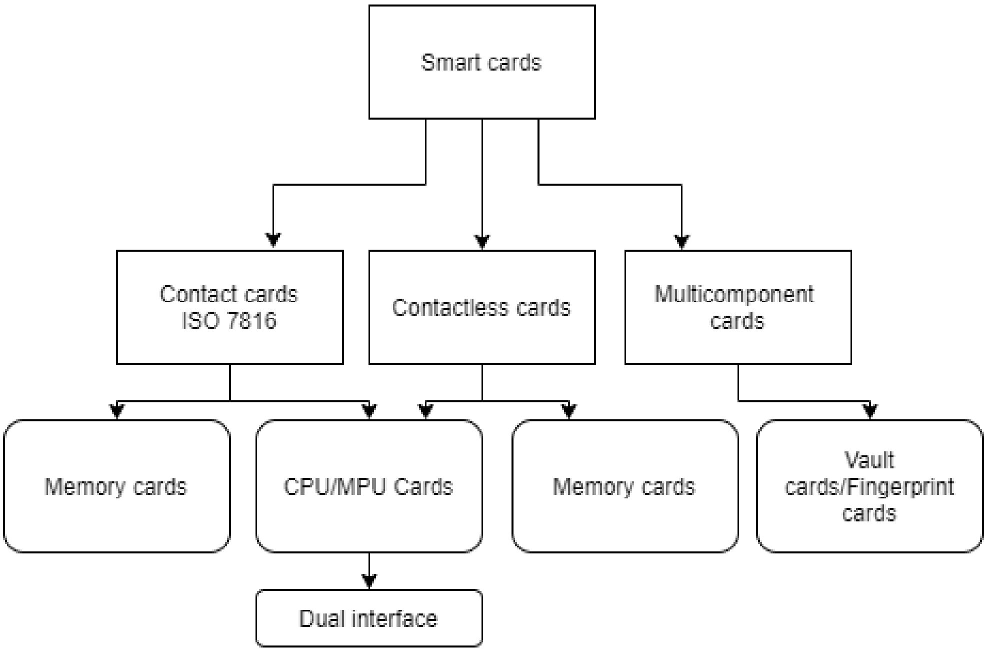


Figure 2.1: Smart card types

2.3.1 Contact Cards

Contact cards are the most common type of smart card. The card is connected to a reader by electronic contacts located on the outside of the card. This connector is bound to the encapsulated chip in the card.

2.3.2 Contactless Cards

Contactless smart cards communicate with a reader without the physical insertion of the card. That is possible thanks to RFID, which is a technology that uses radio waves.

2.3.3 Memory Cards

It is important to know that memory cards have no processing power for data management and cannot manage files in any way. They all communicate through synchronous protocols.

[13] Three types of memory cards are recognized:

- Straight Memory Cards
  - just store data and have no data processing capabilities
- Protected / Segmented Memory Cards
  - built-in logic to control the access to the memory of the card
- Stored Value Memory Cards
  - designed for the specific purpose of storing values
  - incorporate permanent security measures - password keys or logic hard-coded into the chip

2.3.4 CPU/MPU Microprocessor Multifunction Cards

Unlike memory cards, multifunction cards are capable of data processing. Their memory is allocated into independent sections or files assigned to a specific function or application. All that is managed by a microprocessor or microcontroller chip.

The chip is similar to those found inside a personal computer. To be able to manage data in organized structures, it uses a card operating system. Besides typical features, the COS controls access to the on-card user memory. This feature allows the card to be multinational and serve multiple purposes. That brings great convenience for the card user, simply because multiple cards can be replaced by one. [13]

For the purposes of this thesis, one particular feature is important. There are configurations of chips that support PKI. Either by using an onboard math co-processor or Java Card with virtual machine hardware blocks. [14]

A Java Card is a smart card that can be programmed in a high-level language instead of assembly language. Nowadays, it is the most commonly used type of smart card mainly because it is easily programmable, including security methods. [14]

2.3.5 Smart Card Readers

The main purpose of readers is to send and receive information to the smart card. Typically, a reader interfaces with a PC for the majority of its processing requirements. Same as smart cards, we recognize two types of readers depending on the way they communicate. There are contact and contactless readers.

Contact readers require a physical connection to the card. Direct coupling provides relatively fast a reliable communication. A communication protocol and electric interface of contacts cards are defined in ISO7816. [15] ISO7816 protocols specifies:

- *I*SO/IEC 7816-1 specifies physical characteristics for cards with contacts.
- *I*SO/IEC 7816-2 specifies dimensions and location of the contacts.
- *I*SO/IEC 7816-3 specifies electrical interface and transmission protocols for asynchronous cards.
- *I*SO/IEC 7816-4 specifies organization, security and commands for interchange.

Contactless readers work with a radio frequency which allows communication while the card is close to the reader. This type of communication is called NFC. Same as the contact readers, there are set of standards that define the requirements for the smart card, communication, and the reader. The set of standards is named NFC ISO/IEC14443 and it is divided into four separate layers same as ISO7816.
[16] ISO14443 layers define:

- physical characteristics
- radio frequency
- initialization and anti-collision and transmission protocol
- signal interface

Smart cards which use NFC are widely used these days. The most common example from everyday life is credit/debit cards, which usually support both contact and contactless communication. NFC is generally considered less reliable than a physical connection. On the other hand, the NFC chip is less likely to get damaged while you are wearing it in your wallet.

2.4 Convenient Technology for this Thesis

As it is described in chapter 1, the goal of this thesis is to create a plugin for KeePass that adds a second factor into the authentication process. A KeePass password manager is very convenient for this project because it can create new plugins, including key providers. It is one of the very few open-source password managers, which are considered safe at the same time. On top of that, I personally prefer to store the password database locally instead of sending it into the provider's cloud.

The second factor of authentication is the possession factor. Smart cards, especially Java cards, are easily programmable. Besides running your own Java applets on the card, they support cryptographic functions that allow securing the communication between the card and the plugin. Also, it stores a database safely in its memory, protected by a pin. Moreover, the card can be programmable to be very difficult to emulate and to communicate with the plugin exclusively. All this makes it very difficult for the attacker to break this layer of security even if he gets in possession of the card. For these reasons, Java cards proved themselves useful for the purposes of this thesis.

Adding another layer of security into the authentication process makes it significantly more challenging for the attacker to get access to the database.

Chapter 3

Analysis and Design

In this chapter, we analyze the technology and algorithms used for our implementation. Then we describe the KeePass plugin system alongside Java applets. Special attention will be given to communication between the plugin and Java card same as to cryptographic methods used for encrypting the communication channel.

3.1 KeePass Password Manager

KeePass is a free, open-source password manager. It is licensed under GNU General Public License. Besides passwords, KeePass stores all associated data such as usernames, URLs, or notes. Everything is stored in a single database encrypted using the most secure encryption algorithms as ASE-256, ChaCha20, and Twofish. The database is locked with a master key. The master key is based on one or multiple key sources. Each of the sources may be used on its own or in combination with others.

[17] There are 3 possible sources for a master key:

- master password
- key file - any file located on the system or external storage(Smart card, USB drive)
- Windows user account - Windows Data Protection API is used

KeePass is primarily compatible with Windows operating system. However, it can be run on other systems via Mono or Wine. This thesis is focused on KeePass running on Windows operating system. Therefore, our plugin is not compatible with other platforms.

3.1.1 KeePass Plugin System

Keepass offers an extensive plugin system. At the official KeePass website is many plugins and extension created by either user themselves. Plugins are divided into categories, according to their functionality.

[18] KeePass plugin categories:

- Cryptography & Key Providers
  - allow you to use new encryption methods, use different key sources or add 2FA
- Backup
  - provides new methods to backup your database or provide automatisation
- I/O & Synchronization
  - loading, uploading and synchronizing a database with a remote storage provider
- Utilities
  - modify or add functions directly related with using the KeePass app e.g. reminds you to change entry password once in a while.
- Integration & Transfer
  - improving the way KeePass communicates with web browsers or any other software.
- Import & Export
  - Allow you to import/export your database to/from different formats as JSON, CVL etc.

All KeePass plugins need to be written in C#, using the .NET Framework. The reason is that the plugin needs to derive from the KeePass plugin class. By overriding the methods and properties of the plugin class, you can customize the behavior of your plugin. In Figure 3.1 you can see a minimal plugin, which shows how it derives from the KeePass plugin class.

The IPluginHost is an interface providing you access to KeePass's internals as the KeePass main menu or currently opened database. Two functions are more important than the others and always need to be overridden. It is Initialize and Terminate.

In the *Initialize* function is called immediately after KeePass loads your plugin. The first thing you get is an IPluginHost interface reference, and then it is up to you to initialize everything you need. If everything is initialized successfully, you must return true. In case you return false, KeePass will unload your plugin immediately.

The *Terminate* function is called shortly before KeePass unloads your plugin. You cannot abort this process, but you can, and you should use it as the last chance to clean your resources. [19]

Besides already mentioned functions, there are some other important conventions, which must be followed:

- "*The namespace must be named like the DLL file without extension*
- *The main plugin class (which KeePass will instantiate when it loads your plugin) must be called exactly the same as the namespace plus "Ext"*
- *The main plugin class must be derived from the KeePass.Plugins.Plugin base class*" [19]

After all the conventions are followed, you just move your plugin folder including its DLL files into KeePass plugin folder. This folder can be accessed in KeePass by clicking on "Tools" → "Plugins" → button "Open Folder". Then you just restart KeePass and your plugin will be loaded and ready to use.

```
using System;
using System.Collections.Generic;
using KeePass.Plugins;
namespace SimplePlugin
{
        public sealed class SimplePluginExt : Plugin
        {
                private IPluginHost m_host = null;

                public override bool Initialize(IPluginHost host)
                {
                        if(host == null) return false;
                        m_host = host;
                        return true;
                }
        }
}
```

Figure 3.1: A minimal KeePass plugin [19]

3.1.2 KeePass Key Providers

KeePass key providers are the most important category for the purposes of this thesis. They allow you to implement alternative ways of unlocking the database. A key provider plugin derives from the *KeePassLib.Keys.KeyProvider* class. Using the *Add* method of the *KeyProviderPool* class in the *KeyProviderPool* interface, the plugin registers itself into the key provider pool.

As you can see in Figure 3.2, the most important function is *GetKey.* It gets the key from you and uses it for opening the database. A key source is optional, and it is completely up to you where you get it from. In section 2.4 is briefly described why Smart card, particularly Java card is a good choice for storing your database key.

If you look at existing plugins, you find out that there are multiple types of this category of plugins. The first type is focused on quick unlock, e.g., KeePassWinHello. After unlocking the database regularly, the plugin uses Windows Hello technology for reopening the database, so you do not have to use your password every time. This plugin indeed makes KeePass more user-friendly but does not improve the security of your database.

Another type, the one we are about to create, provides a new method to unlock the database. The method can be either used by itself or combined with other already existing methods. As you can read in section 2.1.4, it is highly recommended to use multiple factors of authentication. Therefore, you should always use one of the original methods, such as a master password with your authentication plugin.

3.2 Java Card

Java Card is a Smart Card that allows Java-based applications called applets to be run. This section describes significant parts of Java applets, installation process, communication protocols, and commands, which the card can process.

3.2.1 Java Applets

Java applets are small applications written in Java that compiles to Java bytecode. As you already know, this bytecode is executed within Java virtual machine. Similar to KeePass plugins, some conventions have to be followed in order to create a functional Java card app. In this section, we will look at the Java applet development process and describe how to follow these conventions. [20]

3.2.1.1 Declaring the Package

Typically, you can bundle related Java Card applets into a package. Therefore, it is necessary to specify the package the applet belongs to. To make it

```
using System;
using KeePass.Plugins;
using KeePassLib.Keys;

namespace KeyProviderTest
{
        public sealed class KeyProviderTestExt : Plugin
        {
                private IPluginHost m_host = null;
                private SampleKeyProvider m_prov
                = new SampleKeyProvider();

                public override bool Initialize(IPluginHost host)
                {
                        m_host = host;

                        m_host.KeyProviderPool.Add(m_prov);
                        return true;
                }

                public override void Terminate()
                {
                        m_host.KeyProviderPool.Remove(m_prov);
                }
        }

public sealed class SampleKeyProvider : KeyProvider
{
        public override string Name
        {
                get { return "Sample Key Provider"; }
        }

        public override byte[] GetKey(KeyProviderQueryContext ctx)
        {
                return new byte[]{ 2, 3, 5, 7, 11, 13 };
        }
}
```

Figure 3.2: Simple KeePass key provider [19]

easier to develop your own packages, sample packages are usually part of the development kits and can be used. [20]

```java
package com.sun.javacard.samples.wallet;
```

Figure 3.3: Package declaration [20]

3.2.1.2 Import of the Java Card Framework

A set of classes and interfaces for Java Card application programming is defined by Java Card technology. A package named javacard.framework defines classes and interfaces that are essential for developing Java Card applets. The most important class in this package is the base **Applet** class.

```java
import javacard.framework.*;
```

Figure 3.4: Import of Java Card framework [20]

3.2.1.3 The base Applet Class

All Java Card applets extend from the base class, which is defined by javacard.framework package. The base **Applet** class defines methods that a Java Card applet uses to communicate with the JCRE. javacard.framework.Applet methods:

- deselect()
- install (byte[] bArray, short bOffset, byte length)
- process (APDU apdu)
- register()
- select()

```
public class ClassicApplet1 extends Applet
```

Figure 3.5: Extending the base Applet class

3.2.1.4 Applet Installation

The base **Applet** class defines the install method. This method must be implemented in all Java Card applets. You can imagine it as the **main** function in programming languages like C or C++. This method is always the very first called by the JCRE. The purpose of the method is to create an instance of an applet and give it control. What the implementation of this method does is up to the applet designer. However, it is necessary to call the applet's constructor to create and initialize an instance of the applet.[20]

```
public static void install(byte[] bArray,
                short bOffset, byte bLength)
{
new ClassicApplet1(bArray, bOffset, bLength);
}
```

Figure 3.6: javacard.framework.Applet install method

As you can notice in Figure 3.6, the **install** method accepts three parameters:

- "*bArray is an array of type byte that contains installation parameters.*"
- "*bOffset is a variable of type short that contains the starting offset into the array.*"
- "*bLength is a variable of type byte that contains the length, in bytes, of the parameter data in the array.*"

Simply speaking, the method accepts byte array with installation parameters, which usually includes applet configuration values as the size of internal files and applet initialization values. [20]

3.2.1.5 Selecting and Deselecting the Applet

After the Applet is successfully initialized, it waits in a suspended state until JCRE selects it. This process starts when the JCRE receives a *SELECT* APDU command. This command is recognized by a specific header value and once it is recognized JCRE compares the AID values in the data field with a list of registered applets on the Smart card. If the applet with matching AID is found, the *select* method from *javacard.framework.Applet* class is called. The method informs JCRE if the applet is ready to process a request, by returning true. If it is not, it returns false. How the *select* method decides whether or not return true, it up to the applet's constructor. Typically, it checks if the number of remaining tries to unlock the card by using a pin is greater than zero. This way, if anybody were trying to authenticate himself with an incorrect pin, the applet refuse to be selected. [20]

It is possible to run only one applet at a time. So in case you need to use another applet registered on the card, you need to deselect the one you are currently using. This process is automatically triggered once the APDU *SELECT* command with different AID is received. Similarly to the *select* method, the *deselect* method is defined in the base Applet Class. On the other hand, *deselect* method is a void function, which means it does not return anything. It is supposed to allow the applet's constructor to clean up all the resources. Typically, it serves to reset a pin flag that indicates if the PIN is validated.[20]

```
public void deselect() {
    pin.reset();
}
```

Figure 3.7: Deselect method example

3.2.2 Communication with Applet

All incoming requests are sent to the Java card through the APDU object. This object is an instance of the *javacard. framework.APDU* class. Understanding an APDU format is crucial for Java Applet development. The host application sends APDU Command to request any action from the Java Card. The APDU command consists of multiple fields, as you can see in the table 3.1.

The first byte of the APDU command is CLA, which indicates a category of APDU commands. The second byte determines an instruction that is supposed to be executed. The third and fourth bytes are instruction parameters for the command, e.g., offset into a file at which to write the data. Those four bytes are mandatory, unlike the following bytes. [21]

There is the “Lc field”, which carries a piece of information about the number of bytes in the data field. The “Data field” is supposed to be as big as the Lc byte says. However, if there is no command data, both of these fields are absent. Similarly, if there are no data expected in the APDU response, the “LE field” is not present in the APDU command. [21]

An APDU response has only two fields. The first of them is optional, and it is called “Response data” field. Therefore, if no data is expected, only the “Response status” is sent back to the host app. The response status a return value that informs the host app if the instruction were executed or if there were an issue. This is necessary to keep in mind, if you are developing an app along with an applet. [21]

All APDU commands are processed by the applets *process* method. This method is defined, along with the earlier mentioned, in *javacard.framework.Applet*. As you can see in Figure 3.8, the method accepts only one parameter, and that is the APDU object. To properly process the APDU object, you must first get a pointer to the APDU buffer encapsulated in the object. All the data from an APDU command, including its header, are written into the buffer by JCRE after receiving the APDU command. As the given example of implementation of the *process* method shows, a switch method is typically used for recognizing which of the implemented instructions should be executed. [20]

```
public void process(APDU apdu) {
    byte buffer[] = apdu.getBuffer();
    switch (buffer[ISO7816.OFFSET_INS]) {
        case INSTRUCTION1:       INSOneFuntion(apdu);
                            ISOException.throwIt
                (ISO7816.SW_NO_ERROR);
        case INSTRUCTION2:       INSTwoFunction(apdu);
                            ISOException.throwIt
                (ISO7816.SW_NO_ERROR);
        default:       ISOException.throwIt
                (ISO7816.SW_INS_NOT_SUPPORTED);
    }
}
```

Figure 3.8: Process method

Table 3.1: Format of APDU commands and responses

APDU Command		
Field	**Description**	**Length (bytes)**
CLA	Instruction class - indicates type command	1
INS	Instruction code - indicates specific command	1
P1- P2	Instruction parameters	2
LC Field	Number of bytes present in the data field	0, 1 or 3
Data Field	Command data	Command data length
LE Field	Number of data expected in APDU Response	0, 1, 2 or 3
APDU Response		
Response Data	Response Data	Response length
Response Status	Status bytes SW1 and SW2	2

3.3 Security Analysis of Naive Implementation

As you already know, this thesis aims to use a Java card as the second factor of authentication to unlock the KeePass database. A naive, simple implementation of this process will be used to point out possible security threats.

3.3.1 Description of the Implementation

The naive implementation consists of Java Applet and KeePass plugin. The applet is very simple. After receiving the correct APDU command, it sends a 6-byte array in the APDU response, used as a database key in the plugin. The key is hardcoded in the Java applet code by the author of the applet. The KeePass plugin is from “Key Providers” category, which is described in subsection 3.1.2. That means its goal is to get the database key from the Java Card and use it to unlock the database. This process gets done in a few simple steps:

- KeePass plugin identifies if there is a smart card reader connected to the PC
- Detect Smart card and send APDU command to select the authentication applet
- The plugin sends APDU command in order to receive a database key

- Unlock the database with the received database key

This very simple implementation provides some level of protection, but it has significant issues that will be pointed out in this section.

3.3.2 Threat Model

To analyze possible threats, you need to look at all of the components participating in the authentication process and analyze their communication. As figure 3.9 shows, there are three of them. Each one of them has its weak points in this implementation.

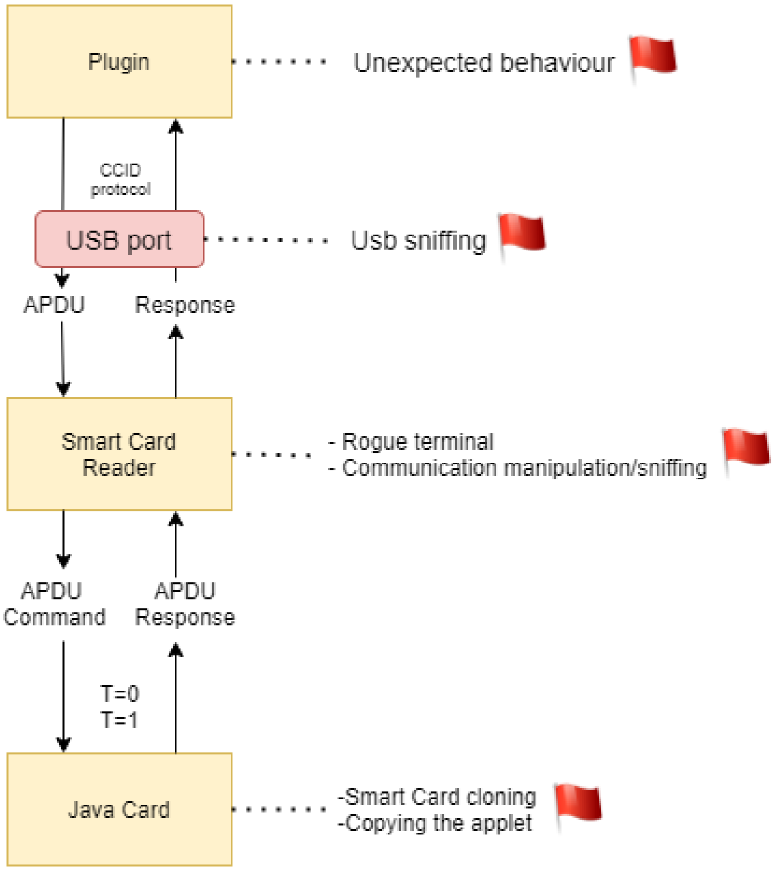


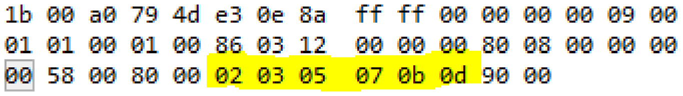
Figure 3.9: Threat model of naive implementation

First of all, with the Java applet described above, the Smart Card is easily emulated. That presents a significant security threat, especially with other factors such as unencrypted communication or allowing the attacker to get physical access to your card. All the attacker needs to know to emulate the behavior of your card is the applet's AID (Application Identifier) and the database key.

One of the ways to get this information is to sniff the communication between the Smart Card and other participants. That can be done in multiple ways. One option is an attack by a rogue terminal, where the Smart Card reader either record or influence the communication between the Smart Card and the off-card entity. That means the reader can be a weak point, and it needs to be considered as a threat. Another way of getting the information needed for emulating the card requires physical access. As the AID is hard-coded in the KeePass plugin, the attacker can find it there. Once he has the AID, there are less than 2^8 possibilities for *INS* byte. Therefore, it is a matter of minutes to try all possible values of the instruction byte and try which one of them makes the Java card return the database key.

Although, the easiest way to get needed information is by scanning the USB port to which the Smart Card reader is connected. This attack is called USB sniffing, and it does not require any special equipment nor advanced knowledge. That makes them very likely to happen and presents a significant security threat. Especially because in this implementation, no encryption is used. Moreover, the USB communication is typically not encrypted, and CCID(chip card interface device) protocol is no different. That means the attacker can create a copy of your Java Card by following these steps:

1. Start scanning the USB port - It can be a background process, so a common user probably will not notice anything unusual
2. Wait until the user uses his card to open the KeePass database
3. Read the communication between the Java Card and the plugin - find AID and Database key
4. Write an applet that responds in the same way as the original one and has the same AID
5. Install the applet on the new Java Card



```
1b 00 a0 79 4d e3 0e 8a  ff ff 00 00 00 00 09 00
01 01 00 01 00 86 03 12  00 00 00 80 08 00 00 00
00 58 00 80 00 02 03 05  07 0b 0d 90 00
```

Figure 3.10: Stolen database key by using USB sniffing

This attack requires just a basic understanding of Java Applets development and the KeePass plugin system. As Figure 3.10 shows, the database key is sent in plain text with no encryption. And with very little communication between the plugin and the applet, the key can be found pretty quickly.

In addition, there is an even easier way to unlock the database. If the attacker sniffs the database key, there is no need to develop the Java applet.

The original KeePass plugin can be replaced with a plugin with the database key hardcoded in its code. Therefore, the attacker does not even need a Java card to open the database. Besides, the database key is only a 6-byte array created by the author of the applet. That does not meet the requirements of the safe as it is described in section 3.7.

To prevent this type of attack, the card's behavior needs to be much harder to emulate. The first thing that makes it harder for the attacker is implementing a PIN. If a PIN protects the card, it stops the attacker from trying all possible values of *INS* and getting a key. On the other hand, as long as the communication is not encrypted, the attacker is able to read the PIN along with the AID and the database key. Therefore, the communication needs to be encrypted in both directions.

In conclusion, the biggest threat presents non-encrypted communication along with a very simple applet that is easy to emulate. The diagram 3.11 shows the attacker's options to overcome this layer of security. As was already mentioned, one of the goals of this thesis is to prevent the attacker from doing so. That can be done by encrypting the communication, implementing *Secure channel*, protecting the card with a PIN, using longer and correctly generated database key, and handling unexpected behavior of both the card and the KeePass plugin.

3.4 Secure Channel Protocol

As is pointed out in the previous section, it is crucial to prevent the database key from being stolen. The biggest security threat presents non-encrypted communication, but there is more. Besides scanning the USB port and reading the communication, the attacker can influence the communication and perform a "Man-in-the-middle" attack. Secure Channel Protocol is supposed to prevent all this from happening. The protocol provides the three followings levels of security:

- **Entity authentication**
  - the card authenticates the off-card entity, and the off-card entity may authenticate the card by proving they both have knowledge of the same secret
- **Integrity and Data origin authentication**
  - receiving entity ensures that the data being received actually came from the authenticated sending entity
- **Confidentiality**
  - the transmitting data is not viewable by an unauthorized entity

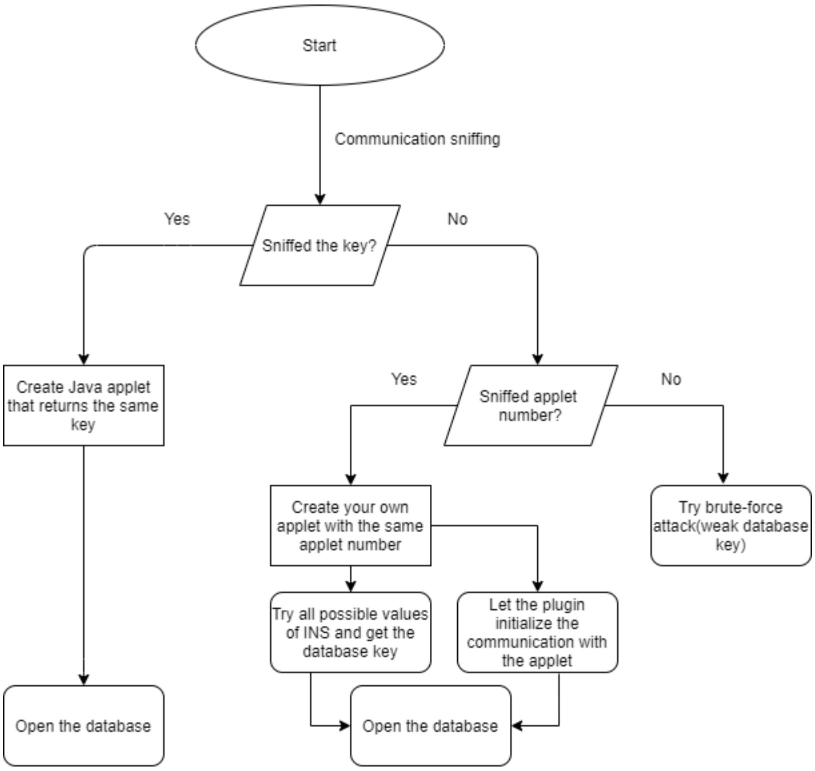

Start
Communication sniffing
Sniffed the key?
Yes
No
Create Java applet that returns the same key
Sniffed applet number?
Yes
No
Create your own applet with the same applet number
Try brute-force attack(weak database key)
Try all possible values of INS and get the database key
Let the plugin initialize the communication with the applet
Open the database
Open the database

Figure 3.11: Attacker's options to emulate the Java Card

To achieve all this, a **Security Domains** are used. The domains act as the off-card representatives of off-card authorities. Therefore, once the SCP is initialized, the KeePass plugin actually communicates with the security domain, not the Java applet itself. This way, the applet is protected against processing APDU commands from the unauthorized off-card entity and ensures the card will not provide any confidential information to an attacker. [22]

3.4.1 Entity Authentication

Off-card entity authentication is extremely useful for the purposes of this thesis. The Java Card must not send the database key to anybody unless it is evident it is the KeePass plugin that it is communicating with. This way, it keeps the database key safely stored in the card's memory with minimal options for the attacker to steal the key.

The Secure Channel initiation and off-card entity authentication imply the creation of session keys. Those are derived from card static keys that have to be **shared secret** between both participants. Therefore, the static keys need to be safely stored. Because if the attacker gets to know them, he can act as the original off-card entity and the Secure Channel is compromised.

3.4.2 Explicit Secure Channel Initiation

The off-card entity may explicitly initiate the Secure Channel. Two commands allow the entity to do so. The initialization always starts with INITIALIZE UPDATE command sent by the off-card entity. This command is coded according to the table 3.2. In “Data field” is a *host challenge*, which is random data unique to this session. The card on receipt of this challenge generates the session keys and first cryptographic value. Then the card cryptogram, along with the Sequence Counter, The Secure Channel Protocol Identifier, and the “card” challenge, is sent back to the off-card entity via APDU Response.

Table 3.2: INITIALIZE_UPDATE command

Field	**Value**	**Meaning**
CLA	80-83 or C0 - CF	CLA - Secure messaging ISO/IEC 7816 standard
INS	50	INITIALIZE UPDATE
P1	xx	Key Version Number
P2	00	Reference control parameter P2
Lc	08	Length of host challenge
Data	xx xx..	host challenge

During the processing of the APDU response that includes the “card” challenge, the off-card entity creates a second cryptographic value, also called *host cryptogram*. The cryptogram is passed to the card by using *EXTERNAL AUTHENTICATE* command that is coded as table 3.3 shows. Along with the cryptogram, the off-card entity creates and sends a *Message Authentication Code*.

Table 3.3: EXTERNAL AUTHENTICATE command

Field	**Value**	**Meaning**
CLA	84 - 87 or E0 - EF	CLA - Secure messaging ISO/IEC 7816 standard
INS	82	EXTERNAL AUTHENTICATE
P1	xx	Security level
P2	00	Reference control parameter P2
Lc	10	Length of host cryptogram and MAC
Data	xx xx..	Host cryptogram and MAC

After the card receives the command, it verifies the MAC and uses it to create the Initial Chaining Vector used for the verification of C-MAC and/or R-MAC.

In summary, during the Secure Channel initiation, the card and the KeePass plugin (off-card entity) verify each other by generating and verifying each other's cryptograms. Furthermore, they generate session keys and initialize ICV, later used for verification of C-MAC or R-MAC.

3.4.3 Message Integrity

The message integrity is provided by verifying a MAC. The MAC can be generated by both the off-card entity and the card. The entities both have the same session keys. That means the receiving entity is able to perform the same operations and compare its generated MAC with the one it received.

During the Explicit Secure Channel Initiation, the SCP02 mandates the use of MAC for *EXTERNAL AUTHENTICATE* command.

To algorithm ensuring the integrity is Triple-DES in CBC mode described in section 3.6.2. All 8 bytes of an Initial Chaining Value is equal to zero, and it is always a part of the *EXTERNAL AUTHENTICATE* command. [22]

3.4.4 Message Data Confidentiality

The message data confidentiality is provided by encrypting the data field by applying multiple chained DES operations. For the encryption, the session keys generated during the Secure Channel Initiation are used. The data can be encrypted both ways if the Java Card supports it. Commonly, the card does not support data encryption in the APDU Response data field.

3.4.5 Security Levels

The Security level determines how the Secure Channel Protocol operates. During the Secure Channel Initiation, the minimum acceptable secure messaging protection is set. However, the security level can be increased or decreased for a single message.

The Security level is established based on a bitmap combination of the following values: C_MAC, R_MAC, AUTHENTICATED, and C_DECRYPTION. Each value determines if the corresponding operation needs to be performed. If the C_MAC value equals one, the command MAC check is required after receiving the message from an off-card entity. R_Mac level corresponds to Response MAC check, and C_DECRYPTION informs that the data field needs to be decrypted using the session keys. [22]

3.4.6 Cryptographic Keys

A Secure Channel Protocol is based on the shared secret principle. The secret is 3 double length DES keys that are being used only during the Secure Channel Initiation. Each one of them has its own purpose.

The Secure Channel encryption key that the protocol uses for Secure Channel authentication and encryption. The Secure Channel MAC key that is being used for MAC verification and generation. And the sensitive data during the initiation is being encrypted and decrypted using the data encryption key. [22]

Table 3.4: Secure Channel keys

Key	**Usage**	**Length**
Secure Channel encryption key (S-ENC)	Authentication & encryption(DES)	16 bytes
Message authentication code key (S-MAC)	MAC varification and generation	16 bytes
Data encryption key(DEK)	Sensitive data encyption and decryption	16 bytes

3.4.7 Usage of Secure Channel for this Thesis

The Secure Channel protocol is widely used in the implementation part of this thesis. It needs to be pointed out that this protocol has three versions: SCP01, SCP02, and SCP03. The main difference between the version is the ciphers they use for encryption and decryption.

SCP02 uses a 3DES cipher, and the NIST organization recommends replacing it with a newer SCP03 that uses a cryptographically more secure AES cipher. However, only a new Java Cards support SCP03, and there are still a significant amount of cards that are being used and support only SCP02. The authentication applet is supposed to extend the functionality of Java Cards that are currently being used. Moreover, all Java Cards that support SCP03 also support SCP02.

Even though SCP03 is undoubtedly more secure, currently, it is better to ensure all kinds of users can use the the authentication applet. Therefore, in this thesis, SCP02 is used. As the 3DES cipher is recommended to be replaced by more secure AES-256, additional security measures need to be added. For this reason, the database key itself is encrypted by using the RSA cipher during the transmission.

In conclusion, the compromise between common use-cases and security needed to be found. That is why the SCP02 with other security measures as the use of RSA encryption that partially compensates its drawbacks.

3.5 Communication details

In this section, the communication between the Java authentication applet and the KeePass plugin will be described.

In the beginning, the applet needs to be selected by using the SELECT Command(3.2.1.5). If the applet is selected successfully, the applet initiates the SCP02 as it is described in section 3.4.2.

From the moment the SCP is established, the communication between participants is encrypted. Therefore, confidentiality information can be sent as a PIN code that protects the Java Card. After receiving the correct PIN, the authentication is over. The user proved knowledge of the shared secret and the PIN.

For additional encryption of the database key during the transmission, the RSA cipher is used. Hence, the plugin needs to provide its public RSA key to the card. After receiving the Public key, the plugin requests for the database key itself.

After the plugin receives the key, it opens the database. And the connection terminated. The plugin is deselected, and the card is locked again.

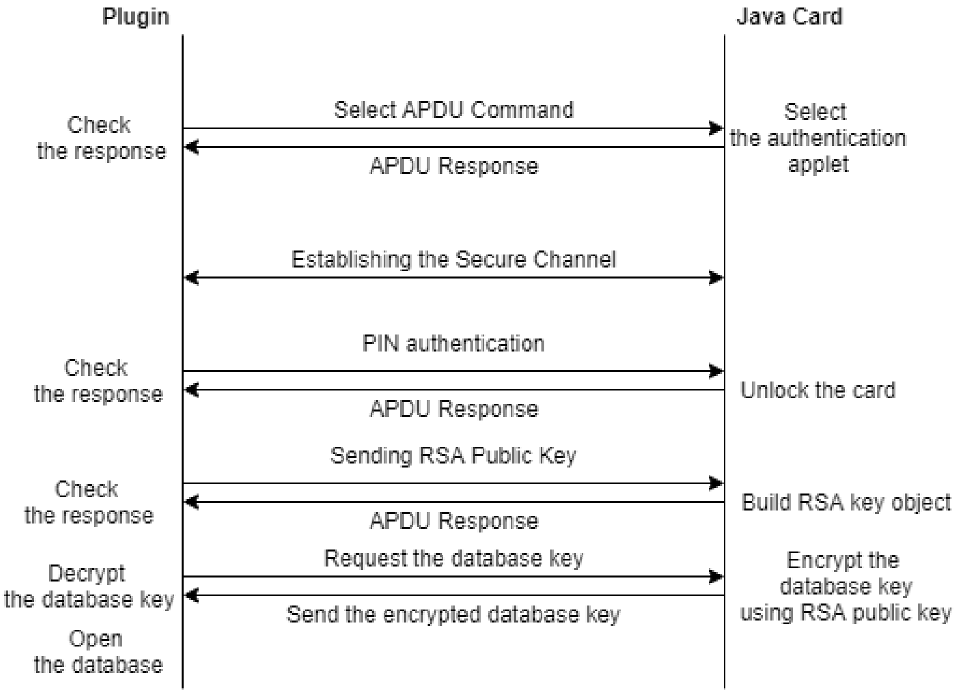


Figure 3.12: Communication diagram

3.6 Cryptography

3.6.1 RSA

The name of this encryption algorithm stands for the MIT scientists (Rivest, Shamir, and Adleman) who first described it in 1977. It is an asymmetric algorithm, which means it uses two different keys to perform the encryption and decryption. Its security relies on the practical difficulty of factoring two prime numbers. also called the *Factoring problem*. It cannot be solved in a reasonable amount of time, thus there is no known way how to break RSA encryption if a large enough key is used.

The RSA algorithm involves four steps. In the first steps the keys generation takes place. They are typically 1024, 2048 or 4096 bits long and the following algorithm is used to generate them. [23]

1. Two distinct prime numbers **p** and **q** are chosen at random
2. The modulo **n=pq** is computed
3. Compute Φ(n), where Φ(n) = (p-1)(q-1)
4. Choose integer **e**, 1 < **e** < Φ(n) and gcd(e, Φ(n)) =1
   - gcd = greatest common divisor
5. Determine integer **d** where e · d≡ 1 (mod Φ(n))

After the key generation, the key distribution takes place. The integers **n** and **d** are used as a Private key. Public key consists of the integers **n** and **e**. The Public key is published, while the Private key needs to stay a secret. [23]

$$c \equiv m^e \ (mod\, n);\ c = ciphertext,\ m = plaintext \tag{3.1}$$

Figure 3.13: RSA encryption

Anyone who wants to send you encrypted data, use your Public key to encrypt the data. After the data is received, you decrypt them by using your Private key. Because you are the only one who has access to your Private key and it cannot be derived from the Public key, there is no way the data will be decrypted by anyone else in a reasonable amount of time. [23]

Even the increasing computing power of today's computers, the RSA is still considered safe and it is widely used. It is very convenient cipher for establishing secure connections between two participants and therefore it is perfect for the purposes of this thesis.

$$m \equiv c^d \ (mod\ n) \tag{3.2}$$

Figure 3.14: RSA decryption

3.6.2 Triple Des in CBC Mode

DES stands for “Data encryption standard” and the algorithm was developed in the early 1970s by IBM company. It is a symmetric-key algorithm, which means it uses the same key during the encryption and decryption.

The algorithm is based on confusion and diffusion. The confusion means that each bit of the ciphertext depends on several parts of the key. This property is needed for hiding the relationship between the ciphertext and the key. The diffusion that if a single bit of the plaintext is changed, then multiple bits in the ciphertext change. This effect is supposed to hide the statistical relationship between the ciphertext and plaintext.

DES is a block cipher, and the data is encrypted in 64-bit blocks. Therefore, the input is 64 bits of plain text that is encrypted into 64 bits of ciphertext. The size of the initial key is 64 bits, but every eighth bit of the key is a parity bit. Thus, the actual key is only 56-bit long.

The algorithm consists of 16 steps. The step of the algorithm is called a round. In each round, both confusion and diffusion are performed on the 32-bit blocks.
DES algorithm:

1. the plain text block is handed over to an IP(Initial Permutation) function
2. the IP is performed on plain text
3. the permuted block is split in half; Left Plain Text(LPT) and Right Plain Text(RPT)
4. both blocks go through 16 rounds of the encryption process
5. LTP and RPT are rejoined
6. A Final Permutation is performed on the block that consists of LTP and RPT

Triple DES expands the key size by running the algorithm in succession with three different keys. That means the plain text is encrypted in 48 rounds, and the resulting key is 168 bits. Simply speaking, triple DES is just running the DES algorithm three times with three different keys.

Block ciphers can operate in numerous modes: Electronic Codebook, Cipher Block Chaining, Cipher Feedback Block, Output Feedback, Counter

Method. The Secure Channel Protocol uses the 3DES in CBC mode. This mode is supposed to expand the diffusion by chaining the blocks of ciphertext. Each block of plaintext before being encrypted is modified(XORed) by the previous block of ciphertext. For the very first block, an Initial Vector is used.

3.7 Key Handling & Control

In this section, the importance of a strong database key will be explained. It needs to be pointed out that the attacker does not need any Java Card as he can develop his own Applet and perform a brute-force attack where all possible combinations of the key are tried for opening the database. That means the longer the key is, the longer it takes to find the right combination.

On the other hand, we are limited by the size of the Java Card storage. Although, even with older Java Cards, there is enough storage for 2048 bytes long database key. Just to put this length in context, even the biggest supported RSA key is 4096 bits. Thus, it is reasonable to consider the key, which is four times longer, resistant against brute-force attacks as long as it is generated by using a secure algorithm.

There are various C# libraries that use secure algorithms to generate keys. However, generating the key in the off-card entity means additional exposure during the transmission to the card. Hence, it is better to minimize the exposure and generate the key on the card itself as long as it provides a secure algorithm.

The majority of Java Cards supports Class RandomData from javacard.security library. The class provides an option to generate random data by using a secure random algorithm. Therefore, there is no reason to generate the key outside the card and risk additional exposure during the transmission. [24]

Chapter 4

Implementation

This chapter describes the implementation of the final part of this thesis. Both, the KeePass authentication plugin and the Java Card Applet, are described with special focus on implementation of security measures described in the analysis part of this thesis.

4.1 KeePass Authentication Plugin

The authentication plugin is a key provider type of plugin from KeePass plugin system described in section 3.1.2. This section describes the details of the implementation, how it implements required security measures and what action take place while you use the plugin.

4.1.1 SCP Initiation and Usage

For a SCP initiation the plugin uses an open source implementation, imported as a DLL(available at [25])The implementation allows the plugin to create instance of GlobalPlatform class. The class stores information about a Secure Channel that is being currently used. It also provides an option to create APDU commands as *INITIALIZE UPDATE* or *EXTERNAL AUTHENTICATE* and process card responses. All APDU commands needs to be processed by a **wrap method**, that modify the APDU according to the current Security Levels.

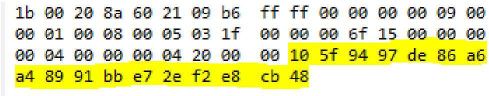
As you can see in figure 4.1, the Secure Channel established with the Java Applet is set to use C_DECRYPTION and C_MAC. Thus, if you use a **wrap** method on a APDU command, the data field is encrypted and the C_MAC is added to the APDU.

With established Secure Channel, the communication is encrypted by using the three session keys. A figure 4.2 shows the APDU command with instruction code equal to 0x20. The command is supposed to unlock the card by using a correct PIN. As you can see, the PIN(1,2,3,4) is encrypted as the

```
CommandAPDU initUpdate = gp.CreateInitUpdateCommand(scKeys,
SecurityLevel.C_DECRYPTION| SecurityLevel.C_MAC,
GlobalPlatform.SCP_02, GlobalPlatform.IMPL_OPTION_I_15);
```

Figure 4.1: Create initialize update command

command was modified before the transmission by the **wrap** method from the Open Source implementation.



```
1b 00 20 8a 60 21 09 b6  ff ff 00 00 00 00 09 00
00 01 00 08 00 05 03 1f  00 00 00 6f 15 00 00 00
00 04 00 00 00 04 20 00  00 10 5f 94 97 de 86 a6
a4 89 91 bb e7 2e f2 e8  cb 48
```

Figure 4.2: Encrypted Data Field in APDU command

The SCP is based on shared secret that is represented by three keys. Therefore, the keys needs to be stored somewhere in your computer. As it cannot be guaranteed that the attacker will not get access to the file where the keys are stored, they are encrypted by using a *ProtectedData* class. It provides a data protection by user credentials. Simply speaking, the data encrypted by using this class, can only by decrypted by the process that is running under the same Windows user. The drawback of this solution is the plugins is no longer compatible with other platforms than Windows. If you want to use the applet on other platforms, the plugin can be modified and use different platform specific encryption.

4.1.2 RSA Implementation

The RSA cipher is used to encrypt the database key during the transmission from the Java Card to the authentication plugin. It means the plugin needs to generate RSA key pair and provide the card with its public key.

The key pair is generated via *RSACryptoServiceProvider* class that is part of the System.Security.Cryptography. Naturally, the longer the RSA key is, the better for the security of the communication. However, the RSA key needs to be stored in the Java Card with a very limited storage size. Thus, the size of the key is 2048 bits, which is a convenient compromise.

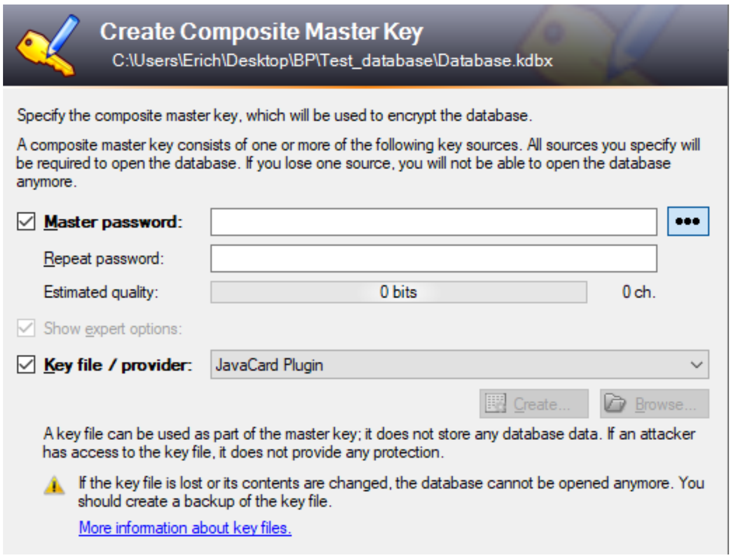
After the key pair is generated, it needs to be sent to the Java Card. The issue is, it cannot be sent in a single APDU as the length of a data field is only 256 bytes. Moreover, the SCP needs to add a C_MAC into the data field to allow the card to verify it is the authentication plugin that sends the key. For these reasons, the RSA key is transmitted in four 64-byte parts. RSA exponent is sent separately after the modulus. After the card receives the plugin's public key, it is used for the database key encryption.

4.1.3 Database Key Transmission

The length of the database key is 2048 bytes. Thus, it has to be divided into smaller parts in order to be transmitted in 256-bytes data fields. Each part is encrypted by using the plugin's RSA public key before it is sent as the section 4.2.3 describes. After each part of the key is received, it is decrypted and stored. Once the key is complete, then it is used to open the KeePass database.

4.1.4 Creating the Database

If you decide to use the authentication plugin to protect your database, you need to select it during the creation of the KeePass database, as is shown in Figure 4.3. The applet is supposed to be used as a 2FA. Therefore it is strongly recommended to use it along with the knowledge factor(Master password).



Create Composite Master Key

C:\Users\Erich\Desktop\BP\Test_database\Database.kdbx

Specify the composite master key, which will be used to encrypt the database.

A composite master key consists of one or more of the following key sources. All sources you specify will be required to open the database. If you lose one source, you will not be able to open the database anymore.

☑ Master password:

Repeat password:

Estimated quality: 0 bits 0 ch.

☑ Show expert options:

☑ Key file / provider: JavaCard Plugin

A key file can be used as part of the master key; it does not store any database data. If an attacker has access to the key file, it does not provide any protection.

If the key file is lost or its contents are changed, the database cannot be opened anymore. You should create a backup of the key file.

More information about key files.

Figure 4.3: Plugin Selection in KeePass

Once the plugin is selected, the plugin initiates the communication with the Java Card. The plugin gives you a list of available Smart Card readers. After you choose the correct reader, the plugin sends the SELECT APDU command(4.4) and establishes Secure Channel as it is described in section 4.1.1.

With a successfully established Secure Channel, the communication is encrypted. Therefore, it is safe to send APDU with the required PIN code and

```
var apdu = new CommandApdu(IsoCase.Case3Short,
isoReader.ActiveProtocol)
{
    CLA = 0x00,
    INS = Constants.SelectAppletINS,
    P1 = 0x04,  // Parameter 1
    P2 = 0x00,  // Parameter 2
    Data = new byte[] { 0x01, 0x02, 0x03, 0x04, 0x05, 0x00 },
};
```

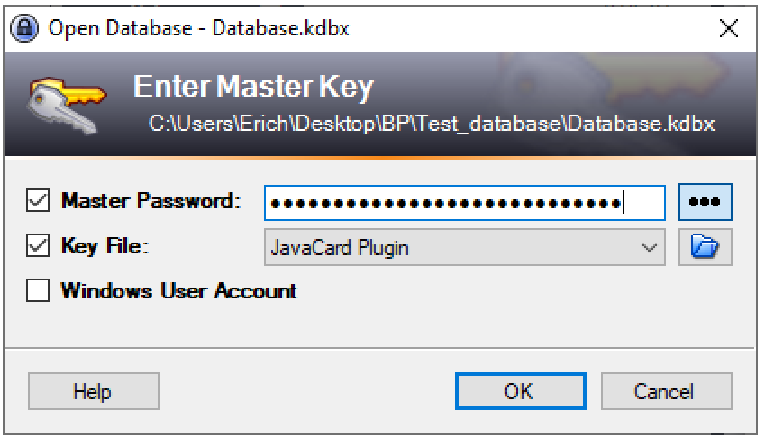
Figure 4.4: SELECT APDU command

unlock the card. If the PIN is incorrect, the plugin terminates the authentication process and interrupts the communication with the Java Card.

If the communication is successfully established and the card unlocked, the plugin sends an APDU command to generate a *Database key* and transmit it to the plugin so it can be set as a database key.

4.1.5 Unlocking the Database

Unlocking the database is very similar process to its creation. After you select the applet as figure 4.5 shows, the communication with Java Card is established exactly the same as in the previous chapter.



Open Database - Database.kdbx

Enter Master Key

C:\Users\Erich\Desktop\BP\Test_database\Database.kdbx

☑ Master Password: ●●●●●●●●●●●●●●●●●●●●●●●●●●●●●

☑ Key File: JavaCard Plugin

□ Windows User Account

Figure 4.5: Selection of the JavaCard plugin

First of all, the Secure Channel is established. Then the plugin asks you for a PIN code and provides it to a card so you can be authenticated. If both of these operations are successful, the plugin sends its RSA public key to the card. Then the encrypted database key is transmitted from the card to the plugin and is used for unlocking the database.

In summary, the only difference is that the database key already stored on the Java Card is transmitted to the plugin instead of generating a new database key.

4.1.6 Additional Features

Two additional features were implemented for the authentication plugin. The first feature allows you to change the PIN that protects the card. The default PIN is "1234", which means it should be changed as soon as possible.

The second feature is called "Invalidate Database Key". It sends an APDU command that invalidates the key that is stored on the Java Card. This feature should only be used if you want to use the card for a new database and generate a new database key.

4.2 Java Card Applet

This section describes the implementation of Java Card authentication applet. It shows how it encrypt the communication and describes security measures as PIN code.

4.2.1 Secure Channel Protocol Implementation

Secure Channel Protocol is implemented in the Java applet by using an interface that is imported from *org.globalplatform*. The Secure Channel Interface is exposed through the *GPSystem.getSecureChannel* method.

To establish the secure channel, the INITIALIZE UPDATE and EXTERNAL AUTHENTICATE commands need to be processed as section 3.4.2 describes. Both these commands are processed by *processSecurity* method. Besides processing the commands, the method generates corresponding responses and sets a Security Level bit map that indicates whether Entity Authentication has occurred and what level of security will be applied.

Once the Secure Channel is established, the authentication plugin sends all commands modified by using method *wrap*. Considering the Security Level the plugin uses, the data field is encrypted with session keys, and the C_MAC needs to be verified. To be able to read the command data, the *unwrap* methods needs to be invoked. Besides the data decryption, the method verifies if the C_MAC is correct to make sure the command was sent by the KeePass plugin.

```
switch (buf[ISO7816.OFFSET_INS])
{
        case INIT_UPDATE:

    case EXT_AUTHENTICATE:
        inlength = apdu.setIncomingAndReceive();
        // process the data from plugin - creating a session key
        respLen = sc.processSecurity(apdu);
        apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, respLen);
        break;
}
```

Figure 4.6: Usage of processSecurity method

4.2.2 PIN Authentication

In order to unlock the card, the PIN is required. The PIN is implemented by using a **OwnerPIN** object. The object defines methods as *isValidated*, *check* or *getTriesRemaining*. Method *isValidated* is at the beginning of each instruction case and check if the PIN has been validated. If not, the method returns false and the applet throws *ISOException* with corresponding APDU Response code.

```
OwnerPIN pin = new OwnerPIN((byte) 3, (byte) 4);
```

Figure 4.7: OwnerPIN initialization

```
if(pin.check(buf, ISO7816.OFFSET_CDATA, (byte) 4)){
        ISOException.throwIt(ISO7816.SW_NO_ERROR);
}else{
    ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
     }
```

Figure 4.8: PIN verification

To validate the PIN, the *check* method is used4.8. If the PIN is correct, it changes a flag in OwnerPIN class that is checked by *isvalidated* method. In case the PIN is incorrect, the flag stays the same and the method return false.

```
if(!pin.isValidated()){
    ISOException.throwIt(IncorrectPIN);
    break;
}
```

Figure 4.9: Usage of isValidated method

If incorrect PIN is inserted three times, the JCRE deselect the applet and terminate the communication with the plugin. If you try to select the applet again, it refuses to get selected thanks to the overridden SELECT method(4.10).

In summary, you need to authenticate yourself in order to communicate with a Java Card applet. Moreover, if incorrect pin is inserted three times, you will not be able to select the applet anymore. Thus, the database key is permanently lost.

```
    public boolean select() {
        if ( pin.getTriesRemaining() == 0 ){return false;}
        return true;
    }
```

Figure 4.10: Implementation of Select method

4.2.3 RSA Encryption in Java Applet

The RSA cipher encrypts a key from the KeePass database before being transmitted to the plugin. The RSA encryption is provided by a class named *Cipher*. The class forms the core of the Java Cryptographic Extension framework and is defined in *javacardx.crypto* library.

The first thing that you need to do in order to use this class for encryption is to use method *getInstance*. As the name of the method suggests, the method returns an instance of the Cipher class. In this thesis, the instance of the class is named “cipherENC” as the figure 4.11 shows.

Once the instance is created, you need to create the RSA key object necessary for the encryption. The RSA key object is an instance of an-

```
Cipher cipherENC = Cipher.getInstance(Cipher.ALG_RSA_PKCS1,false);
```

Figure 4.11: Usage of getInstance method

other class named KeyBuilder, imported from *javacard.security*. The instance of Keybuilder is used by the Cipher class to encrypt the database key. To build a key object consisting of the plugin's RSA Public Key, you need to set RSA exponent and RSA modulus by using corresponding methods as figure 4.12 shows. Both exponent and modulus were received from the KeePass plugin and stored in byte arrays "RSA_PUBLIC_KEY_EXPONENT" and "RSA_KEY_MODULUS".

```
rsaPublicKey.setExponent(RSA_PUBLIC_KEY_EXPONENT,
(short) 0 ,(short) RSA_PUBLIC_KEY_EXPONENT.length);
rsaPublicKey.setModulus(RSA_KEY_MODULUS, (short) 0,
(short)RSA_KEY_MODULUS.length);
```

Figure 4.12: Set parameters of Keybuilder object

For the encryption itself, two *Cipher* class methods are used. The first method is named *init*. It activates the encryption mode of the class and sets a given RSA key as an encryption key. After the encrypt mode is activated, a method *doFinal* encrypts any given array and returns a size of the encrypted data.

In this thesis, the RSA cipher is used for the database key protection during the transmission. In order to transmit the database key, the key is split into sixteen 128-bit fragments, where each fragment is encrypted before the transmission.

```
cipherENC.init(rsaPublicKey, Cipher.MODE_ENCRYPT);
short length = cipherENC.doFinal(BigDatabaseKey, (short)
(SizeOfAPDU*counter), (short)(SizeOfAPDU),outbuffer,(short)0);
```

Figure 4.13: RSA encryption - Cipher class

Chapter **5**

Testing

This chapter describes typical use-cases and how the plugin was tested. Testing of each operation is divided into positive and negative testing.

5.1 Unlocking a Database

5.1.1 Positive Testing

A database is opened in following steps:

1. Click on “File” → “Open” → “Open File”
2. Select “JavaCard Plugin” and insert Master Password
3. Choose a reader that will be used for the communication with Java Card
4. Insert PIN code

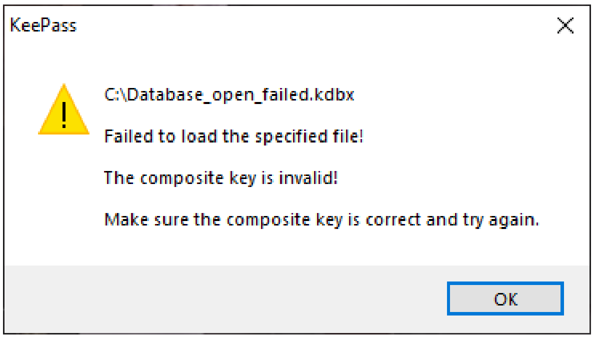
In case the the key stored in the card is correct, the database is opened.

5.1.2 Negative Testing

There are three possible errors may occur. First of all, incorrect PIN code is inserted by the user. This case is described in section 5.2.1. Then there can be an issue with a file that contains shared secret for SCP. If the shared secret is not found or is incorrect, an error message shown in figure 5.2 appears and the authentication process gets interrupted.

Last error that may occur is cause by incorrect key stored in the Java Card. In this case, error message 5.1 appears and the database stays locked.

All the APDU responses are checked by the plugin and in case of any irregularities, the plugin interrupts the authentication process with corresponding error message.



KeePass

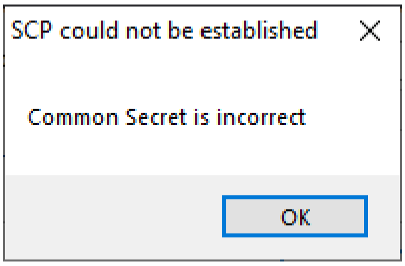
C:\Database_open_failed.kdbx

Failed to load the specified file!

The composite key is invalid!

Make sure the composite key is correct and try again.

Figure 5.1: Error message – incorrect key



SCP could not be established

Common Secret is incorrect

Figure 5.2: Error message – incorrect shared secret

5.2 Pin Authentication & Change

5.2.1 Description

For any communication, the card always require a pin for authenticate a user. The PIN can be changed in following steps:

1. Click on “Tools” → “Change Pin”
2. Choose a reader
3. Insert old PIN to authenticate yourself
4. Insert new PIN

In case the old PIN is correct, the PIN is changed.

5.2.2 Authentication Failed

The first thing that is checked is the length of the given PIN. If the length is incorrect, the plugin does not send it to Java Card and immediately interrupts the authentication process.

If the size is correct, the PIN is sent in the APDU command into the card. If the card returns APDU response with code “Authentication failed”, the plugin interrupts the authentication process along with the communication with Java Card. It should be pointed out that there are only three attempts for inserting the correct PIN code.

Conclusion

The main goals of this thesis were to develop a second-factor authentication plugin that allows unlocking a database using the Java Card technology, analyze possible security threats, and implement security measures that prevent an attacker from getting access to the KeePass database. To fulfill these goals, numerous steps needed to be followed.

Firstly, get familiar with the environment for writing Java Applets and analyze how to properly write them. Upload the Applets and install them to the Java Card. This analysis is covered in section 3.2. At the same time, it was necessary to get familiar with the KeePass plugin system. All important information about the system is described in section 3.1.1.

Secondly, create a naive implementation of both the Java Applet and KeePass authentication plugin. The implementation Recognizes its security threats and analyse what security measures are required. This analysis is covered in section 3.3.

Thirdly, implement security measures that provide protection against security threats from the previous analysis. That basically means to use Secure Channel Protocol described in section 3.4, encrypt the communication between both participants by using RSA cipher as described in section 4.1.2, and protect the Java Card with a PIN code.

Lastly, make the whole environment work and test if the security measures were implemented correctly. The testing is described in chapter 5.

Even though many technical obstacles occurred on the way, all the initial goals were fulfilled. Thanks to the result of this thesis, anyone can install the plugin and use it as an additional security layer that protects the password database.

In conclusion, the plugin and the Java Applet are ready to be used, although there is room for improvement and extensions. Currently, the plugin does not. provide an option to change the shared secret for SCP and an external tool has to be used. In future versions, the plugin should provide this option and allow the user to change this secret and inform the Java Card.

Bibliography

1. CRAFFORD, Lindsay. *7 Bad Password Habits to Break Now* [online]. 2021. Available also from: https://blog.lastpass.com/2021/01/7-bad-password-habits-to-break-now-2/.
2. DIGITAL SHADOWS PHOTON RESEARCH TEAM. *From exposure to takeover* [online]. 2020. Available also from: https://resources.digitalshadows.com/whitepapers-and-reports/from-exposure-to-takeover.
3. NATIONAL CYBER SECURITY CENTRE. Password administration for system owners [online]. 2018. Available also from: https://www.ncsc.gov.uk/collection/passwords/updating-your-approach.
4. NETWORKS, Barracuda. *Best Practices to Defend against evolving* [online]. 2020. Available also from: https://media.bitpipe.com/io_15x/io_150458/item_2091587/Barracuda_Spear-Phishing-Vol5-Dec2020.pdf.
5. NATIONAL CYBER SECURITY CENTRE. Password policy: Updating your approach. *ncsc.gov.uk* [online]. 2018. Available also from: https://www.ncsc.gov.uk/collection/passwords/updating-your-approach.
6. TECHOPEDIA. What is a Security Token? - Definition from Techopedia. *Techopedia.com* [online]. 2017. Available also from: https://www.techopedia.com/definition/16148/security-token.
7. KAUR, Rupandeep. Multi-Factor Authentication: Meaning, Advantages and Disadvantages. *Techthirsty* [online]. 2021. Available also from: https://www.techthirsty.com/multi-factor-authentication-meaning-advantages-and-disadvantages/.
8. LI, Stan Z; JAIN, Anil K. Encyclopedia of biometrics [online]. 2015. Available from DOI: https://doi.org/10.1007/978-1-4899-7488-4.

9. ROSENCRANCE, Linda; LOSHIN, Peter; COBB, Michael. *What is Two-Factor Authentication (2FA) and How Does It Work?* [Online]. TechTarget, 2021. Available also from: https://searchsecurity.techtarget.com/definition/two-factor-authentication.

10. MICROSOFT. One simple action you can take to prevent 99.9 percent of attacks on your accounts. *Microsoft Security Blog* [online]. 2021. Available also from: https://www.microsoft.com/security/blog/2019/08/20/one-simple-action-you-can-take-to-prevent-99-9-percent-of-account-attacks/.

11. *NIST SP 800-63 Digital Identity Guidelines-FAQ* [online]. 2021. Available also from: https://pages.nist.gov/800-63-FAQ/%5C#q-b12.

12. REICHL, Dominik. *Keepass password safe* [online]. 2021. Available also from: https://keepass.info/.

13. NGUYEN, Anthony. *Types of Smart Card* [online]. CardLogix Corporation, 2021. Available also from: http://www.smartcardbasics.com/smart-card-types.html.

14. KASMI, Mohammed Amine; MOSTAFA, Azizi; LANET, Jean Louis. Methodology to reverse engineer a scrambled Java card virtual machine using electromagnetic analysis. In: *2014 International Conference on Next Generation Networks and Services (NGNS)* [online]. 2014. Available from DOI: 10.1109/NGNS.2014.6990264.

15. GUO, Norman. *Smart Card Operation Using Freescale Microcontrollers* [online]. 2012. Available also from: https://www.nxp.com/docs/en/application-note/AN4453.pdf.

16. CHEN, I-Fong; PENG, Chia-Mei; YAN, Zhi-Da. *2019 IEEE International Conference on RFID Technology and Applications (RFID-TA)*. A simple NFC parameters measurement method based on ISO/IEC 14443 standard [online]. 2019. Available from DOI: 10.1109/RFID-TA.2019.8892221.

17. REICHL, Dominik. *Master key - keepass* [online]. 2021. Available also from: https://keepass.info/help/base/keys.html.

18. REICHL, Dominik. *Plugins - keepass* [online]. 2021. Available also from: https://keepass.info/plugins.html.

19. REICHL, Dominik. Plugin development (2.x) - keepass. *Get KeePass* [online]. 2021. Available also from: https://keepass.info/help/v2_dev/plg_index.html%5C#conventions.

20. ORACLE CORPORATION. *Writing a Java Card Applet* [online]. 2001. Available also from: https://www.oracle.com/java/technologies/java-card/writing-javacard-applet.html.

21. ORACLE CORPORATION. *Development kit user guide [online]* [online]. 2021. Available also from: https://docs.oracle.com/en/java/javacard/3.1/guide/extended-apdu-format.html%5C#GUID-0F452782-9402-43E9-B39C-337772C200DF.

22. GLOBALPLATFORM, Inc. *GlobalPlatform Technology Card Specification* [online]. 2018. Available also from: https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf.

23. RIVEST, R.L.; SHAMIR, A.; ADLEMAN, L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* [online]. 2018. Available also from: http://people.csail.mit.edu/rivest/Rsapaper.pdf.

24. ORACLE CORPORATION. RandomData Class. *RandomData (Java Card API, Classic edition)* [online]. 2015. Available also from: https://docs.oracle.com/javacard/3.0.5/api/javacard/security/RandomData.html.

25. SEPULO [https://github.com/sepulo/globalplatform.net?fbclid=IwAR07PvQPH5t4KaPnYX6Mw3okUJmDdhijHrvdBjb5oGOpmLgM_UE_R32eWa8]. 2019.

Appendix **A**

Acronyms

2FA Two-factor Authentication

3DES Triple Data Encryption Standard

APDU Application protocol data unit

C_MAC Command Message Authentication Code

DES Data Encryption Standard

ICV Integrity Check Value

JCRE Java Card Runtime Environment

MAC Message Authentication Code

MFA Multi-factor Authentication

NFC Near Field Communication

PIN Personal Identification Number

RFID Radio Frequency Identification System

RSA Rivest-Shamir-Adleman

R_MAC Response Message Authentication Code

SCP Secure Channel Protocol

SFA Single-factor Authentication

Appendix B

Installation instructions

B.1 KeePass Authentication Plugin

To install the authentication Plugin, all you need to do is move a folder named the “JavaCard plugin” and move it to “KeePass Password Safe 2/Plugins”. No additional action is required.

B.2 Java Applet

To install Java Applet to Java Card, I recommend using “GPshell” available at: https://sourceforge.net/p/globalplatform/wiki/GPShell/.

Firstly, you use “gp -install Java_Applet
bin
FinalTest
javacard
FinalTest.cap”, where the Java_Applet folder is one of the attachments of this thesis.

After the applet is successfully installed, it is necessary to properly store a shared secret for Secure Channel to your “Documents” folder. The default shared secret is three keys, all set to “404142434445464748494A4B4C4D4E4F” (“@ABCDEFGHIJKLMNO”).
These keys need to be stored in the following steps:

1. Create a file where each line is one key (MAC key, ENC key, KEK key)

2. Name the file “Cardkey.txt”

3. Encrypt the file content by using the EncryptSecret program

4. Save the file named “CardkeyEncrypted.txt” to your “Documents” folder

After creating the file with the shared secret, the applet is ready for a test run. The default pin is “1234” and can be changed as described in the implementation chapter of this thesis.

Appendix C

Contents of the enclosed CD

```
│ readme.txt ................................... Contents description
├── JavaCardPlugin ....................... KeePass Authentication Plugin
│   ├── Documentation .......................... Doxygen Documentation
│   ├── JavaCardPlugin ............................... Project Directory
│   └── JavaCardPlugin.sln ......................... VS Code Project File
├── Authentication_Applet ............. Java Card Authentication Applet
│   ├── src ................................. Source code of the Applet
│   ├── bin ................................ Executable files – CAP File
│   └── Authentication_Applet.jcsproj ....................... Project File
├── EncryptSecret .............. Program for encrypting the Shared Secret
│   ├── EncryptSecret ...................................... Source code
│   └── EncryptSecret.sln ........................ VS Code Project File
└── Thesis.pdf .............................. PDF version of the thesis
```