Storage of sensitive data in a Java enabled cell phone

Tommy Egeberg

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2006

Institutt for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

Today's people often need to remember many different passwords to get access to a variety of services. To make the password management task easier, single sign-on solutions (SSO) are implemented. A new idea for SSO implementation is to store the passwords on a Java enabled cellular phone [1].

The idea is that a Java application on a cellular phone communicates with a PC through a special USB Bluetooth device. A hacker will in this situation need both the Bluetooth device and access to the cell phone application to be able to obtain the passwords.

This thesis will present how suited these mobile devices are to store sensitive information like passwords from a SSO solution. It will identify vulnerabilities and threats associated with this way of doing single sign-on. A prototype formed as a digital safe will be implemented. An analysis of this prototype will then be performed, to find it's strengths and weaknesses.

The aim of this investigation is to find out if this new SSO idea will be robust enough and therefore can be a new trend in SSO, as well as identify the ways a Java enabled cellular phone can keep confidential information secure.

Keywords: Single Sign-On, Java enabled cell phone, confidential information.

Sammendrag (Abstract in Norwegian)

Dagens personer må ofte gå rundt å huske mange forskjellige passord, som blir brukt til å aksessere mange ulike tjenester. Single sign-on (SSO) løsninger blir derfor utviklet for å gjøre denne passord håndteringen enklere. En ny ide innen SSO, er å lagre alle passordene på en mobiltelefon med støtte for Java [1].

Ideen går ut på at mobiltelefonen kommuniserer med en PC via en spesiell Bluetooth enhet. I en slik situasjon må en hacker derfor ha tilgang til denne Bluetooth enheten og ha aksess til telefon applikasjonen for å få tak i passordene.

Denne oppgaven vil derfor prøve å presentere hvor egnet en slik Java telefon er til å håndtere og lagre sensitiv informasjon som disse SSO passordene. Den vil identifisere sårbarheter og trusler rundt denne metoden å hådtere passord. En prototype utformet som en digital safe vil be implementert. Denne prototypen vil deretter bli nærmere analysert, for å finne dens sterke og svake sider.

Målet med denne forskningen er å finne ut om denne SSO ideen vil være robust nok til å kunne bli en ny trend innen SSO. Målet vil også være å identifisere de ulike måtene Java mobiltelefoner kan holde konfidensiell informasjon sikker.

Nøkkelord : Single Sign-On, mobiltelefoner med Java støtte, konfidensiell informasjon.

Preface

The master thesis you are about to read is the final work after two years on the master study in information security at Gjøvik university college. I have acquired a lot of interesting knowledge regarding computer security through these years, and this knowledge has been the grounding under the work on this thesis.

Mobile phones has for me always been an interesting topic. These small devices have become incredible popular and are dispensable for many people. A mobile SSO solution developed by Mats Byfuglien makes use of a cell phone with Java to store the login credentials. A research around mobile phones capabilities to handle sensitive information with help of a Java application (MIDlet), was therefore a topic that caught my interest.

Several people have been helping me to complete the thesis within the time constraint. I will first of all like to thank my supervisor Einar Snekkenes, who has given me a lot of ideas and help in darkest moments. I will also like to thank Mats Byfuglien who has read my thesis and given me feedback. Svein Willassen has also given me some ideas and got me into contact with Stig Stavik and Jarle Eide who sent me their papers regarding mobile forensic, and I will therefore like to thank all three of them. Last but not least will I give my gratitude to my girlfriend who has let me work on this project into late at night and given me a morally support.

Tommy Egeberg, 21st June 2006

Contents

List of Figures

List of Tables

1 Introduction

In today's society we have passwords to access many different services. Many different passwords make it difficult to remember all of them, especially the ones you do not use regularly.

A single sign-on solution (SSO) will make it much easier, because you only need to remember one master password while the system keeps track of rest of them [2, 3].

SSO is indeed a security concern, because if a hacker manages to crack the master password, he will gain access to all your passwords. This thesis will therefore investigate if a Java application on a cellular phone is suited to store these passwords in a secure manner.

1.1 Topic

There exist different SSO solutions today, but many of them are far from good. They are often to expensive and lacks mobility. A new approach is to move this SSO solution into a cellular phone [1], to make it harder for a hacker to obtain the stored passwords.

The thesis will therefor focus on how suited a Java MIDlet on a cellular phone is to store sensitive information like passwords. Threats and vulnerabilities will be identified, to find out how robust a cell phone can be against hacker attacks. It will also identify if these phones are able to keep the passwords secure even if they are lost or stolen. From a hacker's point of view: "Will it be hard to obtain sensitive information located on a cell phone"?

Keywords: Cell phones, Java MIDlets, Single Sign-On, sensitive information.

1.2 Problem Description

Single sign-on solutions (SSO) are made because of the growing number of passwords a single user needs to remember. Passwords are therefore often written down, and the same password is used frequently in different sites [4, 5].

A SSO solution tries to handle this password management problem by storing all the user's passwords in one secure location. The user will then need to remember only one master password to access the other ones.

> One password that acts as the "key to the kingdom". [3].

SSO will therefore reduce the administrative costs because of forgotten passwords and the user's day are made easier [3, 6, 7].

Gjøvik University College (GUC) and NISlab have developed a Bluetooth device, which can interchange information with a computer and a cellular phone. A Java MIDlet installed on a cell phone is further used to keep track of the user's passwords. This may be a safer situation, since a personal computer most often is connected to the Internet and makes it vulnerable to attacks. The cellular phone is on the other hand something a user almost always carries and it is easy to detect if it is lost or stolen.

The background of this thesis is the fact that today's SSO solutions seem to be far from perfect, see appendix A. They are often to expensive or they lack mobility. A security analysis will therefor be completed through this thesis, to identify the pros and cons by moving the SSO into a Java MIDlet on a cellular phone.

The problem treated in this thesis may be stated as follows:
Will a Java MIDlet on a cellular phone be a secure location to store sensitive information such as passwords from a SSO solution?

1.3 Research Questions

In order to find out if a Java enabled cellular phone is suited to store sensitive information like a password store from a SSO solution, the following questions needs to be answered.

1. What is already known about security in Java enabled cell phones?
2. Will information stored on a cellular phone be easy to extract?
3. How can we secure the stored sensitive information even if the cellular phone is lost or stolen?
4. What kind of threats is the cell phone be vulnerable to?
5. What kind of countermeasures can be used to reduce or eliminate the threats?

The answers from all these questions will together give us an answer to the main problem.

1.4 Delimitations

This thesis will not try to implement a working SSO solution, but investigate how suited a Java MIDlet is to store the passwords from a SSO. A prototype will be implemented and an analysis will later reveal strengths and weaknesses. This prototype will be like a digital safe, where you can store your passwords and prevent unauthorized access.

The main task is to investigate a Java MIDlet's ability to secure sensitive information and how we can make it as hard as possible for a hacker to gain access to it.

Even if this thesis mainly focuses on passwords from a SSO solution, will much of the discussions also suite other kind of sensitive information stored in a cell phone.

1.5 Importance of the Study

A single sign-on solution makes the password management task easier. A user needs only to remember one master password, instead of tens or twenties of different passwords. The stored passwords can also be auto generated, if a user only needs to remember one such master password. Strong, long and hard to remember passwords can therefore be used, without the user being afraid to forget them.

It is indeed a security concern to store all these passwords in a single location. If the master password is compromised, the hacker has access to all of the stored passwords [7].

If we can implement a SSO solution which can enhance the degree of robustness, the use of many different passwords will be made more secure, and still have an easy way to use them. The SSO idea which stores the login credentials in a Java MIDlet on a cell phone [1], may become a new era of SSO. It will therefore be important to investigate

these phones ability to securely store sensitive information.

1.6 Summary of Claimed Contributions

This thesis is going to investigate the use of Java enabled cell phones and storage of sensitive information. A prototype formed as a digital safe will be developed and later analyzed with the aim of finding strengths and weaknesses associated with cell phones and sensitive information.

This prototype will hopefully also give us an indication of how authentication can be handled in the best suited way, and passfaces will therefore be implemented in the prototype.

The prototype will also be used in an attack. In other words, we will try to extract the sensitive information stored in the digital safe. If this is easily manageable will storage of sensitive information in a cell phone not be a very good solution after all.

1.7 Choice of Methods

This section will describe shortly the methods used to answer the research questions stated in section 1.3.

The study will mainly be a literature study, with focus on related work regarding security on mobile devices. It will be important to have a broad understanding of how the security is taken account of under the J2ME platform. When the development phase of the prototype starts, it will also be important to understand what kind of security capabilities the APIs under the J2ME platform offer. Books, articles, journals and web sites will be used to obtain the required information.

Library catalog, index and abstract searches will be further used. The web will be searched with use of search engines like www.google.com and others [1]. There are also much information available through the search engines that are more related to science like www.scholar.google.com and Scirus [2]. Another approach is to make us of the online databases like Springerlink [3], Science Direct [4], IEEEXplore [5] and ACM Portal [6].

The references and citations found in the literature will be used further to find additional material. The approach for this literature study is taken from [8, p.65-71].

This method will give answer to research question 1.

There will also be implemented a test MIDlet for storage of sensitive information (a digital safe). This prototype will be further analyzed, with the aim of finding strengths, weaknesses and possible changes that should be implemented.

The prototype will also be used in an attempt to hack the phone, and extract the sensitive information. This kind of SSO will not be a very good solution after all, if the stored information is easy to extract. How robust it is against different types of attacks and accessibility of cryptographic algorithms will therefore be parameters that must be investigated further.

[1] `www.sesam.no`, `www.yahoo.com`, `www.kvasir.no`
[2] `http://www.scirus.com` Science-specific search engine
[3] `http://www.springerlink.com` Online engineering database
[4] `http://www.sciencedirect.com` Online technology database
[5] `http://www.ieee.org` Online article database
[6] `http://portal.acm.org/portal.cfm` Full text collection of every article published by ACM.

This vulnerability analysis can give us a firm understanding of how suited a Java enabled cell phone is to store sensitive information, and will answer research question 2.

To answer research question 3, we are talking about authentication, confidentiality. Solution to this will be provided by use of cryptography [9].

Passwords are often used to secure sensitive information [10]. An investigation around the possibilities to use passfaces [11] instead of regularly passwords will therefore be carried out. Other password approaches will also be included in the analysis. The results will be compared, to find the strongest and best solution.

The different solutions' robustness will mainly be tested with brute-force attacks. It will therefore be beyond the scope of this thesis to investigate the strengths and weaknesses of the different cryptographic algorithms.

I will make use of multiple data flow diagrams to answer research question 4 and 5. A cell phone will move from manufacture to store, and later on may a user buys it. The phone may also be broken, and sent to a repair man. Flow diagram will try to find all these data flows. A hacker will namely be able to place himself at many different locations. With this in mind, threats and vulnerabilities which at first seemed unimportant may be discovered.

Threats and vulnerabilities which are found, will then be used to come up with solutions which may decreases or eliminate the particular threat.

1.8 Definition of Terms

Through the thesis' introduction a variety of technical words have been used, and following is a short explanation to some of them.

API: Application Program Interface. This is a set of routines, protocols and tools for building software applications.

Authentication: A way to ensure users are who they say they are [12].

Bluetooth: Bluetooth is a standard that makes it possible for wireless devices to communicate [13].

Confidentiality: Ensuring that information is accessible only to those authorized to have access [14].

Java: Java is a programming language that is supported by most of today's cellular phones. Java 2 Platform, Micro Edition (J2ME) is a subclass of the ordinary Java language, and is developed to support devices with small screens and limited amount of memory and processor speed.

MIDlet: A MIDlet is an application which is developed to run on cellular phones and that conforms the MIDP standard.

MIDP: Mobile Information Device Protocol. This is a set of J2ME API's which define how software interfaces with cellular phones.

Sensitive information: Information that is important to keep safe. It can be fatally if the information is made available to someone that shall not have this information. In this context we focus on passwords.

Single sign-on: A specialized form of software authentication that enables a user to authenticate once and gain access to the resources of multiple software systems [15].

1.9 Outline of the Report

The thesis will first take a look at related work in chapter 2. In this chapter will we take a closer look at the J2ME platform, and present the different security features that are implemented in the CLDC configuration and the MIDP profile. An overview of different data extraction techniques will also be introduced.

The developed prototype will then be explained in chapter 3, where we take a closer look at it's security components and functionality.

In chapter 4, an analysis of the prototype and other security considerations will be performed. In this chapter will a Java MIDlet's threats be discussed together with an overview of different authentications mechanisms. Different attacks will also be presented in this chapter.

A discussion around the result from the analysis will be presented in chapter 5. Further work and conclusion will then be presented in chapter 6 and 7 respectively.

2 Related Work

This thesis tries to find out how well a Java enabled cell phone is to store sensitive information like passwords from a single sign-on solution. A digital safe has been developed and later analyzed, with the aim of finding strengths and weaknesses.

Before the analysis phase we need to look into the theory, to get some fundamental knowledge about the J2ME platform and basic information security.

2.1 Security In J2ME

Java is a platform independent and object oriented programming language. The Java 2 Micro Edition platform (J2ME), is further a Java edition specially designed for small resource constrained devices like cell phones. Almost all of today's cell phones support this technology, and in 2005 the deployment of these phones reached more than 450 million around the world [16].

Before we can start to analyse the developed prototype, we need to get a firm understanding of how security is handled under the J2ME platform. Through the following sections you will get a broader understanding of the different elements of which the platform is divided into and how they work together.

The next section will first describe the overall design principles of the J2ME platform. This is important to understand when we are discussing the different security mechanisms later in the chapter.

2.1.1 The J2ME Platform

Java 2 Micro Edition (J2ME) is a Java platform designed to work on small resource constrained devices. With its small footprints it is therefore a good solution for developing standalone application on devices like mobile phones.

A complete J2ME runtime environment contains a configuration and a profile, together with other optional packages. There are two J2ME configurations, either CDC (Connected Device Configuration) or CLDC (Connected Limited Device Configuration) [17, 18, 19]. The configuration specifies the core libraries and virtual machine features that must be present in a J2ME environment.

In order to provide a complete Java runtime environment the configuration must be combined with a profile. A profile is a a set of higher-level APIs that further define the application life-cycle model, the user interface, and access to device-specific properties [18]. The MIDP (Mobile Information Device Profile)[20, 21] combined with CLDC [22, 23] make up the widely used runtime environment for mobile phones.

The structure of J2ME and its relations to the other parts of the Java family can be seen in figure 1.

Another important concept is the way MIDlets can be joined together in a suite. A MIDlet suite is a collection of different MIDlets packaged together in a single jar file. This concept is important to keep in mind when we talk about the Record Management System in section 2.1.5.

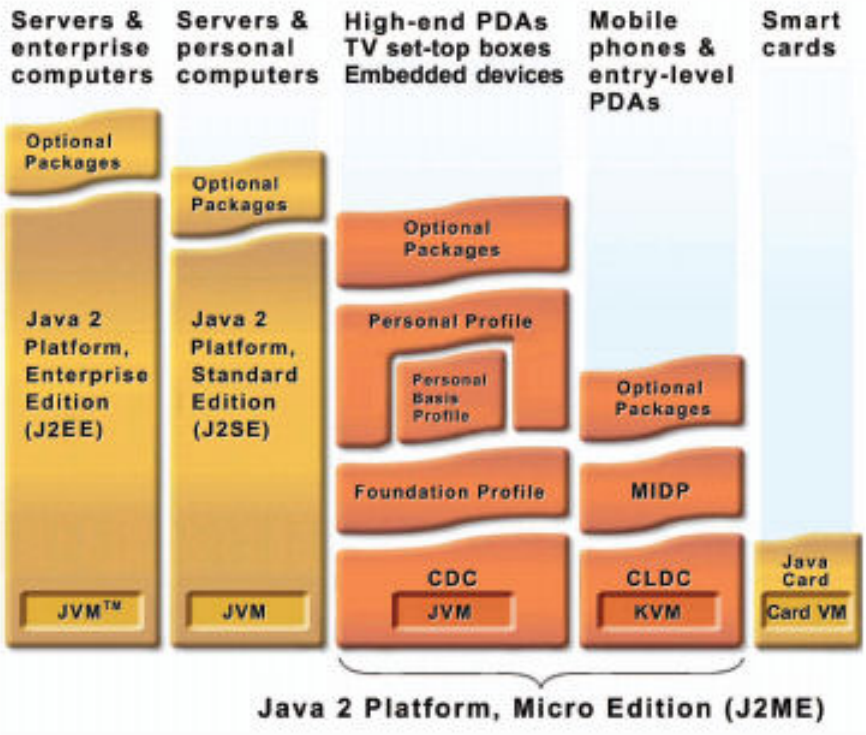


Figure 1: The Java family. Image from [17]

Connected Limited Device Configuration - CLDC

CLDC is a configuration made up of a virtual machine (KVM) and a minimum set of class libraries. CLDC 1.1 is the latest version, and the only requirement to use this configuration is at least 160 KB of nonvolatile ROM and a minimum of 32 KB of volatile RAM (at least 128 KB nonvolatile memory for CLDC 1.0) [9, 22, 23].

When talking about CLDC and security we have three security levels, namely low-level virtual machine security, application security and end-to-end security. The low-level security ensures that the Java bytecode doesn't contain illegal instructions and references to illegal memory locations. This work is done by the class file verifier, see section 2.1.2.

Since the low-level security check only guarantees that the program is a valid Java application, we need the application security level. This level ensures that the application only accesses the libraries and system resources which the device and Java environment allow it to access [23]. Application security is ensured by running the application in a closed "sandbox" environment. The application will in this sandbox only be able to access those libraries and resources defined in the configuration, profile and the device specific classes. Applications can not escape from this sandbox. Malicious applications will therefore be prohibited to access system resources. The sandbox requires that class files are properly verified, ensures that the standard class loading mechanism isn't bypassed or modified and that the native functions accessible by the virtual machine are closed in such a way that the application only can access classes defined in the CLDC, MIDP or manufacture specific classes.

A CLDC implementation shall also ensure that overriding or modification of the pro-

tected system packages are prohibited (javax.microedition.*, java.*). This is important because of the CLDC's ability to dynamically download Java applications to the virtual machine. This can be done by always searching for the system classes first. This dynamically downloading ability is implementation dependent with one restriction; it can only load classes from its own Java archive (JAR). This ensures that applications are disabled to interfere with each other or steal data from each other. Manipulation of the class file look up order should also be prohibited [24, 9, 25, 26, 22, 23].

The end-to-end security level is taking care of secure communication between the J2ME device and servers, with for example encryption. This security level is out of scope of the CLDC specification, and is implementation dependent.

Mobile Information Device Profile - MIDP

MIDP offers the core functionality like user interface, network connectivity, data storage and application management. There are also some requirements that must be fulfilled to make use of this profile. The device must have a display with a minimum of 96x54 pixels with a display depth of 1 bit. There must be a key board or touch screen, at least 264 KB of non-volatile memory and 128 KB of volatile memory, networking capabilities (ex. wireless) and the ability to play tones [21, 19, 24].

Another important point; the device must give the user a visual indication of any network usage generated by applications [21]. This is often done by presenting the user with a small earth icon on the top of the screen. With this in mind it is not possible for an application to send data to for example a server without letting the user aware of the network usage.

MIDP is currently available in version 2.0 [21].

Optional Packages

The optional packages are APIs that can increase an application's functionality even further. The Wireless Messaging API (WMA) is an example of an optional package. With this API you can do even more than you could with only the CLDC configuration and MIDP profile, for example SMS (Short Message Service) sending and WMA push capabilities [27].

There are plenty of optional packages available, but it up to the device manufacture to decide which packages to implement. The Sony Ericsson K700i does for example support the optional packages WMA (JSR 120), MMAPI (JSR 135) and Java 3D (JSR 184) [28]

Another important package, which is not supported by many of the today's cell phones, is the Java APIs for Bluetooth Wireless Technology (JABWT, JSR 82) [29]. This API is used in Mats Byfuglien's mobile SSO solution [1].

The Virtual Machine

A Java platform needs a virtual machine to interpret the Java bytecode. Since the standard Java Virtual Machine (JVM) is to heavy for small resource constrained devices, Sun Microsystems Inc developed the K Virtual Machine (KVM). This is a highly portable Java virtual machine special designed for small devices. The K stands for "kilo", because the memory budget is measured in kilobytes and not megabytes like the standard JVM. The KVM is closely related to the CLDC and is the foundation of the J2ME platform.

The KVM's task is to convert the application's bytecode into machine-level code for the appropriate hardware and OS being used. It shall also manage the systems memory, provide security (reject invalid class files) and manage application threads.

The KVM must have a class loader that cannot be reconfigured, overridden or replaced. The KVM is further not required to support finalizing of objects, and exception handling is limited.

The KVM is designed for resource constrained devices. With it's VM size of about 80KB, reduced memory utilization and the ability to run on devices with a processor clocked at 25 MHz, is it indeed perfect for cell phones [19, 30, 24].

The CLDC HotSpot Implementation is another Java virtual machine [16]. This is Sun's high-performance VM for resource-constrained devices, and was introduced in mid 2002. The CLDC Hotspot implementation is an answer to the new and more powerful handsets, and its goals are:

- Faster performance
- A more robust platform
- Faster time to market

The CLDC HotSpot Implementation delivers nearly an order of magnitude better performance than the KVM while running in a similarly small memory footprint required by resource-constrained mobile phones and personal organizers [16].

2.1.2 Bytecode Verification

Since the Java virtual machine must be able to reject invalid class files, it must support class file verification. The standard verification procedure done in J2SE are way to heavy for devices like cell phones. The verification process in these constrained devices (J2ME platform) is therefore divided into two phases.

- Preverification
- Runtime verification

The preverification phase is done at compilation time on the developer's workstation. In this phase certain bytecodes are removed and a stack map which shall speed up the runtime verification is added. The runtime verification will then make use of the stack map to perform an efficient verification of the class files [22, 23, 19, 9].

The execution of the application is prohibited if the class files don't pass the runtime verification process. An example of the verification process can be seen in figure 2.

2.1.3 Security Architecture

The J2ME environment doesn't only have security features implemented in the virtual machine and configuration. Security features are also implemented in the MIDP profile. In this section you will get a short overview of the security mechanisms that exist in the MIDP profile.

Protection Domains

A protection domain is one of the MIDP's security mechanisms, and it defines the permissions that can be granted to the MIDlet. Permissions consists of two types, namely allowed permissions and user permissions. These permissions are used to protect sensi-

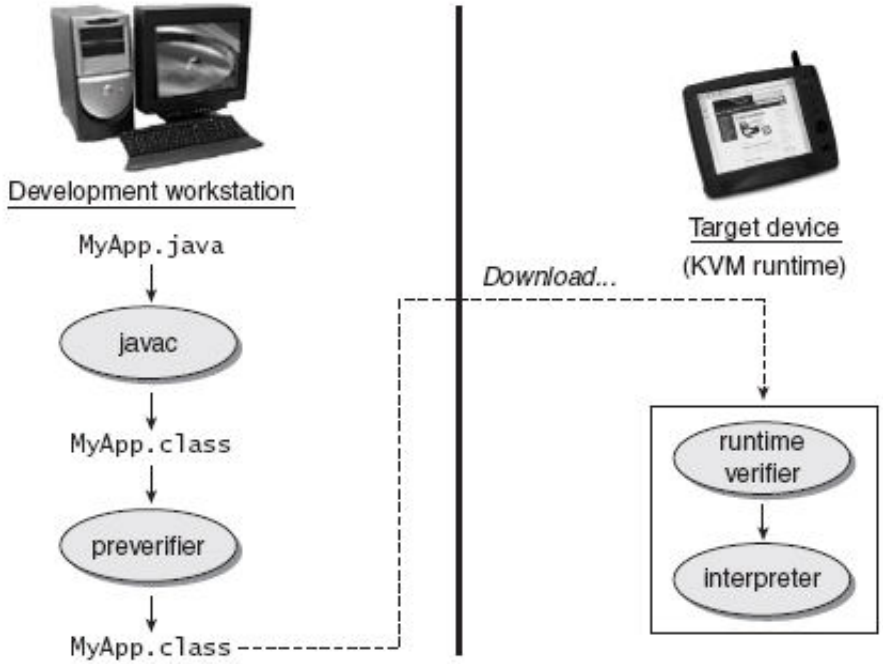


Figure 2: The two phase verification process of class files. Image from [23]

tive operations. In the MIDP 2.0 specification all types of network connection is regarded as sensitive operations, since it may involve monetary costs for the user [31]. Other optional packages can also have their own sensitive operation, which require the same type of permission to be executed.

The user permissions are permissions for which the user is consulted, and there exists three modes for how this can be granted; blanket, session and one-shot. The one-shot (ask every time) mode will always ask the user when the sensitive operation is to be invoked. The session mode (ask first time) will on the other hand only consult the user for permission the first time the operation is to be invoked, and the user decision will last until the MIDlet is closed. Blanket (always allowed) is the last mode, for which the permission is allowed as long as the MIDlet is installed on the device. It is the protection domain which determines the available mode.

Allowed permissions are sensitive operation for which the MIDlet request. If for example a MIDlet wants to make a http connection, it needs to have permission to make this connection. This is done by defining a permission request for http connections in the MIDlets JAD (Java Archive Description) file, like the string:

MIDlet-Permissions: javax.microedition.io.Connector.http

If the protection domain, in which the MIDlet is assigned, allows it;

"allow: javax.microedition.io.Connector.http",

the MIDlet has permission to do an http connection whiteout any user involvement.

Any other type of network connection is as mentioned also regarded as a sensitive operation. All these sensitive operations will therefore need permission to be executed.

Other optional packages can further have their own sensitive operation, which require the same explicit permission request in the JAD file.

If the device can verify the authenticity and integrity of the MIDlet suite, and assign it to a protection domain, the MIDlet suite is said to be trusted [31]. Trusted MIDlets will have their permission requests granted if the domain allows it, and no user permission will be needed. Trusted and untrusted MIDlets will therefore have different interaction modes, depending on the rules in the protection domain.

Manufacturers, carriers, and other MIDP implementors are free to create whatever domains suit their purposes [32].

MIDlet Signing

When talking about protection domains is it also important to mention MIDlet signing, which is done with the use of cryptographic signatures and certificates. Unsigned MIDlets is by default placed in the untrusted domain. A signed and authenticated MIDlet can on the other hand belong to the Manufacture domain, operator domain, trusted 3rd party domain or whatever domain the manufactures, carriers or MIDP implementors create.

To be able to sign your MIDlet, you will need a certificate conformed to the X.509 specification (PKI). The JAR file will then be signed with use of the private key. The generated signature and the utilized certificate will then be added to the MIDlet suite's JAD file.

A signed MIDlet will also assure a user of that the MIDlet hasn't been tampered with and that it comes from an identifiable source. Signing is in close relationship with protection domains, since a signed MIDlet will be bound to a protection domain dependent on the certificate being used, see figure 3.

A more detailed description of the signing procedure can be read in [34]

2.1.4 Cryptography

Security is something that always should be taken care of, especially if we are dealing with sensitive information. Encryption can deal with confidentiality requirements, which is the main goal of this thesis' prototype.

In the standard Java environment (J2SE), you have the ability to use the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE) to secure your applications and files. These packages support a wide range of cryptographic services like digital signatures, message digests, symmetric and asymmetric ciphers, authentication codes, key generators and key factories. And with a support for a wide range of algorithms like RSA, AES, SHA and DES you have a strong fundamental ground for developing secure applications [35].

The problem is that this security package is not included in the CLDC/MIDP environment [9, p. 155]. You will therefore need to develop your own cryptographic routines, or make use of the crypto API's from vendors like Bouncy Castle [36]. With these optional packages you will have the ability to implement applications with the security services of choice.

As we know from cryptography; longer encryption keys enhance the security but require more computing time/power. This is indeed a problem of todays mobile phones, which have a lack of processing power, limited memory and low bandwidth. Elliptic curve (EC) encryption may therefore be the choice, since it is more computationally friendly [9, 37]. The cryptographic library from Bouncy Castle has also got the necessary func-

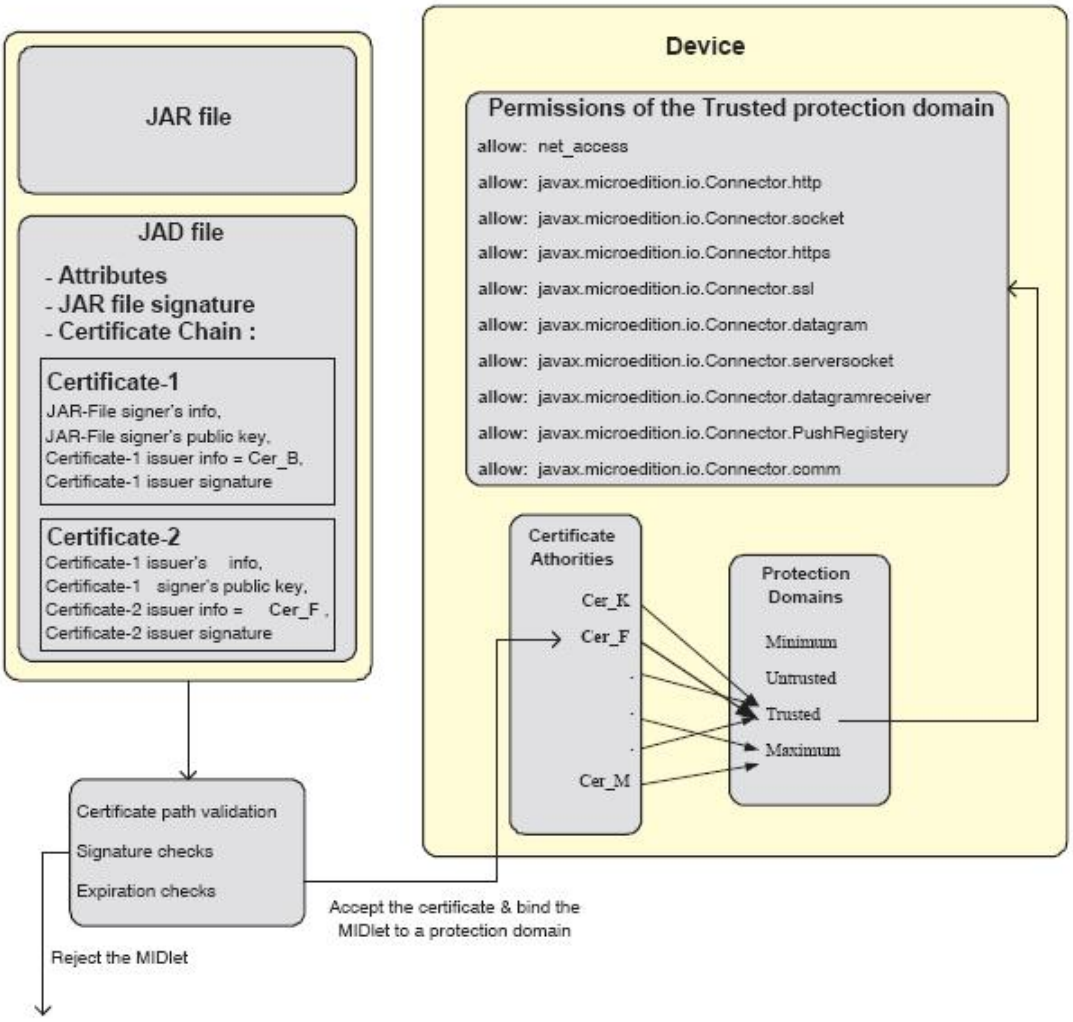


Figure 3: MIDlet signing will bind a MIDlet to a protection domain. Image from [33]

tionality to perform EC encryption.

The Security and Trust Service API (JSR 177) will also be helpful in developing applications which need information security such as confidentiality, integrity and non-repudiation.

The Security And Trust Services API - SATSA

The Security and Trust Services API (SATSA) included with JSR 177, defines four optional packages; SATSA-Application Protocol Data Unit (APDU), SATSA-Java Card RMI (JCRMI), SATSA-Public Key Infrastructure (PKI) and SATSA-CRYPTO [38].

These packages does further define a set of APIs that allows J2ME applications to communicate with and access functionality, secure storage and cryptographic operations provided by security elements such as smart cards and Wireless Identification Modules (WIM) [39].

The SATSA will thus make the integration between smart cards and java applications

tighter, and the following capabilities will be provided to the J2ME platform:

- Smart Card Communication with APDU and JCRMI
- Digital Signature Service and Basic User Credential Management with SATSA-PKI
- General Purpose Cryptography Library with SATSA-CRYPTO

The SATSA-CRYPTO package defines a subset of the J2SE cryptography API [40].

Only a limited amount of Java enabled cell phones are supporting SATSA as this writing. But since the J2ME platform containing the CLDC and MIDP doesn't provide any encryption and signing capabilities, will SATSA most likely be supported in the next generation mobile phones. You will however still have the possibility to make use of a crypto package, like the one from Bouncy Castle [36].

Bouncy Castle

The Bouncy Castle (BC) crypto package is a Java implementation of cryptographic algorithms. BC's light weight API [36] is suitable for use on the J2ME platform, and has a structure similar to the JCE (Java Cryptography Extension).

It has been taken advantages of this API's hash functions and encryption algorithms in the thesis prototype. There are also methods for doing both symmetric and asymmetric encryption/decryption, message signing, key generation, modes and elliptic curve cryptography. This light weight API is also free of charge for commercial and non-commercial use.

A performance overview of different block ciphers implemented in Bouncy Castle can be seen in table 1. This table gives also the same overview of some hash functions. The data has been taken from [41], and has been tested on a MIDP 1.0 device processing a plain text message of 224-bytes. The value is further the average of five runs, where each run does the encryption 100 times.

Algorithm	Average Time (ms)
DES	787
3DES	1,997
AES-Fast	601
AES-Middle	1,463
AES-Light	1,432
Rijandael	3,858
Blowfish	501
IDEA	881
RC2	1,107
RC5	542
MD5	422
SHA1	667

Table 1: Run-time performance of different block ciphers and hash functions implemented in Bouncy Castle, taken from [41].

IAIK JCE Micro Edition

The institute for applied information processing and communications (IAIK) has also developed a light weight cryptography library suitable for cell phones (JCE-ME) [42]. This is a commercial product, but you may download an evaluation version, or have a version for educational purpose.

This package is similar to the package from BC, with functionalities like hash functions, message authentication codes, key and certificate management, symmetric, asymmetric, stream and block encryption. This package is also implemented in Java, and has got a small size.

Other Cryptographic Libraries

Phaos Micro Foundation is another cryptography library designed for small resource constrained devices [43]. It is like BC's and IAKI's libraries implemented in pure Java, and has also got a very similar functionality.

NTRU Neo for Java [44] is another package. Like the others, it supports encryption, signing, hashing etc.

2.1.5 Record Management System (RMS)

If a Java MIDlet has to store information on a cell phone for later use, the J2ME platform gives the only ability to make use of the Record Management System (RMS). Since we are dealing with sensitive information it is important to know how this RMS is working.

The RMS stores information almost like in a database, where the record store can be associated with a table. The MIDlet is free to create as many record stores it needs, and the name is case sensitive. No MIDlet can create two record stores with the same name. The record store is further time stamped with a value describing the last modification. Each modification will also update the record store's version number. This is a value incremented for each operation that modifies the content.

A record store can only be accessed from MIDlets from the same *suite*, unless it is explicit declared in the source that other MIDlets can access it.

Record stores are created in platform-dependent locations, which are not directly exposed to the MIDlets [45]. A record in the record store is further an array of bytes with a unique record id. This record id works as the primary key, and is an incremental value that is never reused. The structure of the records in a record store is given in table 2.

Record ID	Data
1	Data 1 ....
2	Data 2 ....
3	Data 3 ....
5	Data 5 ....
6	Data 6 ....
n	Data n ....

Table 2: The structure of the records in a record store.

2.1.6 Push Registry

The prototype developed will also make use of the push registry functionality to automatically start the Password Store MIDlet when it receives a SMS message on a given port number. With this mechanism, a remotely deletion of the sensitive information may

be implemented as a security mechanism for situations where the cell phone is lost or stolen.

When discussing the push registry we need to get familiar with the application management system (AMS). The AMS is responsible for every MIDlet's life-cycle on the device. In other words; handling installation, activation, execution and removal of MIDlets [46].

There are two ways to activate a MIDlet with push. Timer based alarms or as in our example by an inbound network connection (SMS, TCP socket or UDP datagram).

Push operations are like other network connections considered as sensitive operations. These sensitive operations must as explained in section 2.1.3, request permission before they can be used. To request permission to use push, the following string must be added to the MIDlet's permission property in the JAD file:

```
javax.microedition.io.PushRegistry
```

To make the MIDlet push enabled, the MIDlet needs to register with the push registry. This can be done static or dynamic. Dynamic registration is done at runtime with help of the PushRegistry API while a static registration is done by defining a MIDlet-push attribute in the MIDlet's JAD file. The AMS will then register the requested address. If a static registration is trying to register an address already bound, the installation will fail. The AMS will also automatically unregister a MIDlet's push registrations when the MIDlet suite is uninstalled.

An example of a static registration can be seen in the string below:

```
MIDlet-Push-1: sms://:3536,PasswordStore2,*
```

This must be added to the MIDlet's JAD file, and it informs the AMS that the PasswordStore MIDlet shall be started when a SMS at port 3536 is received. The * means that the SMS can be sent from everywhere. See section 3.3 for information about how to send SMSs to a given port number.

As can be seen in figure 4, the AMS keeps track of push registrations and automatically starts an application if a registered event occurs.

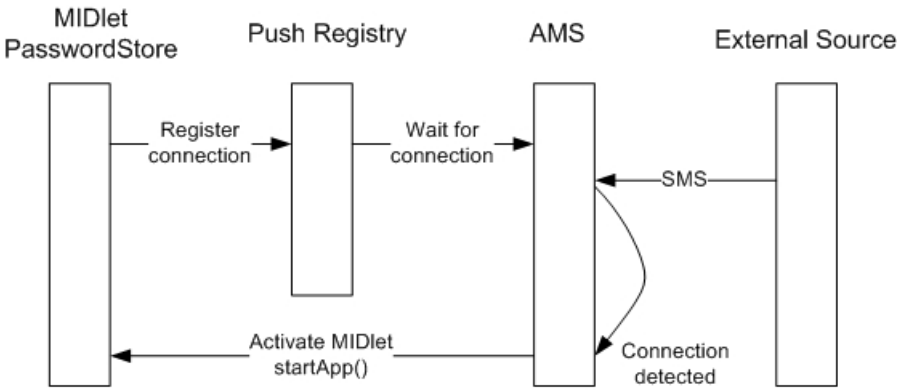


Figure 4: MIDlet activation with push registration.

2.1.7 Summary

A complete J2ME runtime environment contains a configuration and a profile, together with other optional packages. The configuration specifies the core libraries and virtual machine features that must be present, while the profile is a set of higher-level APIs that further define the application life-cycle model, the user interface, and access to device-specific properties.

The CLDC operates with three security levels, namely low-level virtual machine security, application security and end-to-end security. The low-level security ensures that the Java bytecodes doesn't contain illegal instructions and references to illegal memory locations. The application security is ensured by running the application in a closed "sandbox" environment, ensuring that only allowed libraries and system resources are accessed. The end-to-end security is implementation-dependent and done by for example encryption.

The KVM is a virtual machine converting the application's bytecodes into machine-level code, and is closely related to the CLDC. The KVM must support class file verification, which ensures that applications do not perform any dangerous operations. The verification process is divided into two phases; preverification and runtime verification. Application execution is prohibited if it doesn't pass the runtime verification.

A MIDlet is also associated with a protection domain, defining the permissions that can be granted to the MIDlet. Permissions consist of two types; allowed permissions and user permissions.

User permissions are permissions for which the user is consulted, containing three modes; blanket, session and one-shot. The allowed permissions are permissions which can be done whiteout user consulting, if the protection domain in which the MIDlet is associated allows it. Unsigned MIDlets are by default placed in the untrusted domain, while signed MIDlet can be placed in the manufacture domain, operator domain or trusted 3rd party domain.

The optional SATSA package or cryptographic libraries from for example Bouncy Castle can make your MIDlet even more secure, with abilities to secure confidentiality and integrity.

2.2 Cell Phone Forensic

Mobile phones have become a very popular and important way to communicate and process information of different kinds. The information in these phones may have value as evidence in investigations, and a lot of work has therefore been done to develop methods to extract these evidence items from terminals. Since the evolution has given the phones modern functionality like camera, multimedia, Java etc., more of these evidence items are now stored in the internal memory than on the SIM card. Proper forensic methods to analyze such memory, including deleted files are now available [47].

Java applications will also make use of this internal memory to store its information. To find out the Java MIDlets ability to keep information confidential, these forensic methods need to be examined.

Before we are discussing the different memory extraction techniques, it important to understand how cell phones are built. They are very much constructed like a regular computer, with a processor, the memory and a hard drive as the main components. Today's cell phones don't contain hard drives, but most often an integrated non-volatile

flash memory circuit. The flash memory will be used to store all information that needs to persist during a power loss. The central processing unit (CPU) controls the communication circuits of the phone. The RAM may be implemented as an integrated part of the CPU, or as a separate circuit, and will work as a intermediary storage.

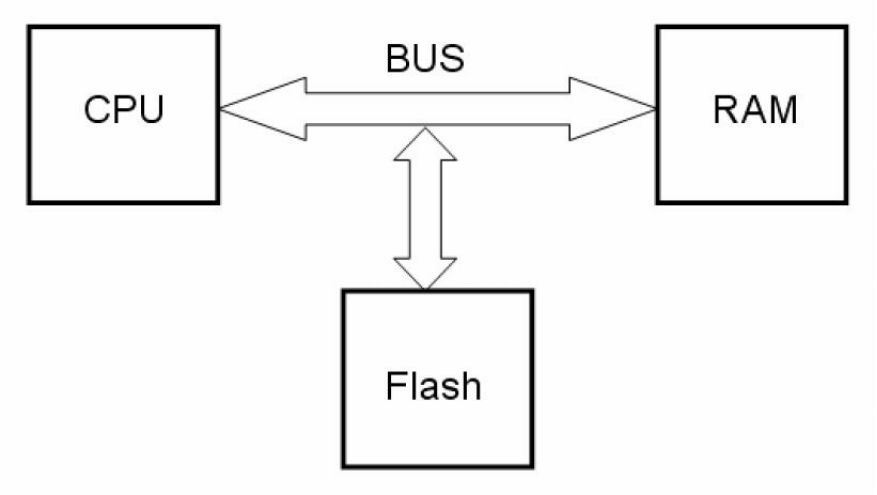


Figure 5: The architecture of a cell phone. Image from [47].

The illustration in figure 5 gives an overview of the components a cell pone consists of. The CPU will in addition to these components also communicate with the SIM card and possible other external storage media like SD or MicroSD [47].

Such external memory cards are often formatted with a conventional file system like FAT, and can therefore easily be imaged and analyzed with standard forensic tools meant for disk drives (e.g. The Sleuthkit software package [48]). A list of commonly available memory cards used in cell phones can be seen in [49]. If a Java MIDlet stores its information in this kind of memory modules, it is quite easy to extract the information. But it is out of the scope of this thesis to go deeper into forensic methods addressing this external storage media. If you are interested in this topic, a good start would be to visit the Sleuthkit's web page [1].

2.2.1 Methods

In [50] different forensics methods to extract information from mobile devices are described. Several forensics applications are also described in [49]. With many of these tools you can read the cell phones internal memory and SIM card entries. Many of these tools are also available in forensics editions, only available for law enforcement agencies and with even more data acquisition tools [51].

Even if these forensics editions should be available to law enforcement agencies only, they will likely be available in criminal networks too. They may even have developed their own special software.

Different kind of information can be stored in both SIM cards and external flash memories. Most of the information will anyway be stored in the internal flash memory [47]. A Java MIDlet's information will also be stored in the internal memory, but the actual

[1] www.sleuthkit.org

location is manufacturing dependent [45].

Most of today's phones can be connected to a computer with a cable or with wireless interfaces such as bluetooth or IrDa. With special software like Nokia PC Suite [2] or MOBILedit [3], will a user be able to access information like SMS messages, images, received calls etc. Direct access to the phone memory will on the other hand not be possible. MOBILedit does also have a forensic edition available for law enforcement agencies. Some of these phone managers will anyway be able to do a backup of the cell phone, including the installed Java MIDlets. Even the RMS related to the MIDlet is it possible to back up with this procedure. A closer look at how this backup procedure may decrease the MIDlet's overall security is discussed in section 4.4.

The different software available will as mentioned also be able to extract information from the SIM card. There will not be stored any information regarding the Java MIDlet on the SIM card, and no further discussion about SIM card information extraction will therefore be discussed.

Tools called flasher can also be used. This is tools freely available on the Internet, and can be used to update the phones software, access the phones memory directly, remove a Service Provider Lock or change the phones IMEI code (International Mobile Equipment Identity). Stolen phones IMEI code will be blacklisted in a register called The Equipment Identity Register (EIR). This EIR is an optional register, and it up to the vendor to implement it or not [50]. A phone with a blacklisted IMEI code will not be able to call out or receive messages, and this is why some people make use of flashers to change the IMEI code.

Making use of the phone menu system is another way to extract information. You will simply use the keypad of the phone, and enter different codes. The phones IMEI code will for example on most phones be found by entering the string $*\#06\#$. With this method you may also find the software version, running tests or whatever the current phone manufacture has implemented.

Another approach to extract data from a cell phone's memory would be to implement an application with the ability to extract the raw data from the phone using the phone's native API [52]. This kind of applications would require direct access to the phone's hardware. A native application for the device's operating system must therefore be implemented in order to make this work. The C++ API would be the choice, since it provides a great deal of functionality which the J2ME lacks. In fact, this kind of application would be extremely difficult or impossible to implement in Java, since the JVM restricts access to the needed functionality in the underlying operative system [52].

This kind of raw data extraction will even make it possible to extract deleted information. In [53] has this method also be been tried on a Windows Mobile smart phone.

The result from these native applications was that even some deleted information was able to be extracted on the Windows Mobile phone, see the results in [53]. There were on the other hand some problems with the memory extraction on the Symbian phones. More details about that can be read in [52].

FExplorer [4] is a file manager available for Nokia Series 60 phones with Symbian OS. With this software you are able to back up your Java MIDlet's RMS. This can also be done

[2] `http://www.nokia.no/support/software/pcsuite.php`
[3] `http://www.mobiledit.com`
[4] `http://www.newlc.com/FExplorer-v1-15.html`

with software like SeleQ [5] or FileMan [6].

Java enabled phones without the Symbian OS may also back up RMS data. With for example Oxygen Phone Manager II [7] you can make a complete backup of your phone, including Java MIDlets and their RMS data. This backup files will then be available for further analysis using a hex viewer. Sensitive information stored in the RMS must therefore never be stored as plaintext. With software like MOBILedit [8], you will also be able to complete this backup procedure.

The best way to analyze the internal memory content will after all be to digitally image the memory content. Manufacturers have often the ability to get access to the phones memory content and update the software. Memory imaging will therefore be possible on many phones. This will however require knowledge to the programming interface of the phone, something that often is kept by the manufacturers themselves.

Two different approaches of physical imaging of a phone's memory is discussed in [47]. These approaches have for its object to desolder and image a memory integrated circuit and the use of built-in test methodology to image memory contents [47]. These techniques needs deep knowledge about the memory circuits, and great care must be used since the circuits may be easily damaged.

After a successful memory extraction, an image of the memory may be analyzed further with for example a standard hex editor [9].

[5] http://www.ximplify.com/
[6] http://www.smartphoneware.com
[7] http://www.opm-2.com
[8] http://www.mobiledit.com
[9] http://www.winhex.com

3 Developing J2ME Applications

In this chapter you will be introduced to the development process of J2ME applications. You will get an overview of the tools you need and how to deploy your final application. A description of the prototypes developed in the thesis will also be given. This will give a better understanding of how it works before the analysis in chapter 4.

3.1 Development Tools

Before we can start developing mobile applications, we need some tools. The development of the prototype used in this thesis was done in an Integrated Development Environment (IDE) from NetBeans [54]. This IDE requires the J2SE to be installed. You are able to download a bundled version containing the NetBeans IDE and the J2SE from [55]. You also need the NetBeans Mobility Pack [54], to develop applications designed for the J2ME platform. The installation procedures can be found together with the tools.

With the required tools just mentioned you are ready to start developing J2ME applications. The IDE's integrated tools like code obfuscator (ProGuard), code completion, emulators and much more, will give you a lot of help in this phase.

The emulators are very useful for application testing. When you have implemented something new, can your application first be tested on an emulator. Emulators will thus ease your debugging, since you do not need to deploy the application to the cell phone every time you are going to test it.

3.1.1 Deploying the MIDlet

When you are finished with the development phase of your application, it is different ways to get the application installed on your cell phone. The best way may be to place the applications (JAD and JAR files) on a web server. With the cell phones browser you can then go to that address, and download the application.

The process of verifying if the MIDlet is trusted is done before the application is installed, by checking the signature attributes in the JAD file. If for example the MIDlets requested permissions (in the JAD file) doesn't belong to the domain the MIDlet is associated with, then the installation fails [31], see section 2.1.3 for more information about protection domains.

Other ways of deploying the application may be with use of Bluetooth or Infrared communication or with a cable connecting the phone and a computer in which the application resides. A verification of the application will also be done before the installation starts.

3.2 Password Store

Password Store is the name of the prototype developed together with the work on this thesis. This application is made with the goal to test if Java enabled cell phones are able to store sensitive information in a secure manner. It is worth to mention that this is a test application, and all of the required functionality which must be implemented in a possible commercial version may not be implemented or tested thoroughly.

The application works like this:

When the application starts, the user is prompted with a master password form, see figure 6. The correct master password must be entered to get access to the applications functionality and stored information.

The first time the application is executed after installation, the password entered will be stored in the RMS as the master password. In a final version this password should be entered twice, to make the user assure that the right password has been entered.

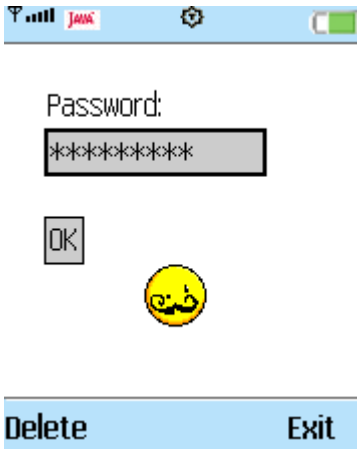


Figure 6: The prototype's master password form.

The master password is not stored as plaintext, but as an SHA1 hash digest [56, 57] in the cell phones record store named "USER".

The SHA1 hash function is among the most used, but its out of the scope of the thesis to investigate the strength of this function any further.

After a correct master password has been entered the applications gives the user access to other functionalities.

The applications main goal is to store different login credentials. This can easily be done by accessing the new password command. You will then be prompted with a form containing fields to enter a name or description, together with username, password and the URL of the site in which the information shall be used. When this information is added, the password is AES [58, 59] encrypted and stored in the cell phones record store "FIELDS".

DESCRIPTION|www.abc.com|USER|34|æ?EsÃüKÂu$OiP-si

The string above is an example of the information stored in the record store "FIELDS". The number 34 is the field's identification number, which is an incremental value for each new record added to the table. The symbols after the last |, is the AES encrypted password.

The AES encryption process makes use of a 128 bit key. This key is derived from the master password together with the entered username as salt and a iteration count of 20. This key derivation is done by a password based key derivation function (PBKDF). The PBKDF is formed as defined by PKCS (Password-Based Cryptography Standard) 5 V2.0 [60].

AES encryption was chosen because this algorithm has got good support in the API

from Bouncy Castle. You can namely chose from three different versions, which differ in performance (AESEngine, AESLightEngine and AESFastEngine). The fact that today's cell phones have no problem to encrypt information with AES, has encryption with Elliptic Curve been omitted, even if this was suggested in the introduction of the thesis. AES is further among the most used algorithms and hard to crack.

Figure 7: The passface command.

The possibility to make use of passfaces [11] as the master password is another functionality of the prototype. By selecting the passface command, figure 7, the user is prompted with a grid of 12 different smiley faces. Each of the smiley faces maps to the corresponding button on the keyboard. In other words, the first image maps to the number one button, the second to the number two button, up to the twelfth image which maps to the ♯-button on the keypad.

After the first image is selected, the user is prompted with a new grid containing 12 images. These 12 images are different from the first 12, see figure 8 and 9. Four such grids are prompted to the user.

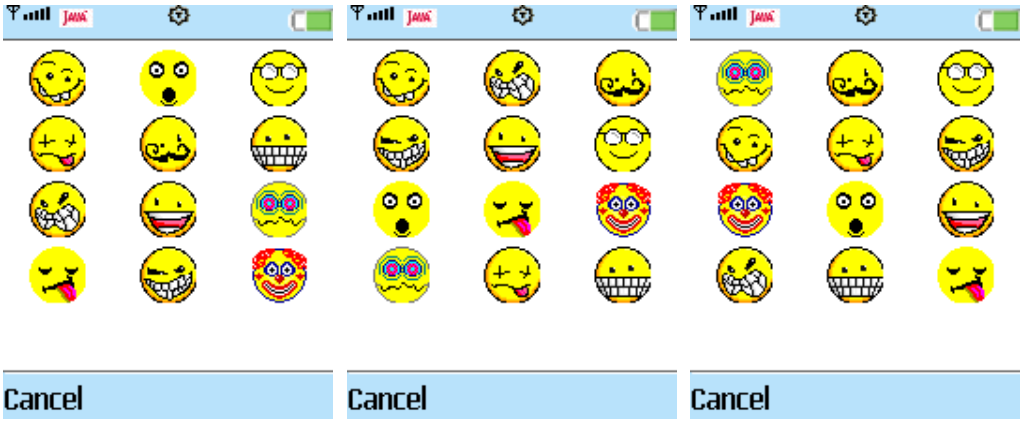


Figure 8: The first grid of the passface mechanism in three different permutations.

The user must pick the right face from each of the four grids to get access to the application.

The 12 faces in each of the four grids are shuffled around every time the passface

authentication mechanism is executed. The same image will therefore most likely map to a different button on the keyboard, than it did the first time it was displayed. Figure 8 shows the first grid in three different permutations. As you can see the smiley faces are shuffled around.

Figure 9: Example of the second grid of the passface mechanism.

The user is free to choose between passface authentication or the regular PIN code. The master password may also be changed at a later time. Since the master password is the "key to the kingdom" [3], it should be handled with care and changed on a regular basis.

This master password update procedure requires the user to enter the old password before a new one can be set. This will hinder an attacker to change the password if the real owner forgets to close the application and leaves it unattended.

Another important functionality, which is not implemented in the prototype, would be automatic closing after a given time of inactivity. Let's say that the application shuts down after three minutes of inactivity. In that way, the forgetful person doesn't need to worry about if the application is running or not. An attacker will then need to get access to the application within these three minutes to change the master password or read other stored passwords.

This mechanism may easily be implemented by a thread which runs the exitMIDlet function, after the given time if inactivity.

A complete data float diagram (DFD) of the authentication procedure can be seen in figure 10.

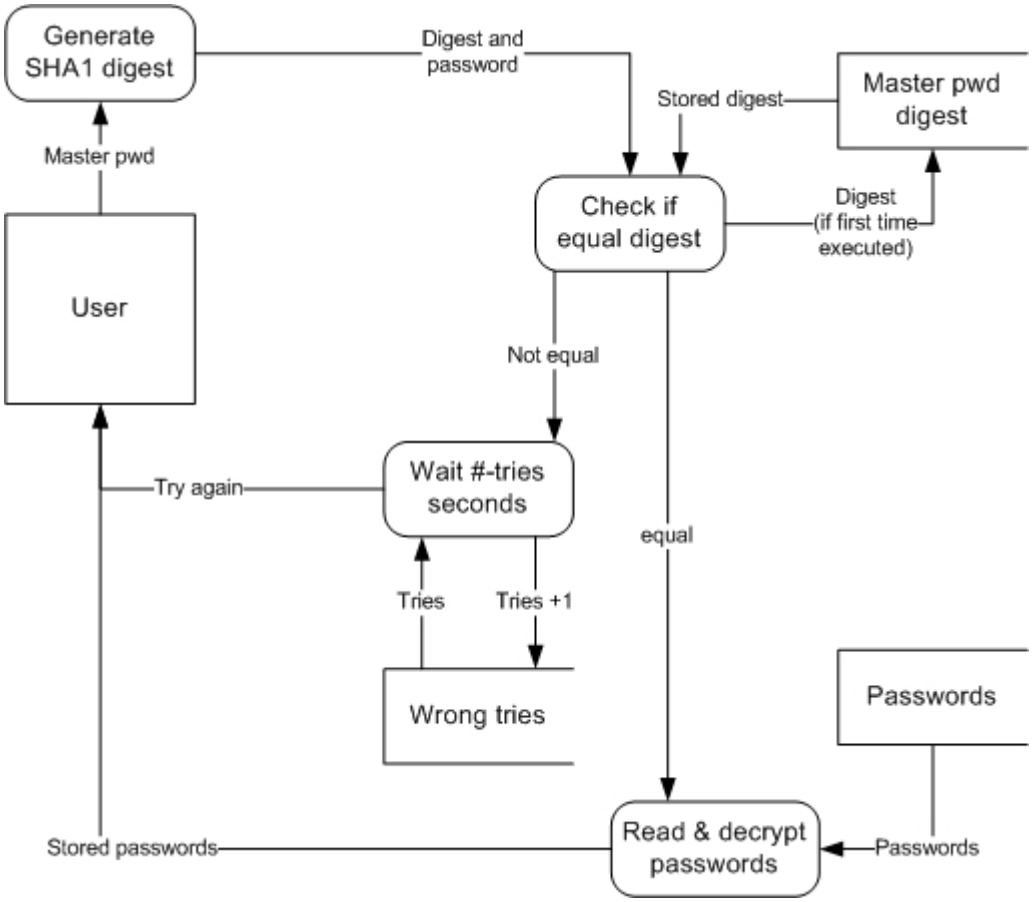


Figure 10: DFD of the password store prototype's authentication procedure.

3.3 SMS-Delete

SMS-Delete is the name of the MIDlet which gives the user the ability to do a remote deletion of the sensitive information stored in the Password Store application.

If the cell phone, in which the Password Store MIDlet resides, is lost or stolen, it is useful to have the ability to remotely delete the sensitive information.

The SMS-Delete MIDlet works as follow. When it starts, the user is given the opportunity to enter the phone number of the lost phone, and the master password which protects the sensitive information. A SHA1 digest is then computed from the entered password, and sent to the lost phone with help of a regular SMS message. These SMS messages are in the prototype being sent on port number 3536.

You can invocate the following functions, to send SMS messages to a given phone and port number.

```
smsCon = Connector.open("sms://" + phone number + ":" + 3536);
smsCon.send(message);
```

When the lost or stolen phone receives this SMS, the Java application manager knows it shall start the Password Store application and delete the information stored in the RMS. The push functionality is disused in more details in section 2.1.6.

A data flow diagram of the SMS-Delete MIDlet can be seen in figure 11.

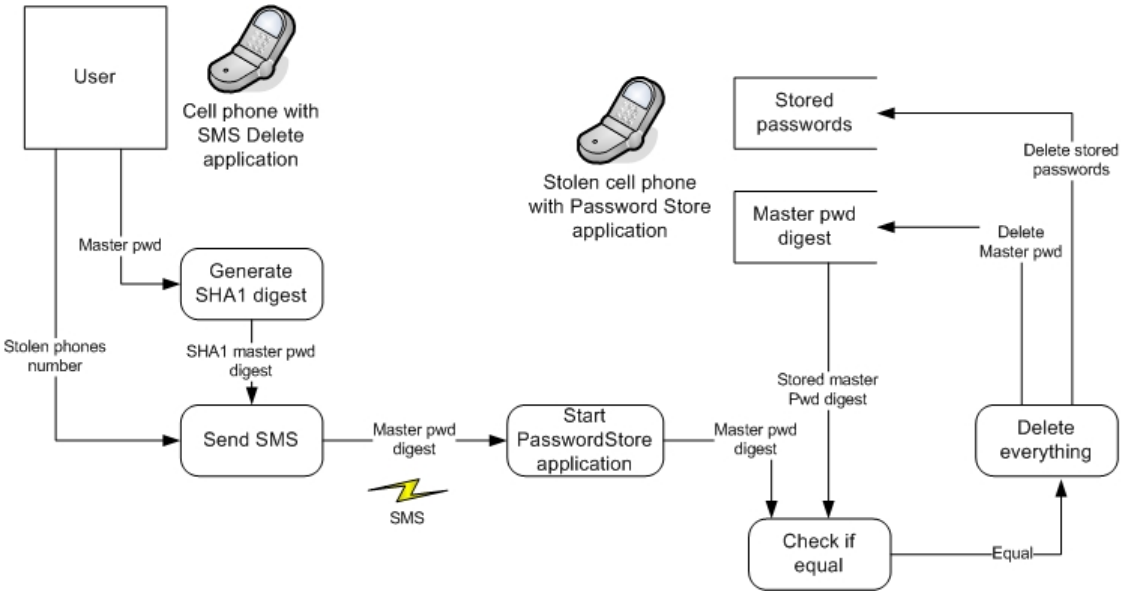


Figure 11: DFD of the SMS-Delete prototype

The user must, as mentioned, enter the master password of the Password Store MIDlet to be able to remotely delete the record store. In a final version this password should be a password that differs from the master password used to enter the application. So that there is one password for each functionality.

4 Security Analysis

There are several papers on J2ME and security of mobile devices, but they all tend to focus on security of the communication link between the cell phone and a server [10, 37, 61, 62]. A security analysis of a single MIDlet will therefore be done in this chapter, with the aim of finding strengths and weaknesses. The analysis will focus on the prototype developed together with the work on this thesis.

Through this chapter the prototype MIDlet will be called "Password Store". The other MIDlet which perform a remotely deletion of the RMS, will be called "SMS-Delete". The last MIDlet developed will be named "Stealing-MIDlet", because it steals information stored in the RMS by other MIDlets.

An overview of different attack types will first be identified. This will be done with the help of flow diagram. With this work, an identification of threats which seemed unimportant in the first place may be discovered.

We will then take a closer look at the master password authentication mechanism. In this section different approaches are discussed to find out which method is the best one to secure the stored information.

An overview of different attack types is then discussed in the end of the chapter, where we take a closer look at the Stealing MIDlet and the use of phone managers.

4.1 Threats and Countermeasures

A cell phone with a Java MIDlet may be vulnerable to many different threats. With help of flow diagrams, it will be easier to identify these threats. This is done by placing a hacker in every possible situation, from the production of the cell phone to the Java MIDlet has been installed on the device.

Each of the flow diagrams will identify the hacker's possible actions. Different countermeasures will then be discussed, with the aim of eliminate or reduce a possible consequence.

4.1.1 Attacking a New Phone

In the first situation, the cell phones are being produced and sent to a store. A possible user may then come to this store and buy a phone. In this flow, a hacker A, may place himself between the cell phone supplier and the store, or between the store and the user, see figure 12.

A possible hacker (A) may be placed either as a worker at the cell phone supplier, a worker at the store or as an external participant like a transporter. Hacker A may also be a person working with the manufacturing of the phone.

Some of the threats hacker A may be able to carry out:

- Update the cell phone software with an older and maybe vulnerable version.
- Install a trojan software, for example a keyboard sniffer.

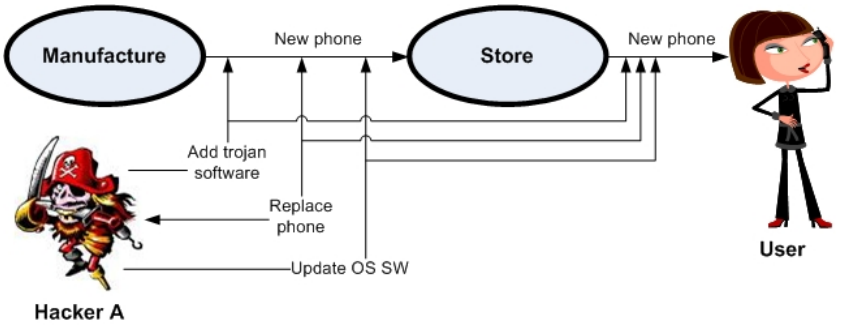


Figure 12: Data flow between the cell phone supplier, store and user.

Hacker A may materialize these threats by replacing the phones with new ones, or install or update the phones directly. The hacker A must in other words be able to physically handle the cell phone.

Consequences

The consequences of the threats are that the cell phone already has trojan software installed when a user buys the phone. In the case of a keyboard sniffer, every button the user pushes may then be sent to the hacker. He may therefore get both the master password and the other login credentials sent to himself when the user installs and uses the Password Store MIDlet.

Encryption will in this situation also be futile. The keyboard sniffer will recognize the pushed buttons and will in that way know what has been typed in before the encryption process has been carried out.

Countermeasures

If a trojan software or an old operative system software with exploitable vulnerabilities is installed on the cell phone, there are not much that can be done with respect to the development of the MIDlets.

One thing a user should be aware of is the cell phone's software version. This should be the newest available to the current phone model.

The software version can often be found easily by enter a sequence of different buttons. On many Sony Ericsson models you may find the version by entering $> * << * < *$ (the $>$ and $<$ means to push the joystick to the right or left). On many Nokia phones this can be achieved by the sequence $*\#0000\#$. These secret phone codes can easily be found by searching the web for the relevant phone model.

If the cell phone shall handle very sensitive information, an idea would also be to re-flash the phone memory with an image that is known to be clean. The user will in that way be sure of that no trojan software is hidden in the phone. This must then be done without the use of external persons. A hacker B can namely, as explained in the next section, appear in such a situation.

Many of these counter actions will be hard to carry out for a regular user. An easier approach could be to implement an information page, which returns the desired values. An attacker could however easily hard code this information page and fool the user.

4.1.2 Update the Phone

Another situation may be when a user needs to update/upgrade the cell phone's software. A hacker B will then, as illustrated in figure 13, appear as a threat. He is placed between the user and the supplier doing the updating work, for example by examination of the users or suppliers mail. The hacker may also be a person working at the supplier, and in that way have full access to the phone and lot of time to perform an attack on it.

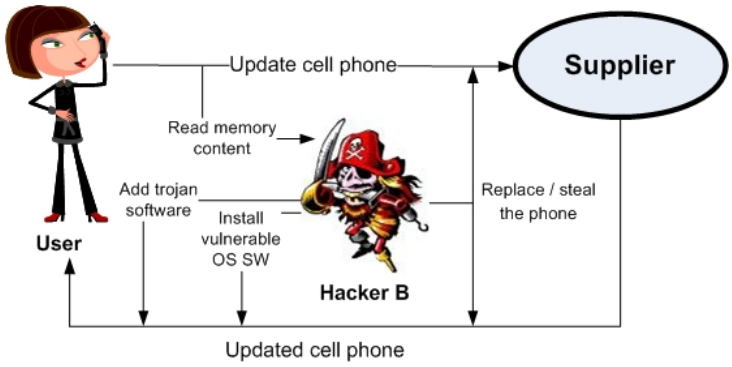


Figure 13: Data flow between the user and supplier, when the phone is sent for updating.

Hacker B will in addition to the threats made by hacker A, have the possibilities to get access to the users personal data stored in the phone. As discussed in section 2.2 and 4.3, this can be done with forensic methods or with a stealing MIDlet. The easiest way for a hacker in this situation will be to make a backup of the phone, as discussed in section 4.4. He can then perform a brute-force attack on the stored Java MIDlet, with help of the information from this backup.

The threats made by hacker B:

- Update the cell phone's software with an older and maybe vulnerable version.
- Install a trojan software, for example a keyboard sniffer.
- Extract user data from the cell phone's memory.

Hacker B may install and update the software directly or by replacing the phone with another one. In the case were the hacker is working at the supplier, he has directly access to the phone, and has got a lot of time to perform the attack. If he has to crawl the user's mail, it may be easier to install trojans or update the software on another identical phone and replace it with the user's phone. It may even go quite some time before a phone replacement is discovered. The hacker may also steal the phone without replacement of another one.

If hacker B is going to install a trojan software or update the software version, it should be done after the phone has been updated by the supplier. If this is done before the updating, it may be discovered and erased by the supplier. This would not be a problem if the hacker is the one doing the actual updating/upgrading work. He will then have the possibility to read the phone's memory before the updating procedure and install trojan software after updating has taken place.

Consequences

The consequences in this case are that the user's data are extracted from the phone. All data stored in the phone will therefore become known to the hacker, including the data stored in the Password Store MIDlet, see section 4.3 and 4.4.

Another consequence is that the phone is updated with hidden trojan software. Like the case with hacker A, this may be a keyboard sniffer. The result will anyway be that the master password and the stored login credentials in the Password Store MIDlet become known to hacker B.

The user's cell phone may also be updated with a new version of the Password Store MIDlet. This new version may include hidden code (trojan), which for example gather information and sends it to hacker B. The hacker must in this case know the source code of the Password Store application. If he doesn't have this source code it may also be possible to implement a similar application. This will then require that the hacker knows the fundamental functionality like the name of the record stores, encryption algorithms being used and what the GUI looks like. Anyway, it's essential that the new hacker MIDlets is looking identical for the user, to prevent detection.

Countermeasures

Some countermeasures to the threats hacker B is in charge of may be to encrypt the stored information. If the information stored is encrypted with a key derived from the master password, it is important that this master password is a strong one. A short master password will namely easy be brute forced, like the example in section 4.3.1, but this assumes that the hacker knows how the encryption is done or has got the source code. "Security by obscurity" may therefore be a countermeasure. This is not a very good protection mechanism and should never be used as the only protection. But it may increase the overall security to some degree. Code obfuscation may also be a keyword in this process, which makes it harder for an attacker to de-compile the source code.

The user should personally check the size and information of the MIDlet. If the Password Store MIDlet has got a greater or smaller size after he gets the phone back, it's time to get suspicious. A properly implemented hacker MIDlet will in any case have the same size, to prevent discovery.

Signing the MIDlet will also be a way to hinder a stealing MIDlet to be installed. A signed MIDlet should namely never be updated with an unsigned MIDlet, as stated in the MIDP specification [21]. But this can easily be bypassed if the hacker cooperates with hacker D, who may be in possession of the signing key.

If the cell phone's data is extracted with forensic methods, an encryption will be harder to decrypt. The hacker will in such a situation only have binary file containing encrypted information. He will therefore need to find out where in this file the encrypted information resides, and perform a brute force on that data. This information may be able to get from the cell phone specification of current interest.

If a brute force attack on data extracted from forensic methods shall succeed, will hacker B also need to know how the encryption is done, with respect to algorithms and key derivation.

The user may also delete the Password Store MIDlet or other sensitive information stored in the phone before it is shipped for updating. Even if deleted information is not impossible to extract, it will require more work and knowledge from the hacker to be

carried out. If deletion should be convenient, a proper method for data synchronization should be implemented. The user will then be able to restore the deleted information when he gets his phone back, and there will be no need to enter every single record once again [1].

Last but not least, the user should activate both the PIN code and the cell phone lock. This will make the hacking work way harder, even if it exist flashing software to bypass such protection [2]

4.1.3 Phone Sent to Repairman

In the next case we are discussing the situation when a phone is broken and sent to a repairman. The threats made by this hacker (C) are the same threats discussed in the section about hacker A and B.

If the phone is going to be repaired, it is often something like a new display, new buttons or other hardware. This means that hacker C may install the trojan software or update the operative system before the actual repair of the phone has been done, since the repairer most likely not are going to re-flash the cell phone software. The hacker should in any case install the trojan after the repair has been done, to maximize the probability of not being detected.

The threats created by hacker C:

- The memory content is extracted.
- Trojan software is installed.
- Cell phone software is updated/downgraded.
- The phone is stolen or replaced.
- The MIDlet is updated with trojan functionality.

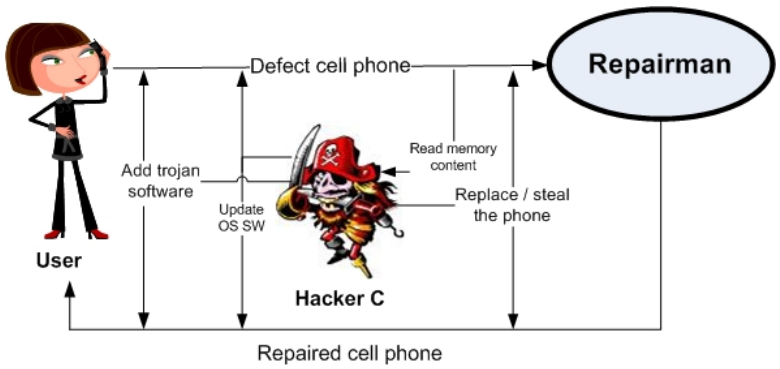


Figure 14: Data flow between the user and repairman, when the phone is sent for repair.

[1] An analysis of a synchronization interface is not done in this thesis, and is an item of further work.
[2] Twister flasher is a flashing utility, available from `http://gsmserver.com/software/Twister.php`.

Consequences

If these threats materialize, the consequences will be much like in the two previous examples. The phone has been updated with software able to read the users interaction with the keyboard, or the memory content has already been extracted.

If the memory content has been extracted it may be just a matter of time before the stored login credentials become known to the hacker, even if the content is encrypted. A fast computer may complete a password crack with many times more guesses per second than the cell phone itself can carry out. A pre-created dictionary may also be used, to speed up the work.

An attack may also be done if the MIDlet has been updated with a password cracking mechanism (like a Stealing MIDlet), but slightly less effective. The hacker in this situation will also need to have the phone over some period of time to complete the cracking, depending on the size and configuration of the password. Source code or knowledge of the MIDlet's functionality must also be known to the hacker, to make this method work.

Countermeasures

The same countermeasures as for the examples about hacker A and B will also here manifest itself. In addition may an updating of the Password Store MIDlet prevent trojan code to be hidden in the application. A MIDlet installed with the same name and vendor are treated as an update. This can be used to prevent hacker MIDlets in the same way the hacker MIDlet is installed as an update, see section 4.3 for more information about the Stealing MIDlet. In other words, the user should update the Password Store MIDlet when he gets his phone back, to be sure of the integrity of the application and overwrite possible trojan code.

4.1.4 Attacking the Developer

Hacker D is important to take a closer look at. It is this person who can get his hands on the MIDlet's source code. He will also be able to replace the developed MIDlet's JAD/JAR files with his own. As can be seen in figure 15, he is placed at the developers spot.

The threats made by hacker D will therefore be:

- Sniff source code.
- Add hacker code to the source.
- Replace JAD/JAR.
- Steal private signing key.

Consequences

If some of the threats made by Hacker D materialize, this could lead to great consequences. The source code may go astray, giving the hacker community knowledge about encryption and hash algorithms being used, functionality, GUI, etc. But as mentioned, the MIDlet's security shouldn't be compromised because of this. It is the protection mechanism itself that shall keep the information safe, and not the fact that it is not known to the environment. For example will a firewall protecting a network be a protection mechanism. The fact that the hacker has knowledge of this, wouldn't compromise the security

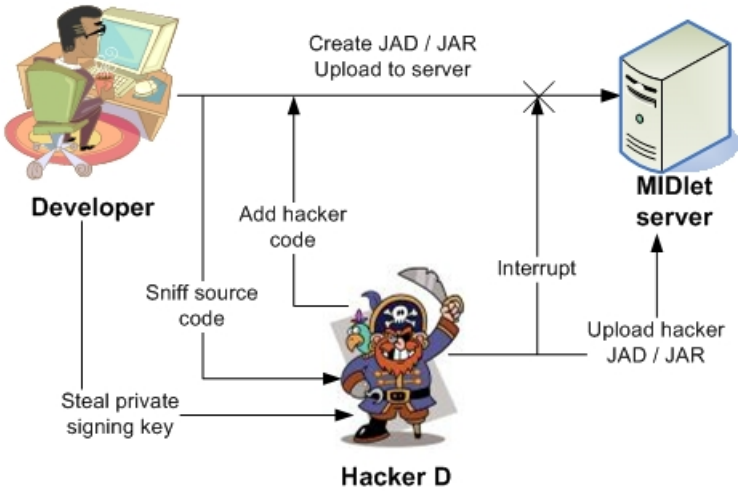


Figure 15: The data flow between the MIDlet developer and the MIDlet server.

of the network cf. Kerckhoff's principle [3].

The problem goes even further. A Stealing MIDlet can easily be implemented with help of the source code, see section 4.3. The Stealing MIDlet can be implemented with functionality which sends the login credentials to the hacker or store them for later retrieval. This Stealing MIDlet's JAD/JAR files can then be replaced with the original Password Store MIDlet's JAD/JAR files on the server. A new user will in this way download a trojan infected Password Store MIDlet from the server.

Hacker code can even be injected in the original source code before a signing procedure. A signed Password Store MIDlet could therefore also be infected with trojan functionality. Signing a hacker MIDlet may also be performed if the private key is stolen.

Countermeasures

There are some different actions that can be taken to handle these threats. Firstly, a user will be assured of the origin of the MIDlet if it is signed. A replaced JAD/JAR file on the server may also be signed, but not with the same key as the original developer is in possession of. A user should therefore carefully examine the signed MIDlet; to find out if it is comes from a trusted source. In any case, the hacker may also steal this private signing key. If this happen we are in trouble, but it is out of the scope of the thesis to analyse how a private key should be secured.

4.1.5 MIDlet Installation Request

Hacker E is in close cooperation with Hacker D. Since Hacker D is the one who has the ability to sniff the Password Store MIDlet's source code. Hacker E will most likely use this source code to develop his own hacker MIDlet and upload it to a server he is in possession of. When a user wants to install a version of the Password Store MIDlet, he will try to contact the server in which is it located. Hacker E will interrupt this request, and relay the user to the hacker server. The user will therefore download a version of the hacker MIDlet, instead of the real Password Store MIDlet. This is illustrated in figure 16.

Hacker E may also work on his own, and develop a hacker MIDlet with the same

[3] Kerchoff's Law: A cryptosystem should be secure even if everything about the system, except the key, is public knowledge [63]

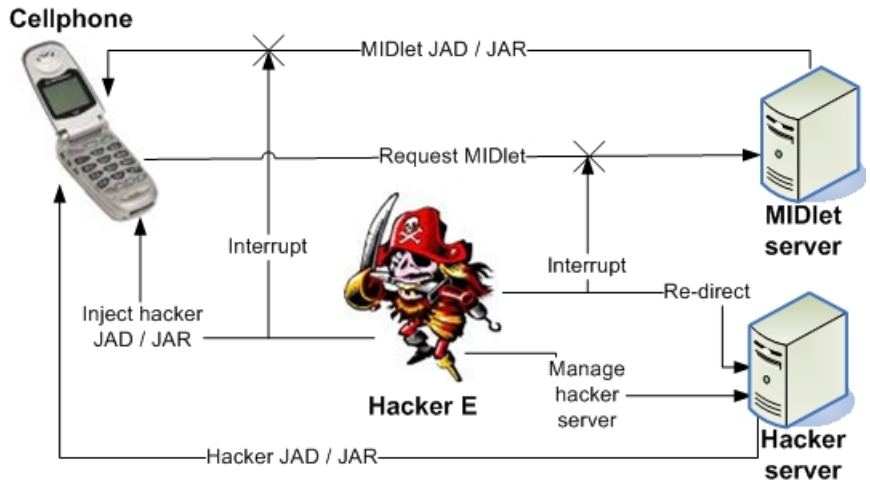


Figure 16: Hacker E may interrupt a MIDlet installation request, and re-direct it to the hacker's server.

functionality as the Password Store MIDlet. But he must in such a situation first examine the Password Store MIDlet thoroughly, to find out what it looks like and how the main functionality works. A bad imitation will easily be detected, and that is not the goal from the hacker's point of view.

The threats from hacker E will therefore be:

- Interrupt a MIDlet request and re-direct it to a hacker server.
- Interrupt the originally reply, and inject hacker JAD/JAR.

Consequences

The consequences of a successful attack from Hacker E will be that a Hacker MIDlet imitating the Password Store MIDlet is installed on the user's cell phone. This MIDlet could for example store login credentials as plain text, or send it to the hacker by SMS. The result will anyway be that the user installs and makes use of a vulnerable MIDlet, and whiteout any knowledge about it. In any case, the trojan functionality hidden in the MIDlet is it up to the hacker to decide.

Countermeasures

Hacker E will as mentioned only create threats concerning the MIDlet. MIDlet signing will therefore be a way to prevent this threat to materialize. We must then assume that the private key used in the signing procedure has not been compromised. The user should then carefully examine the signature of the downloaded Password Store MIDlet. If the downloaded MIDlet is not signed, and the user knows that is should be, it is time to get suspicious. We will in such a situation most likely deal with an unsigned Hacker MIDlet.

It would also be a good idea to carefully examine the downloaded MIDlet's functionality, size, GUI appearance and name. If the downloaded MIDlet then has a web page with screen shoots and descriptions of functionality, name etc., can differences be detected.

Any differences may be a result of a badly implemented hacker MIDlet.

4.1.6 Sniffing

Hacker F is the person who is sniffing the keystrokes on the target phone. He may also be able to interrupt keystrokes and inject his own, see figure 17. If Hacker F shall succeed, he must cooperate with Hacker A, B, C or D. This cooperation must be in place because Hacker F has not the same ability to install the sniffing/trojan software on the phone. But if we push things to extremes, will everyone be able to steal a phone and do whatever they want with it. It is only the knowledge which sets the limit.

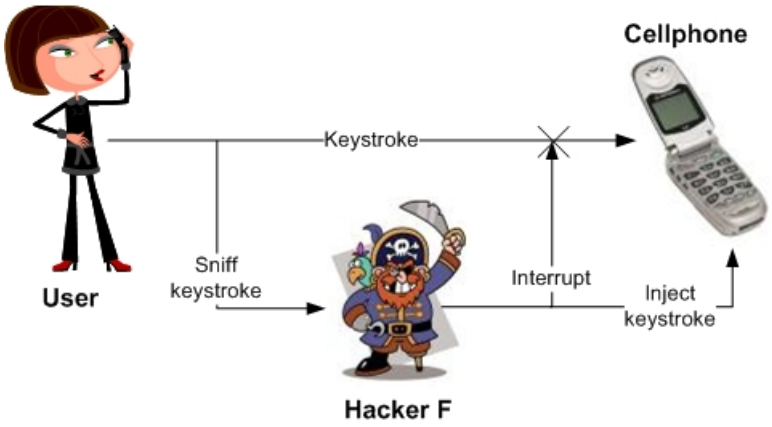


Figure 17: A hacker F sniffing the users keystrokes.

Consequences

If a hacker F is able to interrupt keystrokes and inject his own, could a variety of different situations arise. Let's say, the user enters his master password for the first time. The hacker can then interrupt these keystrokes, and enter his own chosen one's. The user will then think his password is the one he has just chosen, while the password in fact is a password chosen by the hacker. Even the encryption of the later entered login credentials will be under the influence of this, since the master password is acting as the key in the encryption process.

The hacker may also be interested in a DoS attack. Lets say he inject his own keystrokes when the user wants to enter his login credentials to different sites. For example could a user's password to a email site be "Asklo3xRo6". The hacker could for example replace this password with "qwerty". When the user wants to log into his email account, using the Password Store MIDlet [4], the login will fail. A bad password has namely been entered. Many situations like this will weaken the trust to the Password Store MIDlet, and this may be the goal in the hacker's point of view.

If hacker F also has the ability to sniff the entered keystrokes, will both the entered master password and the entered login credentials go astray. This is a dangerous situation, because the hacker will then have the possibility to log into the users different sites right away and without any brute-force or dictionary attack. This is done by sniffing the login credentials as they are entered into the Password Store MIDlet.

Interruption and injection of keystrokes may therefore lead to very bad master pass-

[4]Mats Byfuglien [1] has developed a full working prototype of this mobile SSO solution.

word, and further a bad encryption of the later entered login credentials. The MIDlet's trust may also disappear.

Countermeasures

We can prevent hacker F to prove a success in some different ways. Like previously mentioned, it is important to examine the Password Store MIDlet's and try to find out if it is the software it claims to be. This can be done by an inspection of the signature, the name of the MIDlet, name of vendor, MIDlet size, appearance, functionality etc.

The sniffer software may also be a native application hidden in the cell phones OS. If this is the case, it is important to update the cell phones software, by re-flashing it with an OS known to be clean. As stated in [64] and [65], there are an increasing amount of viruses developed to target cell phones. It is nothing that may hinder these viruses to work as keystroke sniffers. The anti virus software producers will therefore have an important role in the future of the mobile phones. Anyway, the keystroke sniffers have to send information back to the hacker. This will charge the user. Suspicious network traffic can therefore be detected with a detailed bill, making the user aware of the problem.

4.1.7 Attacking the SMS-Delete MIDlet

Hacker G is a hacker that operates in the link between a cell phone containing the Password Store MIDlet and a cell phone with the SMS Delete MIDlet installed. The SMS Delete MIDlet will mainly be used in situations where the phone containing the Password Store MIDlet is lost or stolen, and is used to remotely delete the sensitive information. In this thesis hacker G will be the one who has the ability to steal phones. An illustration of the data flow between the SMS Delete MIDlet and the Password Store MIDlet can be seen in figure 18.

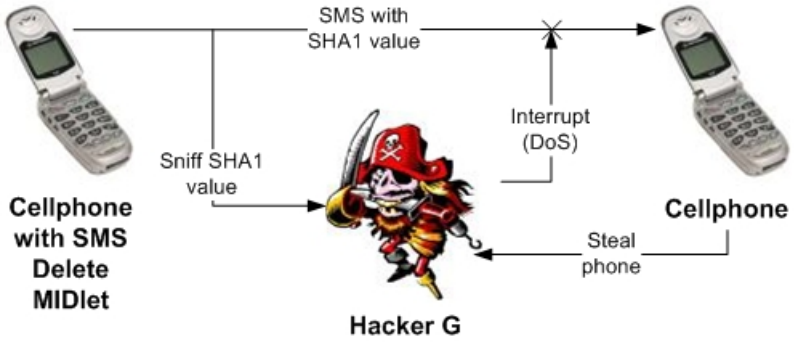


Figure 18: The data flow between the SMS-Delete MIDlet and a phone with the Password Store MIDlet installed.

Hacker G can in this situation sniff the SHA1 value sent from the SMS Delete MIDlet. If he also starts a DoS (Denial of Service) attack on the Password Store MIDlet, he will prevent the SMS Delete MIDlet to do a remote deletion.

The threats made by hacker G will therefore be:

- The phone is stolen by the hacker.
- SHA1 value sent from the SMS Delete MIDlet is sniffed.

- DoS attack on the cell phone containing the Password Store MIDlet, to prevent a remote deletion.

Consequences

If hacker G manages to sniff the SHA1 value sent from the SMS Delete MIDlet, he can make us of this value in a brute-force attack. This can be done really fast on a regular computer, especially if the password from which the SHA1 value is generated is a short one. The hacker must know what kind of hash algorithm that has been used to succeed with this.

If the hacker also manages to prevent the remote deletion he can use the broken password to log into the Password Store MIDlet and gain access to the stored login credentials. The Password Store MIDlet will start up automatically when it receives a SMS message from the SMS Delete MIDlet. This auto start feature can easily be disabled by the cell phone's underlying OS.

Countermeasures

Since the SHA1 value being sent from the SMS Delete MIDlet can be sniffed, and later be used in a brute-force attack, it must be handled with care. In the prototype design will this SHA1 value be the same as the value generated by the master password. This is not a good solution, because a single password is used in two different domains.

The prototype should therefore be implemented again with two different passwords. One password that should be used by the SMS Delete functionality and one used as the primary master password to enter the Password Store MIDlet. Since there will be two different passwords, can a sniffed SHA1 value from the SMS Delete MIDlet not be used in a process to access the Password Store MIDlet, even if it is successfully broken by a brute-force attack.

Automatic startup of a MIDlet can easily be disabled by the cell phone's underlying OS. A proper encryption of the stored login credentials will therefore be extremely important, since this is the only protection mechanism we can count on. The encryption must be able to resist an attack over the period of time it takes the user to update his passwords.

4.1.8 An Overview

The cell phones are exposed to many different threats. An overview of the possible hackers will therefore be presented. This overview is illustrated in figure 19, and may be a help in understanding the big picture. A big version of this illustration can be found in appendix B.

As can be seen from figure 19 there are many different hackers operating in different areas as the phone is handled. Many of these hackers may cooperate, or they may even be the same person. If hacker G steals the cell phone, some requirements must be fulfilled in order to extract the information stored in the phone. If he for example is going to extract the internal memory content by forensics methods, he will need information about how this is done and how the Password Store MIDlet encrypts and stores information. This information can he get from hacker D. If he on the other hand is going to use the Stealing MIDlet approach, see section 4.3, he will also need to get the source code or information about the functionality from hacker D.

In figure 20 there is an illustration of a hacker that is not included in the overview

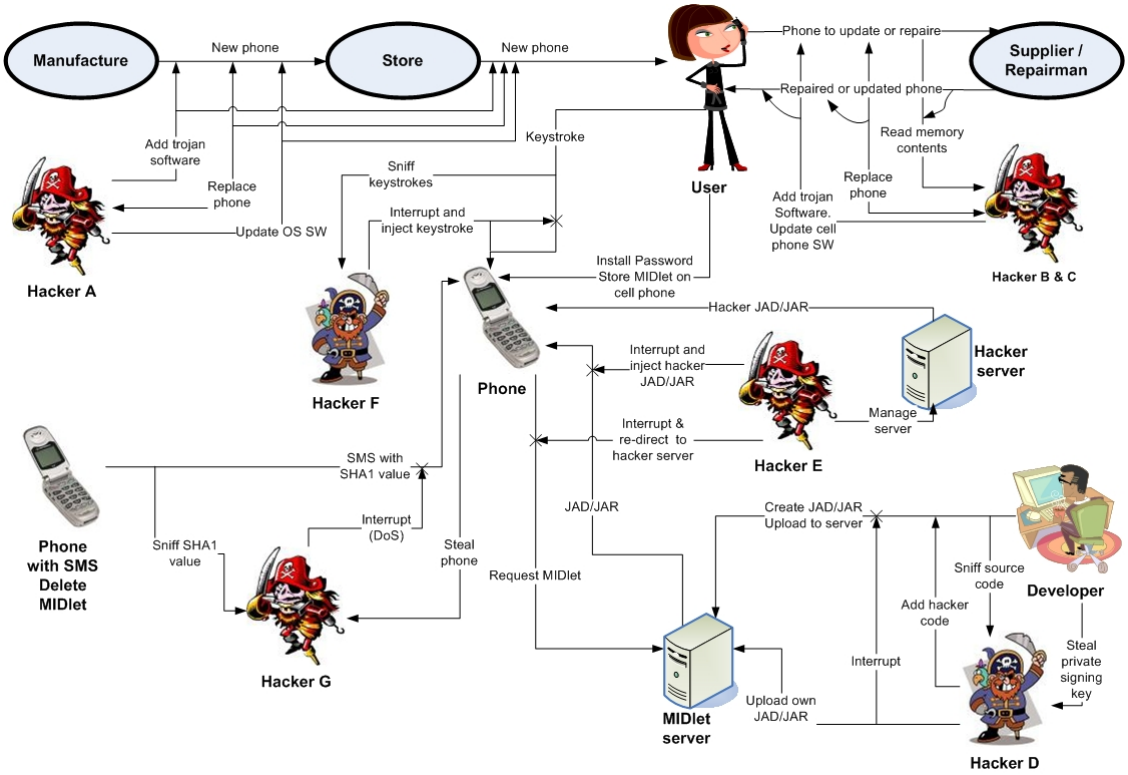


Figure 19: Complete overview of possible threats.

illustration above. Hacker H in this figure may be a person who wants to get his hands on the information stored in a particular phone. He may therefore cooperate with some of the other hackers to achieve his goal.

The phone can he get from hacker G, who steals it from the target person. He could also steal it himself, but hacker G is in this setting a professional thief. He may get a stealing MIDlet from the hacker server managed by hacker E. This MIDlet can then be used to extract information from the MIDlet which holds the information hacker H wants. Hacker D can then give away source code or information about encryption and hash algorithms. This information must be used to perform a brute-force attack. Hacker H may also use other procedures like forensic methods or phone managers, to extract the wanted information. Information from hacker D will anyway be important, if a brute-force attack shall be carried out. If the goal is to prevent the real user to use his stored information, he may even delete the phones content.

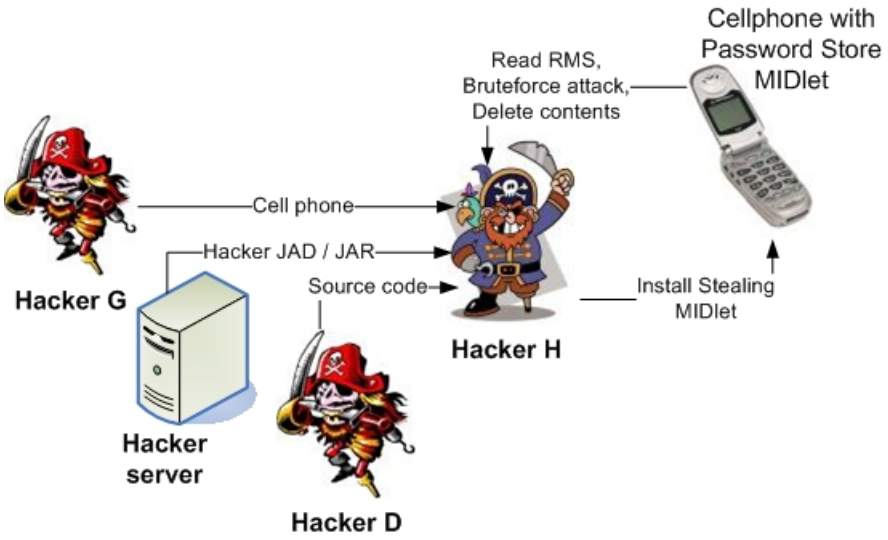


Figure 20: An example of how the hackers may cooperate to achieve their goals.

4.2 The Master Password

If the phone contains sensitive information, it is important that the information stored remains secure. One way to do this is to implement a master password, which is used to prevent unauthorized access and as the key in the encryption process of the stored sensitive information.

The master password is a user chosen password that must be typed in before you gain access to the stored login credentials. There are many varied technologies providing this kind of SSO solution, for example ticket based, certificates and cookies [2]. A good overview of SSO can also be found in [1].

There are different approaches to implement the master password. Probably the most used method is to make use of a regularly password together with a username. But to enhance the security a multi factor authentication procedure is to prefer [11]. This means that you don't rely on only one security mechanism, but put two or more mechanisms together to enhance the robustness.

An example of this type of authentication is the use of biometric together with a password (something you are and something you know). In this situation you need to know the real password together with the right biometric (for example a fingerprint), to gain access to the application or system. A PIN code (Personal Identification Number) together with a speech recognition mechanism would also be a good authentication mechanism. With help of J2ME's Speech API (JSAPI, JSR 113) [66], this could easily be implemented. The problem is that none of today's Java phones support this API yet, but this will most likely happen in the near future [5].

Another important thing to think about is the password configuration. The fact that several characters maps to the same button on the small keyboards, will require to push one key multiple times to get to the right character. If for example key number 3 maps to the character d, e and f, you have to push it 3 times to get character f. If a password on these small devices shall contain letters, it must then have the last entered character

[5] A topic for further work would be to implement and analyse a speech recognition MIDlet, see chapter 6.

visible for the user to know what has been typed in. This will make the password vulnerable to eavesdropping, since the last entered character can be seen on the display, see figure 21. A password containing only digits can on the other hand replace the last entered number with a star (∗) right away, and in that way make the eavesdropping task harder.

Figure 21: The last entered character must be visible, because many different letters maps to the same button.

At this point, you know that there are several methods to implement a master password. In the prototype, you can choose from two different mechanisms, PIN and passfaces. Passfaces are discussed further in section 4.2.2.

The master password PIN code in the prototype must contain at least 8 digits from 0 to 9. There is no need to enter a username, since this is cumbersome and time consuming to enter on small keyboards.

Let's say you had to enter both username and password. Like mentioned, it is cumbersome to enter text strings with the cell phone's small keyboard. The username will therefore most likely be a short one, containing only a few characters, lets say 4. The password will must likely also contain only a few digits, let's say 4 also here, like a regular PIN code.

If a brute force attack on the username and password approach needs to search through every possible combination of usernames and passwords to get the right combination, the numbers of tries as explained in equation 4.1 are needed [6]. In this equation its supposed a username of 4 letters from the Norwegian alphabet containing 29 different characters. The password is supposed to contain 4 digits from 0 to 9. This is just for demonstration; the username would most likely also contain lowercase, uppercase and digits, resulting 68 different characters.

[6]Keyspace: In cryptography, an algorithm's key space refers to all possible keys that can be used to initialize it. Taken from `http://en.wikipedia.org/wiki/Keyspace`.

$$\begin{aligned} keyspace &= 29^4 * 10^4 \\ &= 2,9^4 * 10^4 * 10^4 \\ &= 2.9^4 * 10^8 \\ &= 7.07281 * 10^9 \end{aligned} \tag{4.1}$$

An approach with only a PIN code will then require almost 10 digits to be equally strong as the username and password approach with respectively 4 characters each, see equation 4.2

$$\begin{aligned} 7.07281 * 10^9 &= 10^x \\ \ln(7.07281 * 10^9) &= \ln(10^x) \\ \ln(7.07281 * 10^9) &= x * \ln(10) \\ \frac{\ln(7.07281 * 10^9)}{\ln(10)} &= x \\ x &= 9.849591992 \\ x &\approx 9.9 \end{aligned} \tag{4.2}$$

Even if a username and password in the combination approach are more than 4 characters long, people often tend to pick easy to remember username and passwords, like the name of their pets, girl friends and so one. In [4] you can see a more thorough analysis of passwords.

Since a relative long PIN code needs to be entered if it shall obtain the same strength as a username and password approach, special mechanisms can be used to make the PIN easier to memorize. Pass phrases is one such mechanism.

4.2.1 Pass Phrases

Pass phrases [67] is a not very well-known method to generate passwords. With this method a user may pick a long and hard to break password, which is also easy to remember. By thinking of a sentence or phrase, the user can extract the first character (the last, the first two, etc.) in each word of the phrase to form the password or PIN. The user can then push the desired button on the keyboard, in which the character maps.

> Mom is 45 years old, and has her birthday at the 22'nd of October.

The example phrase above would form a 14 characters long password when the first character of every word is chosen. The password will then look like this:

> M, i, 4, y, o, a, h, h, b, a, t, 2, o, O.

The same approach may be used on a numeric PIN code as well, since each of the letters in the alphabet maps to a digit from 0 - 9 on the cell phone's keypad. The only difference is that you will get a number when you push the button containing the desired letter. For example will the letter M map to the digit 6, i will map to 4, y to 9 and so one, resulting in a 14 digits long PIN code:

> 6, 4, 4, 9, 6, 2, 4, 4, 2, 2, 8, 2, 6, 6.

Even if this passphrase approach may give you a long PIN code, it may not be as good as username and password. The number one and zero button doesn't map to any letter, as can be seen in figure 22.

Figure 22: No letters maps to the 0 and 1 button on a cell phone's keypad.

The 0's and 1's will therefore only be included in the PIN code if they are part of the original phrase. This will result in smaller entropy, containing digits only from 2 to 9, omitting 0 and 1. The result from equation 4.3 shows that there are 23 times more combinations in 14 digit PIN code containing the letters from 0-9, than in the case with the digits from 2-9, and the bit space will decrease in a value from 47 (B1) to 42 (B2), see equation 4.4.

$$\begin{aligned} 8^{14} * x &= 10^{14} \\ x &= \frac{10^{14}}{8^{14}} \\ x &\approx 23 \end{aligned} \tag{4.3}$$

$$\begin{aligned} B1 &= \log_2(10^{14}) \\ &\approx 47 \\ B2 &= \log_2(8^{14}) \\ &= 42 \end{aligned} \tag{4.4}$$

Since the passphrase mechanism can be used in both the username and password approach and in the PIN code approach, to make the memorizing task easier, a username and password approach is desirable. Even if this will be at the sacrifice of the usability, it will give a much higher entropy and in that way make it more secure. The original way to use pass phrases is to generate the password out of the hole sentence (including every word and whitespace) [67]. This will be really cumbersome on a cell phone's keypad. The usage of pass phrases are therefore in this setting used to generate long and easy to remember passwords.

4.2.2 Passface Authentication

Another authentication approach is to make use of passfaces [11, 4, 5]. This is a method which prompts the user with grids of different faces. If a user picks the right faces in

right order, he or she will gain access to the system. See section 3.2 for more information about the passface mechanism implemented in the prototype.

The passface authentication mechanism is a nice feature on a device with a keyboard with only a few buttons. But how secure is this mechanism compared to a regular password or PIN code (Personal Identification Number)?

As explained in section 3.2, the passface mechanism contains 4 different 4 X 3 matrices, in which the correct faces must be chosen. This means you have to choose among 12 different faces four times a row. The keyspace is then $12^4 = 20736$.

This keyspace means that there are 20736 different combinations of the 12 faces in the 4 matrices. If we then compare this amount with a 4 digit PIN code, containing 10000 combinations, we have a stronger authentication mechanism. A 4 digit PIN code will namely have a keyspace of $10^4 = 10000$

We can then find the bit-space these two equations require. The bit-space is a description of how many bits required to handle the combinations digitally, or the key space changed into a binary format (0's and 1's). The bit-space is often used for comparison of different security approaches and must be calculated to find out the robustness against brute force attacks. The bit-space for handling the passface combinations will be 14.34. Since computers are operating with only 0's and 1's, this means that it is required 15 bits to handle this number binary, see equation 4.5.

$$\begin{aligned} bit-space &= \log_2(keyspace) \\ &= \log_2(20736) \\ &= 14.34 \end{aligned} \tag{4.5}$$

This passface method requires 1 bit extra to handle the different combinations, than with the 4 digit PIN approach. The bit-space of a 4 digit PIN is namely 13.29, see equation 4.6.

$$\begin{aligned} bit-space &= \log_2(keyspace) \\ &= \log_2(10000) \\ &= 13.29 \end{aligned} \tag{4.6}$$

As the equations describe, the keyspace and bit-space of the passface authentication mechanism is greater than a 4 digit PIN code. The passface approach is as strong as a PIN code containing 4.32 digits, see equation 4.7. This means that you need a PIN code with more than 4 digits to have the same protections as the passface mechanism gives.

$$\begin{aligned} \#\ digits &= \frac{\log_2(20736)}{\log_2(10)} \\ &= 4.32 \end{aligned} \tag{4.7}$$

The reason for comparing the passface mechanism with a 4 digit PIN code, is that this is a very common authentication mechanism in cell phones. Even if this PIN code gives a relatively poor bit-space, it is combined with other mechanisms which make it strong. It is usually only 3 wrong attempts before the authentication procedure is locked.

When talking about a 4 digit PIN code and the passface mechanism, the best authentication solution will therefore be to make use of passfaces.

A 4 digit master PIN code can, as discussed in section 4.3, fast and easy be broken by a brute-force Stealing MIDlet. A very strong password should therefore be chosen, to make this task as hard as possible.

4.2.3 Putting It All Together

Since there are many different ways to implement the master password, an overview of all the methods discussed are collected in a single diagram, see figure 23.

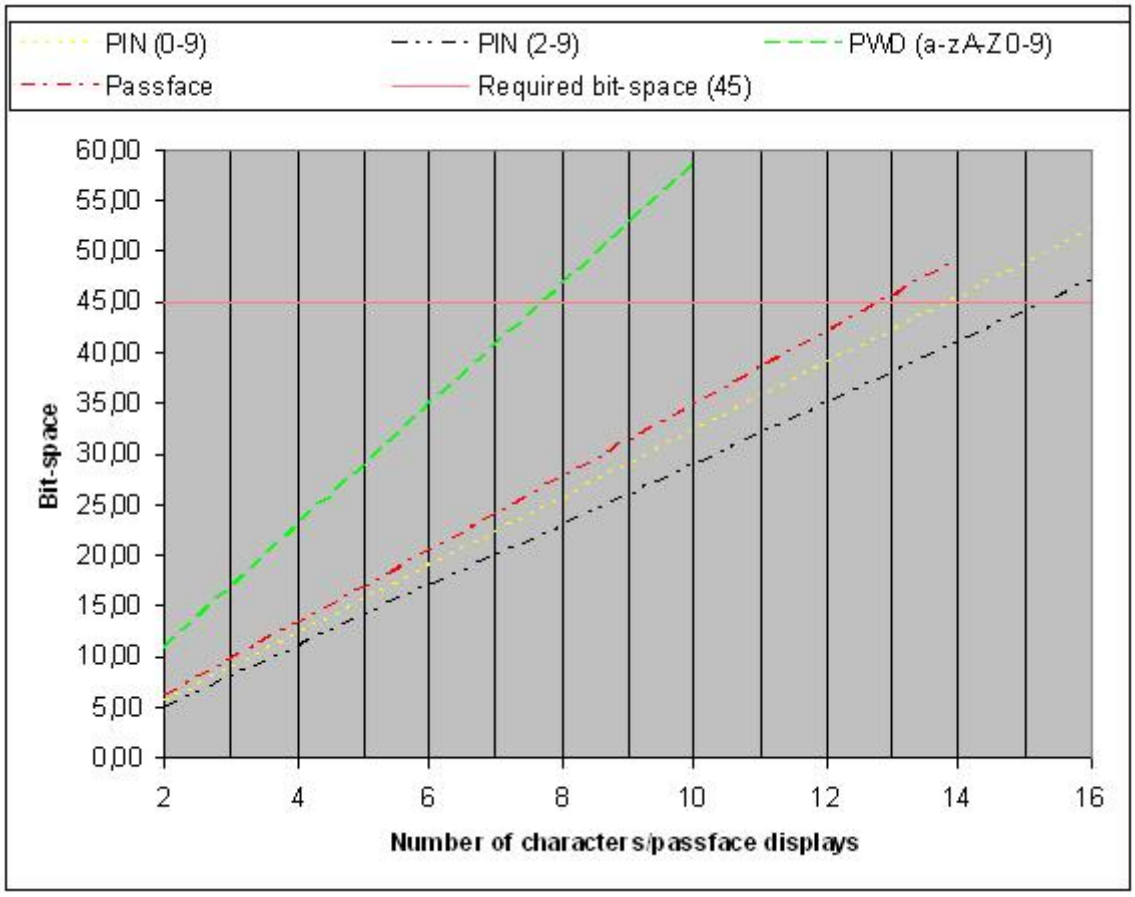


Figure 23: Comparison of different master password configurations.

For each of the password configurations, you can see the relationship between the bit-space and the number of characters. For example will a PIN code with digits from 0 to 9 require a password with 14 characters, to reach a bit-space of 45.

Since we are dealing with passwords stored in a binary format containing only 0's and 1's, we will make use of the bit-space as a comparator. A brute force attack will in the worst case need to test every possible combinations of this 45 bits word, to be 100 percent sure to find the right one. A 44 bit word would on the other hand be the average space needed to go through to break the password, since one extra bit in a binary amount will double it.

A bit-space of 45 will require an attack time of more than 15 years, with basis in the

Sony Ericsson k700i's guessing rate, see figure 24. The guessing rate of the Sony Ericsson k700i and Nokia 6230i is a result from the brute force test discussed in section 4.3.1.

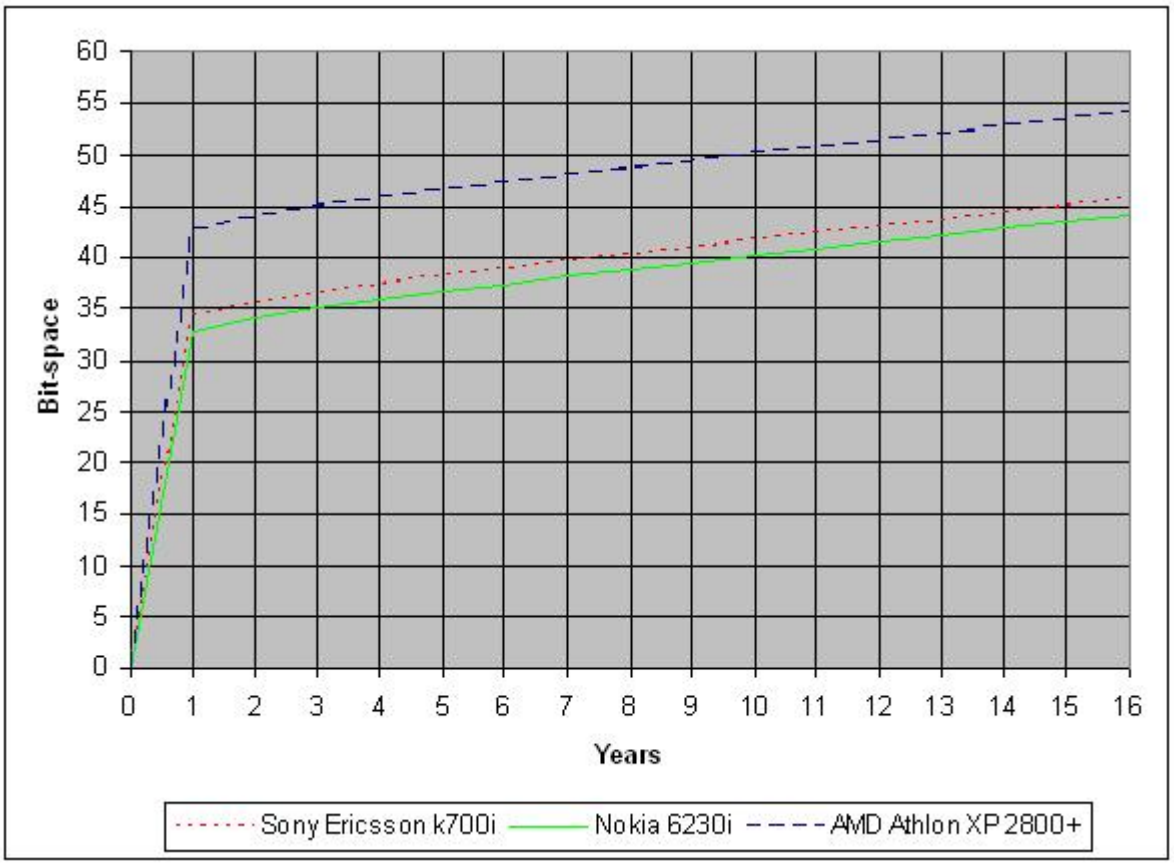


Figure 24: An illustration of how many years needed to break a password of different bit-space. Moore's Law taken into consideration.

It is also important to take a look at the speed of regular computers. Like explained in section 4.4, the SHA1 digest may easily be downloaded to a computer where the brute-force attack may be performed offline. It is stated in [68, s.11], that an AMD Athlon XP 2800+ may perform 212 thousand SHA1 guesses per second.

Moore's Law has also been taken into consideration when this calculation has been made. Moore's Law states that the processors speed will double every 18'th month. A cell phone with 603 guesses per second will then be able to perform 1520 guesses per second in the end of the second year from now. This is illustrated in equation 4.8, where GS_2 is the number of guesses after 2 years. The time in years is here divided with 1.5, since 18 months is the same as one and a half year.

$$\begin{aligned} GS_2 &= 603 * 2^{(\frac{\text{Time in years}}{1.5})} \\ &= 603 * 2^{(\frac{2}{1.5})} \\ &= 603 * 2^{1.33} \\ &= 603 * 2.52 \\ &\approx 1520 \end{aligned} \tag{4.8}$$

The AMD Athlon XP 2800+ processor will as another example increase its guessing rate to 534 thousand guesses within two years, see table 3.

Year	AMD Athlon XP 2800+	Sony Ericsson k700i	Nokia 6230i
0	212000	603	224
1	336529	957	356
2	534207	1520	565
3	848000	2412	896
4	1346116	3829	1422
5	2136826	6078	2258
6	3392000	9648	3584
7	5384464	15315	5689
8	8547304	24311	9031
9	13568000	38592	14336
10	21537858	61261	22757

Table 3: Increased guessing rate per year, according to Moore's Law.

As discussed trough this section, there are different password configurations available. There are also different methods to break these passwords.

Even if the password is theoretically breakable, it should not be breakable in practice. In other words, the password should be strong enough to resist an attack over the period of time the information it protects is sensitive. Let's say a cell phone containing the password to a mail account is lost. If an attack on the master password would require 1 year to succeed, it will be plenty of time to generate a new fresh password to this mail account.

The problem is that a copy of the SHA1 hash can be downloaded to a hacker computer with help of phone managers. An attack can then be performed on the hacker's computer offline. Account lockout and increasing wait time when a wrong password attempt has been made, is therefore of no value.

A master password must therefore be able to resist an attack over the period of time the user updates his master password and the login credentials stored in the phone. If a user updates the master password and the stored login credentials with a frequency of once per year, the password or encryption must resist an attack for a time way longer than this period to be on the safest side.

According to the calculations made through this chapter, the best approach would be to make use of a pass phrase approach on a username and password configuration. This would lead to a strong protection. But as mentioned, it is cumbersome to enter long sentences on a cell phone. A good trade-off between security and usability would therefore be to make use of a single password with upper and lower-case letters and digits (no username), even if this will lead to an easier eavesdropping, since the last

entered character must be visible. A pass phrase approach can further be used to make the memorizing task easier.

The chosen password should be more than 8 characters, to prevent an easy successful brute-force attack. 8 characters will namely require a brute-force attack over a time period of 7 years with basis in an AMD Athlon XP 2800+, and more than 19 years with basis in the speed of a Sony Ericsson k700i. The user should also change his login credentials regularly, and at least once a year.

The prototype should therefore be implemented again, and add a password instead of a PIN code in the master authentication procedure. This password should then be at least 8 characters, and contain upper and lower-case letters and digits.

A multi factor authentication procedure with a PIN and speech recognition would also be a good solution. Another approach which is not discussed is to store half of the SHA1 master digest in the cell phone's RMS, and the other half on the Bluetooth device. An approach like this will require booth the cell phone and the Bluetooth device, in order to break the password. More information about this can be read in chapter 6, regarding further work.

4.2.4 Forgotten Master Passwords

Forgotten master passwords is a problem with password protected applications, and there exist no good solution to forgotten master passwords [69].

There does on the other hand exist some methods trying to handle this problem. Cognitive and associative passwords are one such method [70]. This can be done by identifying different questions you may be asked to answer, if you forget the real master password. But it is important to remember that this is a master password, which lead to sensitive information, and therefore it should be handled with care. Different question like this can often easily be identified by a hacker, and may be easier to procure than the real master password.

You may also be e-mailed a new generated password, but this is not a good solution for a MIDlet. Helpdesk will not be suitable either.

The best solution may be to let the user enter a new master password, but by doing this all of the information stored in the application must be deleted. This is a way to enter the application again, but you have to do the hard work of reentering the deleted login credentials (this may be a problem, since the user most likely has forgotten them). This is the method which the password manager in the web browsers from Mozilla (Firefox) are using [71]. Also in Kwallet [72] and many other password manager software that are available (Password Gorilla [73], KeePass Password Safe [74], Oubliette [75] etc.), the only solution to forgotten master passwords is to delete the secured password store, and reenter a new master password.

4.3 The Stealing MIDlet

An important point about RMS storage is that the stored information easily can be accessed. With a little bit of programming, you are namely able to install a MIDlet able to read everything stored in the RMS by another MIDlet. Source code of the stealing MIDlet can be found in the appendix D.

This RMS extraction is straight forward, and the Password Store prototype's master password is therefore almost worthless, since this is implemented with the aim of pre-

venting other people to get access to the stored sensitive information.

Figure 25 gives an example of how the Stealing MIDlet may look like. Here will all the record stores used in the Password Store MIDlet be listed. It is even possible to enter the name of the preferred record store, to view the actual contents.

Figure 25: The Stealing MIDlet, listing the Password Store MIDlet's record stores.

To make this Stealing MIDlet work, the Stealing MIDlets JAD file must have some attribute values identical to the values found in the Password Store MIDlet's JAD file. The attributes that must be identical is:

- MIDlet-Name
- MIDlet-Vendor

A hacker will therefore only need to rebuild the Stealing MIDlet with these new and identical values to be able to extract information from the RMS.

An installation of the Stealing MIDlet will overwrite the stored Password Store MIDlet (the Password Store MIDlet will therefore not be able to start again). Even if the Password Store MIDlet is overwritten, the content of the RMS will be intact and open for the Stealing MIDlet to read. This procedure is described as an "MIDlet Suite Update" in the MIDP specification [21, p. 15].

> A MIDlet suite update is defined as the operation of installing a specific MIDlet suite when that same MIDlet suite (either the same version or a different version) is already installed on the device.

To make this technique work the hacker must:

- Be in possession of the cell phone with the Password Store MIDlet installed.
- Know what the Password Store MIDlet's JAD file looks like (name, vendor).
- Implement a Stealing MIDlet, or use the one developed together with this thesis.
- Update the Stealing MIDlet's JAD file, so that it is almost identical to the Password Store MIDlet's JAD file.

- Be able to install the stealing MIDlet on the cell phone.

The content of the JAD files from the Password Store and Stealing MIDlet can be seen in table 4 and 5 below. With these JAD files, the Stealing MIDlet may extract RMS information from the Password Store MIDlet.

MIDlet-1: PasswordStore„PasswordStore
MIDlet-Jar-Size: 102217
MIDlet-Jar-URL: PasswordStore.jar
MIDlet-Name: Password Store
MIDlet-Permissions: javax.microedition.io.PushRegistry,
javax.wireless.messaging.sms.receive, javax.microedition.io.Connector.sms
MIDlet-Push-1: sms://:3536,PasswordStore,*
MIDlet-Vendor: Tommy Egeberg
MIDlet-Version: 2.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
SMS-Port: 3536

Table 4: The Password Store MIDlet's JAD file.

MIDlet-1: StealingMIDlet„StealingMIDlet
MIDlet-Jar-Size: 4743
MIDlet-Jar-URL: StealingMIDlet.jar
MIDlet-Name: Password Store
MIDlet-Vendor: Tommy Egeberg
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0

Table 5: The Stealing MIDlet's JAD file.

Since a RMS extraction is that easy, a proper encryption will therefore be especially important. But how can this be done if the stored information can be extracted?

The master password SHA1 value can, as just mentioned, be extracted by running the Stealing MIDlet. This value can then be used together with a pre-created dictionary of SHA1 values and in that way find the real master password. Table 6 gives an example of how a SHA1 dictionary may look like.

If you have got the master password SHA1 value, a search through the dictionary will give the right password if an identical SHA1 value can be found. An example of the Stealing MIDlet reading the master password SHA1 value can be seen in figure 26.

As can be seen in figure 26, the SHA1 value of the master password used in an installation of the Password Store MIDlet maps to the seventh row of the SHA1 dictionary, see table 6. This means that the used master password is 12345677.

This SHA1 value can also be extracted by other methods like explained in chapter 2.2, but this is way more difficult than an installation of the stealing MIDlet. Another way to get hand on the master password's SHA1 value is discussed in section 4.4.

This test was also performed on the Opera Mini web browser [76]. This is a Java application aimed for mobile phones. When you install this MIDlet you will have no idea of what kind of information that it stores in the RMS. But by updating the Stealing

Figure 26: The Stealing MIDlet reading the master password SHA1 value.

Row	Password	SHA1 value
1	12345671	".ðŸ:- ^ Eåñrh.õ9 1>Ÿ*
2	12345672	äj$x¿yž!ç'Å.ØIüÃ£pîÜ
3	12345673	^ ËM#-óiB‹enÁæi,å-Šôe
4	12345674	xO.rioYÃžMZïÂ.bwfBui
5	12345675	rÅ?Äæwë+Êv3ô'åü!Â6y*
6	12345676	;8qWø.aÚěwô'o{#=m£dl
7	12345677	•yJ¯Üajf>Èîvu|9'▮:£
8	12345678	¼|" '}.2øŠòpY!4è3ccë|
9	12345679	Ä15i¶j.+ñpzB?Å½zßsÊf
10	12345670	ž?<Ä+Óp*>í.y=VÎÂ% på

Table 6: Example of a SHA1 dictionary.

MIDlet with the name and vendor information identical to the ones of Opera Mini, this is easily extracted. The result from this was not surprisingly encrypted data, in two different record stores named h and s.

4.3.1 A Brute-Force Stealing MIDlet

Another usage of the Stealing MIDlet may be when the hacker is in possession of the Password Store MIDlet's source code. If this is the case, he can update the source code, to bypass the master password authentication by for example enter a master password of only ones. The MIDlet may then be given a new functionality, which does a brute force attack on the real master password.

If the master password is a 4 digit PIN code or a passface combination, the brute force task is fast and easy. As can be seen below, a 4 digit master PIN can in average be found in 8 seconds, and this is far from acceptable for protection of sensitive information.

This brute force attack on the master password is set up by trying every possible value from 0000 to 9999. A SHA1 value has been generated from every value and compared to the real master password stored in the RMS. The example here is made for demonstration purpose. The password may be found even faster if the brute force algorithm had been implemented in a more effective way.

A random PIN has been calculated 30 times. The brute force attack has then been run

on each of this generated PIN codes 30 times. The average of each of this 30 tries is then generated, giving us 30 different run time averages. An average of guesses per second is also computed from these runtime averages, which gives us the possibility to compute the average time needed to attack a 8 digit PIN code. The guesses per second has also been used to calculate the time needed to find the password of other configurations like username and password, passface and 14 digit PIN codes.

The cell phones internal clock has been used, and this may include some bias. The MIDP's random function has been used in the generation of the 30 different PIN codes. This may also contain some bias, since this function will generate the same sequence of numbers if the same seed has been used [77].

```
System.currentTimeMillis()
Random.nextInt(10)
```

The test has been run on a Sony Ericsson K700i and a Nokia 6230i. The results can be seen in table 7 and 8 below.

Number	PIN code	Time ms	guesses per second
1	5123	8442	606.85
2	3308	5526	598.63
3	3256	5413	601.52
4	1057	1764	599.21
5	1394	2318	601.38
6	4635	7646	606.20
7	9439	15725	600.25
8	2032	3351	606.39
9	2876	4753	605.09
10	3708	6148	603.12
11	9131	15223	599.82
12	4046	6702	603.70
13	7124	11742	606.71
14	2492	4164	598.46
15	9697	16049	604.21
16	1627	2669	609.60
17	4336	7141	607.20
18	9232	15252	605.30
19	7550	12591	599.63
20	0679	1120	606.25
21	7710	12757	604.37
22	0836	1395	599.28
23	4123	6805	605.88
24	5476	9051	605.02
25	7278	12027	605.14
26	2281	3790	601.85
27	2893	4810	601.46
28	0059	99	595.96
29	4550	7537	603.69
30	1966	3269	601.41
SUM	≈ 4330	≈ 7175	≈ 603.48

Table 7: Results from brute force attack on k700i. 4 digit PIN code, R=603.

The results from the SonyEricsson device gives us a rate of approximately 603 guesses per second, and the PIN code is found in an average time of 7.18 seconds. The Nokia device is a great deal slower, with approximately 224 guesses per second and a cracked PIN code within an average time of 19.58 seconds.

Number	PIN code	Time ms	guesses per second
1	7248	32167	225.32
2	5987	26552	225.48
3	5012	22288	225.48
4	9279	41316	224.59
5	4034	17931	224.97
6	7394	32843	225.13
7	1266	5673	223.16
8	2719	12200	222.87
9	6631	29743	222.94
10	6441	28832	223.40
11	0649	2909	223.10
12	0513	2302	222.85
13	5550	24864	223.21
14	6925	30982	223.52
15	4095	18306	223.70
16	1042	4678	222.75
17	3052	13678	223.13
18	7665	34038	225.19
19	0262	1168	224.32
20	2419	10768	224.647
21	3274	14554	224.96
22	8917	39752	224.32
23	7894	35132	224.70
24	1822	8093	225.13
25	2767	12321	224.58
26	0807	3594	224.54
27	8451	37627	224.60
28	5267	23436	224.74
29	0339	1512	224.21
30	4108	18263	224.94
SUM	$\approx$ 4394	$\approx$ 19584	$\approx$ 224.37

Table 8: Results from brute force attack on 6230i. 4 digit PIN code, R=224.

With this information can also the average attack time of brute forcing an 8 digit PIN code be calculated. In following calculations will the guesses per second rate from the Sony Ericsson k700i device be used, since this was the fastest one in the test.

An 8 digit PIN code can have 10^8 different values (S), and the average attack space (V) will therefore be approximately 26, as illustrated in equation 4.9. This means that an attacker on average has to do go through 2^{26} values with a likelihood of 100% (L) to find the base secret.

$$\begin{aligned} V &= \log_2(\frac{S}{2*L}) \\ &= \log_2(\frac{10^8}{2*1}) \\ &= \log_2(5*10^7) \\ &= 25.58 \\ &\approx 26 \end{aligned} \tag{4.9}$$

The average attack time (T) of the 8 digit PIN code will then be approximately 31 hours, as calculated in equation 4.10. Guesses per second (R=603) comes as mentioned from the 4 digit brute force test taken on a Sony Ericsson k700i phone.

$$\begin{aligned} T &= \frac{2^V}{R} \\ &= \frac{2^{26}}{603} \\ &\approx 111292\ sec \\ &\approx 31\ hours \end{aligned} \tag{4.10}$$

As can seen from these calculations will a 8 digit PIN code be far from good. But with only 2 more digits in the PIN code (10), the average attack time will increase to 96 days, and approximately 116 years when we make use of the passphrase mechanism discussed in section 4.2, see equation 4.11.

$$\begin{aligned} V &= \log_2(\frac{S}{2*L}) \\ &= \log_2(\frac{8^{14}}{2*1}) \\ &= \log_2(2.2*10^{12}) \\ &= 41 \end{aligned}$$

$$\begin{aligned} T &= \frac{2^V}{R} \\ &= \frac{2^{41}}{603} \\ &\approx 3.65*10^9\ sec \\ &\approx 116\ years \end{aligned} \tag{4.11}$$

It is also conceivable that new phones will get faster than this in the future, like described in Moore's Law [78].

Despite the Sony Ericsson phone has almost three times more guesses per second than the Nokia phone, it doesn't affect the number of characters needed in the master password very much. Let's say you are comfortable with a master password able to resist an attack for 1 year, since the stored login credentials should be updated within this time period. If we then compare this time requirement with the number of characters needed in a master password, we get the results illustrated in the diagram in figure 27.

We know that the bit-space is calculated by $V = \log_2(keyspace)$; the number of combinations represented binary. The bit-space may also be represented with time (T) and the guessing rate (G/Sec), see equation 4.12.

$$\begin{aligned} T &= \frac{2^V}{G/Sec} \\ T * G/Sec &= 2^V \\ \log(T * G/Sec) &= V * \log(2) \\ V &= \frac{\log(T * G/Sec)}{\log(2)} \\ V &= \log_2(T * G/Sec) \end{aligned} \tag{4.12}$$

If we combine the previous two bit-space functions, we get the result shown in equation 4.13, where A is the number of different characters in the alphabet and x the number of characters needed to resist an attack for T seconds.

$$\begin{aligned} \log_2(A^x) &= \log_2(T * G/Sec) \\ \frac{x * \log(A)}{\log(2)} &= \frac{\log(T * G/Sec)}{\log(2)} \\ x * \log(A) &= \log(T * G/Sec) \\ x &= \frac{\log(T * G/Sec)}{\log(A)} \\ x &= \frac{\log(2^V)}{\log(A)} \end{aligned} \tag{4.13}$$

If we also are going to take Moore's Law into consideration we need to fix the guesses per second accordingly. We know that the G/Sec will increase with a factor of $2^{\frac{Y}{1.5}}$ where Y is the time in years, see equation 4.8 for an example. But we can not use this factor directly, since the guessing rate will increase gradually. For example will a guessing rate of 100 increase to 200 after 18 months and end up on a rate of 252 after two years. We will therefore get a wrong solution if we use the rate of 252 to compute the bit-space over the hole period. To simplify, we will update the guessing rate once a year, and we get a bit-space $V = \log_2(G/Sec * Sec_{year} * (\sum_{i=1}^{Y} 2^{\frac{i-1}{1.5}}))$, where we add every year's brute-force tries together. This V is then used in equation 4.14 which is used to generate diagram 27.

$$\begin{aligned} x &= \frac{\log(2^V)}{\log(A)} \\ x &= \frac{\log(2^{\log_2(G/Sec * Sec_{year} * (\sum_{i=1}^{Y} 2^{\frac{i-1}{1.5}}))})}{\log(A)} \\ x &= \frac{\log(G/Sec * Sec_{year} * (\sum_{i=1}^{Y} 2^{\frac{i-1}{1.5}}))}{\log(A)} \end{aligned} \tag{4.14}$$

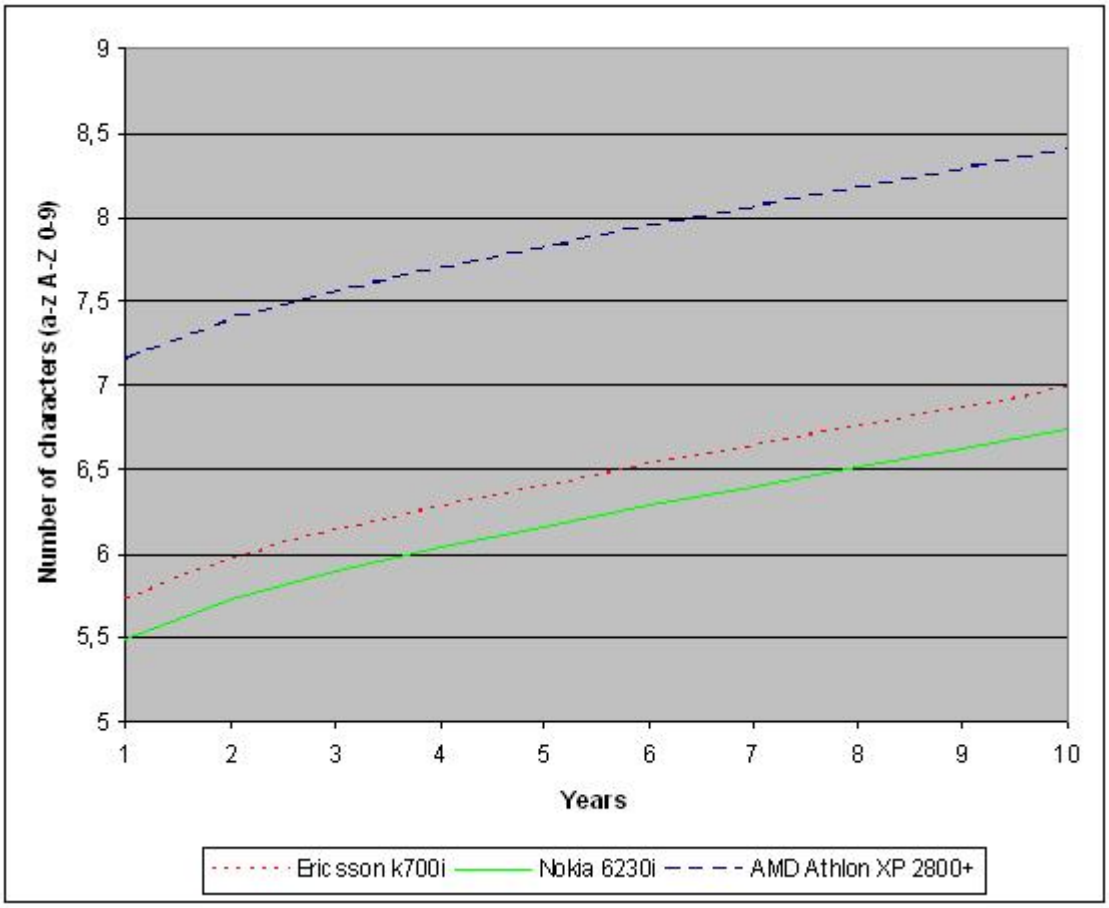


Figure 27: Number of characters needed to resist a brute-force attack from 1 to 10 years.

As can be seen in the diagram above, will a 6 characters long password (a-z A-Z 0-9) resist an attack from both the Nokia and SonyEricsson phone the first two years. A 8 characters long password will on the other hand be needed to resist an offline attack by an AMD Athlon XP 2800+ computer for more than one year.

4.4 Cell Phone Managers

It is like stated, important to encrypt the stored sensitive information. There are namely different ways to extract the stored information. A Stealing MIDlet can easily be installed, and be able to read the information stored in the MIDlet's RMS. Even deleted information can be extracted with forensic methods. In any case, a way more effective method will be with help of a phone manager installed on your computer. This software can then be used to take a backup of a cell phone.

Software like Oxygen Phone Manager II [51] gives you the opportunity to backup your phone like this. In the newest release of this software you are also able to make a backup of your installed Java applications, including the content of the RMS, see figure 28. The backup file can then easily be examined with for example a hex-editor, and the content of the RMS can in this way be extracted.

A possible hacker will therefore only need a couple of minutes alone with your phone, to take a backup of it. Let's say a user leaves his phone on a desk when he goes to the

Figure 28: With a phone manager you may be able to do a backup of your Java MIDlets, including the content of the RMS.

bathroom. A hacker with a phone manager installed on his laptop may then easily take a fast backup of the installed Password Store MIDlet. He will then be able to extract the master password SHA1 value or the stored and encrypted login credentials. The hacker can then start a offline brute-force attack on his own computer. When the user comes back from the bathroom he notices nothing, while the hacker tries to break the master password. If the hacker then manages to break the password, he can later steal the phone. With the right master password he will then have access to all the stored login credentials.

If the right master password has been found, it can also be used to decrypt the stored login credentials. The stored login credentials will then be in the hands of the hacker without the need to access the cell phone more than once.

The problem discussed here is that a hacker may perform an attack on your master password, and in a way you don't even notice. The master password should therefore be strong enough to resist an attack over a period of time, in which you generally update or change it. An attack may also be performed on the stored login credentials, and they should therefore also frequently be updated and as strong as possible.

The time-line in figure 29, illustrates how this attack may elapse. A master password must therefore be able to resist an attack over the period the brute-force attack is performed. The login credentials should therefore be updated on a regular basis, to prevent this attack to succeed. An update will make the hacker's backup file useless, and he must start the attack all over again.

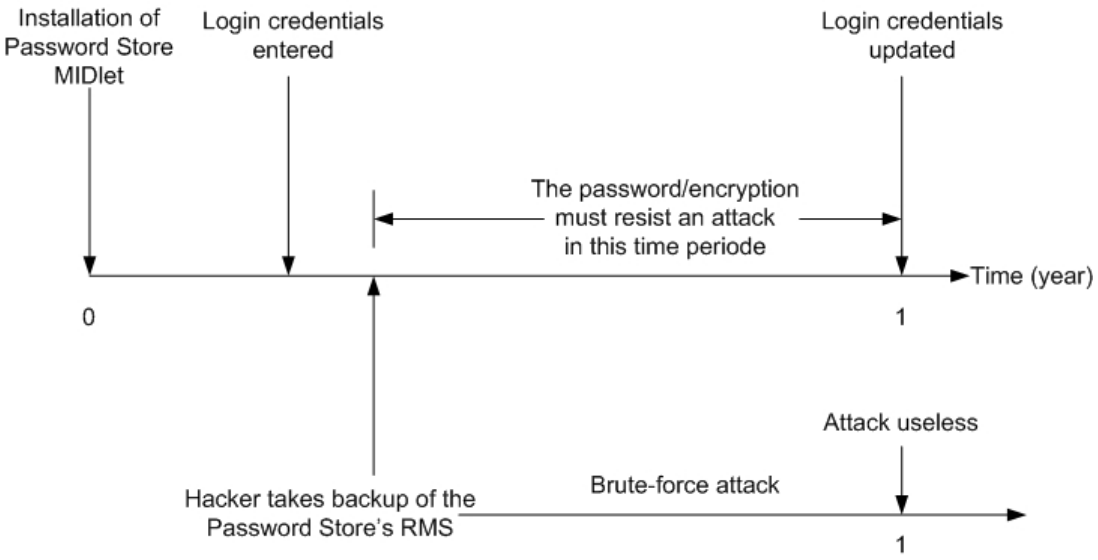


Figure 29: A time-line representation of how long the master password and login credentials must resist an attack

In such an attack, it is the hacker's computer's speed that is the interesting part. As discussed in section 4.2, a cell phone will be able to guess 603 passwords per second. A regular computer is way faster, and may perform 212 thousand SHA1 guesses per second. Since a regular computer is the fastest one, must this be put into consideration when a password length and type shall be recommended.

5 Discussions

This chapter will discuss the results from the security analysis.

5.1 The J2ME Platform

The J2ME platform does have a lot of different security mechanisms implemented. They all aim for an increased robustness, but it is spared the reader to take a closer look at the chapter regarding related work 2, if this is of interest.

Also encryption is possible to perform within a Java MIDlet. External API's will anyway be necessary, since the MIDP doesn't have any cryptographic support. The API from Bouncy Castle [36] was chosen in the implementation of the thesis' prototype. Today's cell phones has further no problem to handle encryption, because of their frequently increased computationally capabilities. A Sony Ericsson k700i (cell phone) is namely able to perform approximately 600 SHA1 computations per second, as the brute-force test in the analysis showed us. With Moore's Law [78] in mind will the next generation's cell phones will be a lot more powerful.

5.2 Passwords

The sensitive information (login credentials) stored in the cell phone are secured by a master password and encryption. This should prevent leakage of the information, even if the phone is lost or stolen.

The thesis has analyzed different password configurations that can be used to prevent unauthorized people to get access to a MIDlet. The result from this analysis shows us that a relative strong password needs to be implemented to resist an attack. The stored SHA1 hash digest can namely easily be downloaded to a computer, where a brute-force attack can be performed. A Stealing MIDlet may also be implemented with the aim of extracting the information stored in the cell phone's RMS. This Stealing MIDlet may also perform a brute-force attack, but way more ineffective than on a regular computer.

The results from the analysis showed that a bit space of 42 would be found within the first year by a brute-force attack on a regular computer. This means that a seven characters long password from a alphabet of 62 characters (a-z A-Z 0-9) would be broken within a year ($V = \log_2(62^7) \approx 42$). On a cell phone would a password with a bit-space smaller than 32 be broken, which is the same as a password with five or less characters ($V = \log_2(62^5) \approx 30$) from the same keyspace.

The analysis did also give an overview of the most optimal password configuration. The passface configuration was better than a 4 digit PIN code. In any case, both of them could easily be cracked with a brute-forced attack, with help of a Stealing MIDlet or a phone manager which takes a backup of the RMS. Because of this fact, would a lock down of the Password Store MIDlet after some wrong attempt not be very useful. This would not be a very user friendly approach either. The passface approach would anyway need to include 13 passface matrices at row, to be able to resist a brute-force attack for more than one year. This will be hard to memorize, and is undoubtedly not user friendly.

The analysis also tried to take a closer look both at a username and password approach, password and PIN, a single PIN code and a single password.

The results showed us that the best approach was to configure the master password as a single password, with upper and lower-case letters, digits and some special characters. This approach will anyway make the password vulnerable to eavesdropping. The last entered character will namely need to be visible on the display as a consequence of the fact that the user has to multi-tap a button to pick the right character (because of the limited number of buttons on the keypad). This solution will further be a good trade-off between security and usability, since a relative strong password can be chosen with as few user interactions as possible.

A passphrase approach can then be used to make the password memorizing task easier. This approach could be used to generate both long PIN codes (only digits) and passwords which also contain non-digit characters. A passphrases approach used to generate long PIN codes would anyway result in lower entropy, since no letter maps to the number one and zero buttons on the keypad. One and zero would then need to be expressed clearly in the passphrase itself, to be included in the generated PIN code.

Biometric authentication like voice recognition could also be a way to secure the application. This will however be a topic for further work, because the lack of support in todays cell phones. The possibility of storing parts of the master password SHA1 digest on the Bluetooth device used in the working solution will also be a topic for further work. This could namely increase the robustness of the application, but no analysis has proved this yet.

An increasing wait-time when a wrong master password is entered is also implemented in the prototype. This will increases the attack time radically if a possible brute-force attack is executed on the cell phone. The problem is that this mechanism is easily bypassed when for example the MIDlet's RMS is downloaded to a computer and the attack is performed offline. In other words, this mechanism will only make it harder for a hacker with little or no knowledge about the other possible data extraction methods.

A forgotten master password is a difficult topic. There exist no good solutions to this problem. One method will anyway be to reset the application. This means that all the stored information is deleted, and the user must enter a new fresh master password. This is not a very user friendly approach, but it is the safest way to hinder unauthorized access.

5.3 Information Stored In RMS

The goal of the project was to find out if information stored in a Java enabled phone could be secured properly. Java MIDlets make use of the RMS to store information for later use.

Information stored in the RMS was easily extracted by a Stealing MIDlet. Results from previously work, also provide methods that can be used to extract deleted information.

Even software installed on a computer is able to read the information stored in the RMS. With some phone managers you will have the opportunity to take a backup of your cell phones Java applications, including the information in the RMS. This backup file can then be examined with a hex editor. Information stored in the RMS is therefore easily available, and proper encryption is therefore very important.

The master password must also be stored in this RMS. A one way hash function

(SHA1) is therefore used to prevent a hacker to easily get his hands on it. This master password is also used to generate a key used to encrypt the stored login credentials (the MIDlet's sensitive information). If a hacker therefore manage to take a backup of the cell phone of interest, this can be disastrous. If he successfully breaks the master password with a brute-force attack, he will namely also have the key to decrypt the stored sensitive information. And all this can be done without the user even noticing it, assuming the hacker is able to do a backup of the phone without being detected.

The result from this is that the master password needs to be strong enough to resist an attack over the period of time the user change the passwords stored in the phone. If a eight characters long password from a alphabet of 62 characters has been used, it will take more than eight years to complete a brute-force attack on a regular computer. But this could be decreased radically if a cluster of computers are cooperating. This has not been taken account for in this analysis. Also this is the worst case scenario. A more effective attack may be performed with other algorithms or a dictionary search. The stored login credentials should anyway be changed on a frequent basis, and at least once a year.

As mentioned in the previous section, the SHA1 digest may be harder to obtain from a hackers stand point if a part of the digest could be stored on the Bluetooth device used in the complete SSO solution.

5.4 Threats and Countermeasures

Flow diagrams were used to find out what kind of threats a cell phone was vulnerable to. These threats were then used to find effective countermeasures that could be used to prevent them from materializing.

A lot of threats where discovered during the process of setting up these data flow diagrams. A hacker will namely be able to place himself anywhere, where he can inject, interrupt or sniff information. Different hackers may even cooperate to reach their goals, which in this situation will be to get their hands on the sensitive information stored in the phone.

If a hacker has been able to install a trojan software on the phone before an user even buys it, this could lead to disastrous consequences. He will then be able to sniff every user interaction with the phone, even the entered master password and the sensitive login credentials. A user should therefore re-flash the cell phone with a OS that is known to be clean, if very sensitive data are going to be stored in the phone. This re-flashing should be done by the user himself or some trusted source, since another hacker also could place himself at this place.

Trojan software can also be hidden in the Java MIDlet itself. This will be straight forward if the hacker has got the source code. All he has to do is to recompile the MIDlet with the new trojan functionality. This could for example be to send information to the hacker or update the MIDlet to not encrypt the information. This trojan infected MIDlet could then be used to overwrite the clean version installed on the phone. This method would require identical values of the MIDlet-name and vendor attributes in the JAD file. Another approach would be to upload the trojan infected MIDlet to the server. Every new user will then install an infected version of the MIDlet. An installation request could also be re-routed to hacker server where the infected MIDlet is located.

The MIDlet should be signed with a private key, to prevent this kind of hacker actions.

This will give a new user the ability to find out where the MIDlet comes from, and that it is a reliable source. A signed MIDlet will also prevent an infected MIDlet to overwrite itself. You will namely only have permission to update a signed MIDlet if the updating MIDlet has got the same signature [21]. The private key used to sign the MIDlet can also fall into the hands of a hacker. This will give the hacker the ability to sign the infected MIDlet. An infected MIDlet will then be able to appear as a trusted one.

An overview of the discovered threats can be seen in table 9. A low number in the feasibility column means that the threat is relative easy to perform. To steal a phone will for example be easier than installing a keyboard sniffer. Each entry in the "needs" column adds one (or more) point(s) to the feasibility. Some of the threats will also get their "needs" from other rows in the table. These "needs" will have a leading number representing the actual row it is taken from, and it's feasibility in parentheses. The entries in the expertise column have been given a count of 2, because expertise requires a lot of time and work to achieve.

Threat	Expertise(2)	Needs	Feasibility
1-Steal the phone	-	The phone	1
2-Disable remote deletion functionality		The phone	1
3-Extract data from RMS	-	The phone Phone manager	2
4-Implement Stealing MIDlet	Java programming	MIDlet JAD information	3
5-Implement hacker MIDlet	Java programming	Source code	3
6-Update cell phone's OS	-	The phone Flasher utility New OS	3
7-Sniff SMS-Delete SMS	GSM network	Various utilities	3
8-Implement keyboard sniffer	Programming Cell phone brand	-	4
9-Install keyboard sniffer	-	The phone 8-Sniffer(4)	5
10-Extract phone memory	Forensic Cell phone Brand	The phone Forensic utility	6
11-Sniff keystrokes		Reply from sniffer 9-Installed sniffer(5)	6
12-Inject keystrokes		Send keystrokes 9-Installed sniffer(5)	6
13-Reroute MIDlet request to hacker server	GSM network	Hacker server The request 5-Hacker MIDlet(3)	7
14-Reply with hacker MIDlet on MIDlet request	GSM network	Hacker server Send reply(SMS) 5-Hacker MIDlet(3)	7

Table 9: Overview of cell phone vulnerabilities.

A hacker placed on the link between the SMS-Delete MIDlet and the Password Store MIDlet was also discovered by the flow diagrams. As can be seen in the second row in

table 9, this feature can easily be disabled on the Password Store MIDlet. This will be further discussed in the SMS-Delete section below.

5.5 Data Extraction

It turned out that data stored in a cell phone easily can be extracted. Even deleted information can be extracted with different forensic methods. A Stealing Midlet will further easily be able to overwrite an unsigned MIDlet. This Stealing MIDlet can then be used to read the information stored in the original MIDlet's record store. In any case, the easiest way will be to use phone manager software installed on a PC or laptop. A backup of the Java MIDlet's record stores can then easily be performed, and the generated backup file will include everything stored by the MIDlet. This backup procedure can be performed in matter of seconds, and a hacker may therefore be able to do this without detection.

5.6 SMS-Delete

The SMS-Delete MIDlet was implemented with the goal to do remotely deletion of the stored information. This could be a nice feature if the phone is lost or stolen. The idea was to send an SMS with the master password digest to the stolen phone. This SMS should then start the Password Store MIDlet and delete the contents, if the received SHA1 digest was right one.

The problem was that the automatic start procedure of the Password Store MIDlet could easily be deactivated. This will give the user no opportunities to delete the stored information. Since information stored in a cell phone's RMS also is easily available, a hacker will also have the opportunity to remotely delete the information to perform a DoS attack.

The hacker could also place himself between these two applications, and sniff the information sent. He would then get his hands on the SHA1 digest, which in turn could be used in a brute-force attack. If the SMS-Delete function should be a part of the working solution should it therefore be implemented using a separate password, and not the master password. One password, one functionality is an unwritten standard.

6 Further Work

The aim of this thesis has been to investigate how suited a Java MIDlet installed on a cell phone is to keep confidential information secure. This has mainly been done by an implementation of a so called digital safe [1]. This developed MIDlet has then been analyzed thoroughly, to find strengths and weaknesses. A lot of work has been done to find out how the sensitive information is stored, and how it can be extracted.

An analysis around the best way to secure the information has also been carried out. This is however also a topic for further work. Speech recognition has not been implemented in the prototype, mainly because of the fact that most of today's cell phones does not support the Java Speech API (JSR 113) [66]. A MIDlet with speech recognition will therefore be an important part of the future work on the project. It will also be important to perform a thorough analysis of this approach, to find strengths and weaknesses and compare it against the findings in this thesis. Other biometric authentication procedures may also be interesting work, to find the best and strongest way to securely store information in a Java MIDlet.

The Security and Trust Service API (JSR 177) is another topic. Also this is omitted in this thesis, because almost none of today's cell phones have support for this API. If this API can add something positive to the authentication and encryption procedure, will therefore be of importance to find out.

If a regular password is going to be used will a proactive password checking mechanism also be a nice feature. A mechanism like this will probably enhance the user chosen password. A Java implementation tested on a cell phone will therefore be valuable, but it is left for further work to find out if this is manageable on resource constrained devices like cell phones.

Another important part to take a closer look at, is the ability to store information on the Bluetooth device. Since this thesis only has focused on the cell phone, this has been out of scope. If it is possible to store half of the master password digest on the Bluetooth device, it will most likely be much harder for a hacker to break into the MIDlet. A closer look at the ability to store some of the login credential's encrypted bytes would also be of great value. Another topic would therefore be to implement such an approach. An analysis would also here be important, to find out if this approach would increase or decrease the overall security.

It is a fact that it is cumbersome and time consuming to enter text on a cell phone's small keypad. A synchronization mechanisms that could be implemented as a web service could therefore be implemented. With a method like this, a user could for example enter all his login credentials with the keyboard on a regular computer. If the phone containing all the information is broken, could this also be a nice way to restore the lost information into a new phone. A thorough analysis of this would be important to carry out. This will namely require other kind of security functions, both on the web synchronization service and on the MIDlet itself. Public key encryption may be a keyword regarding this topic.

To sum up, work regarding biometric authentication will be a topic for further work. This will include the use of the Java Speech API. The Security and Trust Service API

will be another important part in the authentication procedure, but also in the encryption process of stored sensitive information. To increase the robustness of the MIDlet, would it be nice to store some information on the Bluetooth device. The possibility of this must therefore be investigated further. Last but not least will a closer look at a web synchronization service be a valuable topic to further investigate.

7 Conclusion

The main problem of this thesis was to find out if a Java MIDlet on cellular phone is a secure location for storage of sensitive information. Each of the research questions stated in 1.3 will therefor be discussed shortly below.

Question 1: What is already known about security in Java enabled cell phones?
In chapter 2, we took a closer look at the J2ME platform's available security mechanisms. As could be seen in this chapter, a lot of built in security features exist but it lacks the possibility to perform encryption. An external package will therefore be needed to perform this kind of action, like the one from Bouncy Castle [36]. The Security and Trust Service API (SATSA) [38, 39] will in the future also increase a MIDlet's security level, if cell phones start to support this API.

The J2ME platform is evolving. Flaws will must likely arise, but a fundamental security architecture is already in place.

Question 2: Will information stored on a cellular phone be easy to extract?
There are different techniques to extract data from cell phones. Lot of work have especially been done to come up with proper forensic methods to extract information from cell phones (even deleted information). These methods will require deep technical knowledge.

A Stealing MIDlet may also be implemented, which is a MIDlet with identical name and vendor values in the MIDlet's JAD file. When a Stealing MIDlet is installed it will be treated as an update, and the hole application is replaced with the new one. The information in the RMS will on the other hand retain, and can easily be read.

In any case, the easiest way to extract information from a cell phone will be with help of a phone manager. A backup of the phone can be performed within seconds, including the contents of the RMS.

Data stored in a cell phone can therefore easily be extracted.

Question 3: How can we secure the stored sensitive information even if the cellular phone is lost or stolen?
Some kind of authentication mechanisms must be used to prevent unauthorized access. Passwords and passfaces can easily be implemented in a Java MIDlet. Speech recognition may in the future also be an easy matter, if the cell phones start to support the speech API [66]. The results from the security analysis showed us that the best approach would be to implement a single password.

Encryption will anyway be an essential part of a secure Java MIDlet. The password can then be used as a key in the encryption proses. A lot of different encryption algorithms are available through the use Bouncy Castle's [36] package. The master password used to enter the MIDlet must be chosen with care, since it also is used as the key in the encryption process of the stored login credentials. A minimum length of eight characters,

with upper and lower-case letters and digits is preferable. The login credentials should even be updated frequently, at least once per year.

The SMS-Delete MIDlet, which should be used to remotely delete the stored information, was not very helpful. The service could namely easily be deactivated.

Proper encryption and a carefully chosen master password will therefore be the most important points, when information must remain secure even if it is lost or stolen. The stored login credentials should also be updated on a regular basis.

Question 4: What kind of threats will the cell phone be vulnerable to?
Most of today's PCs are connected to the Internet, and a clever hacker may easily find back-doors or exploitable vulnerabilities to get access to it. A SSO solution installed on a PC may therefore easily be broken into. A mobile SSO solution will on the other hand store all the necessary login credentials in a Java MIDlet on a cellular phone. If an unauthorized user shall be able to get his hands on this information, physical access to the phone itself will be important.

This will generate new barriers for the hacker, but a lot of threats are lurking. A hacker may get access to the phone in many different places, and even add trojan functionality to the SSO MIDlet itself. This can lead disastrous consequences; imagine for example that your login credentials to your Internet bank account are lost.

There are too many threats to be mentioned here. The most important threat may anyway be the easiness of taking a backup of the MIDlet's RMS, and do an offline attack on the content. Trojan code can also easily be added to an unsigned MIDlet. Take a closer look at the data flow diagram in appendix B, to get overall understanding of threats associated with a cell phone.

Question 5: What kind of countermeasures can be used to reduce or eliminate the threats?
Actions can be carried out to prevent some of the threats to materialize. One of these actions is to sign the SSO MIDlet with a secret private key. This will give the user an ability to find out if the MIDlet comes from a trusted source. Signing will also make the MIDlet resistant to "Stealing MIdlet attacks", for as long the private signing key isn't compromised.

Extraction of the stored information can easily be accomplished if the phone is lost or stolen. Proper encryption will therefore be an important keyword, which shouldn't be handled leniently.

Another important countermeasure will be to reinstall the MIDlet after for example a phone repair, which will overwrite possible added trojan code.

A strong master password in combination with frequently updating of the login credentials will however be the best way to hinder breach of confidentiality.

The treats a cell phone may be vulnerable to will most often also be threats a regular SSO solution on a computer is vulnerable to. A signed Java MIDlet will be a secure environment to store the sensitive information if the user takes care about password choice, doesn't leave the phone behind on crowded places and frequently updates the login credentials. The important point is however that the MIDlet, hence the encryption, is able to resist an attack over the period of time the information it protects is sensitive.

Bibliography

[1] Byfuglien, M. A mobile single sign on system. Master's thesis, Gjøvik University College, 2006. http://www.hig.no/imt/index.php?id=230#byfuglien.

[2] Apelt, D. Single sign-on: An overview to integration. White paper, Protocom Development Systems, May 2005.

[3] RSA-Security. July 2005. Single sign-on. putting an end to the password management nightmare. http://whitepapers.zdnet.co.uk/0,39025942,60143602p,00.htm (Visited Nov. 2005).

[4] Dhamija, R. & Perrig, A. D´ej'a vu: A user study using images for authentication. SIMS / CS, University of California Berkeley. http://www.ece.cmu.edu/~adrian/projects/usenix2000/usenix.pdf (visited Nov. 2005).

[5] Sapre, C. Analysis and improvements to graphical passwords. http://chirayusapre.net/works/termpaper.pdf (visited Nov. 2005).

[6] Hayday, G. January 2003. Counting the cost of forgotten passwords. http://news.zdnet.co.uk/business/employment/0,39020648,2128691,00.htm (Visited Nov. 2005).

[7] Clercq, J. D. 2002. Single sign-on architectures. In *InfraSec '02: Proceedings of the International Conference on Infrastructure Security*, 40–58, London, UK. Springer-Verlag.

[8] Leedy, P. D. & Ormrod, J. E. 2005. *Practical Research, planning and design*. Pearson Prentice Hall, 8 edition.

[9] Knudsen, J. 2001. *Wireless Java: Developing with Java 2, Micro Edition*. Apress.

[10] Perelson, S. & Botha, R. An investigation into access control for mobile devices. Department of Business Information Systems (http://www.petech.ac.za/rbotha/Pubs/docs/LC_017.pdf (visited Nov. 2005)).

[11] Brostoff, S. & Sasse, M. A. Are passfaces more usable than passwords? a field trial investigation. Department of Computer Science, University College London, London, WC1E 6BT.

[12] Wikipedia. June 2006. Authentication. http://en.wikipedia.org/wiki/Authentication.

[13] Smartcomputing. July 2004. Handheld terms to know. http://www.smartcomputing.com/editorial/article.asp?guid=\&article=articles/hardware/2004/h0707/handheldterms.asp (Visited Oct. 2005).

[14] Wikipedia. September 2005. Confidentiality. http://en.wikipedia.org/wiki/Confidentiality.

[15] Wikipedia. October 2005. Single sign-on. http://en.wikipedia.org/wiki/Single_sign-on.

[16] SunMicrosystems. February 2005. Cldc hotspot implementation virtual machine. http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf (Visited Feb. 2006). Whitepaper.

[17] SunMicrosystems. 2002. Java 2 platform, micro edition. the java platform for consumer and embedded devices. http://java.sun.com/j2me/j2me-ds.pdf (Visited Jan. 2006).

[18] SunMicrosystems. Java 2 platform, micro edition (j2me); jsr 68 overview. http://java.sun.com/j2me/overview.html (Visited Jan. 2006).

[19] SunMicrosystems. May 2000. J2me building blocks for mobile devices. http://java.sun.com/products/cldc/wp/KVMwp.pdf (Visited Feb. 2006).

[20] SunMicrosystems. December 2000. Mobile information device profile (jsr-37). http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html (Visited Feb. 2006).

[21] SunMicrosystems. November 2002. Mobile information device profile for java 2 micro edition version 2.0. http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html (Visited Jan. 2006).

[22] SunMicrosystems. May 2000. Connected, limited device configuration. specification version 1.0a. http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html (Visited Jan. 2006).

[23] SunMicrosystems. March 2003. Connected limited device configuration. specification version 1.1. http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html (Visited Jan. 2006).

[24] Simonsen, K. I. F. J2me security. Master's thesis, University of Bergen, May 2005. Department of Informatics.

[25] Harkey, D., Appajodu, S., & Larkin, M. 2002. *Wireless Java Programming for Enterprise Applications*. Wiley Publishing, Inc.

[26] Keogh, J. 2003. *The Complete Reference J2ME*. McGraw-Hill/Osborne.

[27] SunMicrosystems. March 2003. Wireless messaging api (wma) for java 2 micro edition version 1.1. http://jcp.org/aboutJava/communityprocess/final/jsr120/index2.html (Visited Feb 2006).

[28] SonyEricssonMobileCommunicationsAB. November 2004. K700i, white paper. http://developer.sonyericsson.com/getDocument.do?docId=65842 (Visited Feb. 2006).

[29] SunMicrosystems. Java apis for bluetooth wireless technology (jabwt); jsr 82. http://jcp.org/en/jsr/detail?id=82 (Visited May 2006).

[30] SunMicrosystems. The k virtual machine. http://java.sun.com/products/cldc/ds/ (Visited Feb. 2006).

[31] Levula, J. Midp 2.0's security architecture. http://sw.nokia.com/id/766a3157-e639-4a64-a464-8becb8c56c03/MIDP_2_0_Security_Architecture_en.pdf (Visited Feb. 2006). Forum Nokia.

[32] Knudsen, J. February 2003. Understanding midp 2.0's security architecture. http://developers.sun.com/techtopics/mobility/midp/articles/permissions/index.html (Visited Dec. 2005).

[33] Debbabi, M., Saleh, M., Talhi, C., & Zhioua, S. Security analysis of wireless java. Concordia Institute for Information Systems Engineering http://www.lib.unb.ca/Texts/PST/2005/pdf/debbabi.pdf (Visited Mar. 2006).

[34] Forum-Nokia. July 2005. Getting started with security. http://www.forum.nokia.com/info/sw.nokia.com/id/2fb09348-acd0-45c1-971f-ccdb626f4218/Getting_Started_With_Security_v1_0_en.pdf.html (Visited Mar. 2006). The theory behind the security models used in application installation in Nokia devices.

[35] SunMicrosystems. 2005. J2se: Security and the java platform. http://java.sun.com/security/ (Visited Dec. 2005).

[36] BouncyCastle. Bouncy castle crypto package. http://www.bouncycastle.org (visited Oct. 2005). Light-weight API, release 1.31.

[37] Tillich, S. & Großschadl, J. A survey of public-key cryptography on j2me-enabled mobile devices. Graz University of Technology, Institute for Applied Information Processing and Communications Inffeldgasse.

[38] Ortiz, E. The security and trust services api for j2me, part 1. http://developers.sun.com/techtopics/mobility/apis/articles/satsa1/ (Visited Feb. 2006), March 2005.

[39] Ortiz, E. The security and trust services api (satsa) for j2me: The security apis. http://developers.sun.com/techtopics/mobility/apis/articles/satsa2/ (Visited Feb. 2006), September 2005.

[40] SunMicrosystems. July 2004. Security and trust services api (satsa). http://jcp.org/aboutJava/communityprocess/final/jsr177/index.html (Visited Feb. 2006).

[41] Forum-Nokia. September 2003. A brief introduction to secure sms messaging in midp. Appendix B.

[42] IAIK. Jce-me. http://jce.iaik.tugraz.at/sic/products/mobile_security (Visited Feb. 2006). Light weight cryptography library.

[43] Corporation, P. T. 2005. Phaos micro foundation. http://www.phaos.com/products/crypto_micro/pmf.html (Visited Dec. 2005). Light weight cryptography library.

[44] NTRUCryptosystemsInc. Ntru neo for java. http://www.ntru.com/products/Neo_Java1.pdf (Visited Feb. 2006). Cryptography on Java devices.

[45] Gupta, R. K. Intelligent appliances and j2me's rms. ITtoolbox. http://java.ittoolbox.com/pub/RG120503.pdf (Visited Feb. 2006), 2003.

[46] Ortiz, E. The midp 2.0 push registry. http://developers.sun.com/techtopics/mobility/midp/articles/pushreg/ (Visited Mar. 2006), January 2003.

[47] Willassen, S. Y. Forensic analysis of mobile phone internal memory. Norwegian University of Science and Technology.

[48] The sleuth kit. http://www.sleuthkit.org (Visited Apr. 2006). Forensic analysis tools.

[49] Ayers, R., Jansen, W., Cilleros, N., & Daniellou, R. Cell phone forensic tools: An overview and analysis. Technical report, National Institute of Standards and Technology, October 2005. http://csrc.nist.gov/publications/nistir/nistir-7250.pdf (Visited Jan. 2006).

[50] Willassen, S. Y. Spring 2003. Forensics and the gsm mobile telephone system. *International Journal of Digital Evidence*, 2, 10–11. http://www.ijde.org/docs/03_spring_art1.pdf (Visited Jan. 2006).

[51] Oxygen-Software. Oxygen phone manager for nokia phones (forensic edition). http://www.opm-2.com/ (Visited Jan. 2006).

[52] Stavik, S. Forensic analysis of the symbian/series 60 smartphone os. Nativ aplication extracting memory content, November 2005.

[53] Eide, J. Analysis of evidence in windows mobile-based smartphones. Forensic appliaction on Windows Mobile smart phones, December 2005.

[54] NetBeans. Java development tools, netbeans ide 5.0. http://www.netbeans.org/downloads/index.html (Visited Feb. 2006).

[55] SunMicrosystems. J2se development kit (jdk) 5.0. http://java.sun.com/j2se/1.5.0/download.jsp (Visited Feb. 2006).

[56] Burrows, J. H. & Prabhakar, A. Secure hash standard. Computer security, National Institute of Standards and Technology, May 1993. Federal Information Processing Standards Publication 180-1 http://www.itl.nist.gov/fipspubs/fip180-1.htm (Visited Jan 2006).

[57] Wikipedia. January 2006. Sha hash functions. http://en.wikipedia.org/wiki/SHA-1 (Visited Jan. 2006).

[58] NIST. November 2001. Advanced encryption standard (aes). http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf (Visited Jan. 2006). Federal Information Processing Standards Publication 197.

[59] Wikipedia. January 2006. Advanced encryption standard. http://en.wikipedia.org/wiki/AES (Visited Jan 2006).

[60] RSA-Laboratories. March 1999. Pkcs 5 v2.0: Password-based cryptography standard.

[61] Itani, W. & Kayssi, A. 2004. J2me application-layer end-to-end security for m-commerce. *Journal of Network and Computer Applications*, 27, 13–32.

[62] Lindquist, T. E., Diarra, M., & Millard, B. R. A java cryptography service provider implementing one-time pad. Electronics and Computer Engineering Technology Arizona State University East, 2004.

[63] Wikipedia. April 2006. Kerckhoffs' law. http://en.wikipedia.org/wiki/Kerchoff%27s_law (Visited May 2006).

[64] Stensvold, T. March 2005. Mms-virus truer. http://www.tu.no/nyheter/ikt/article33068.ece (Visited May 2006).

[65] Halvorsen, F. January 2006. Lumske mobilangrep. *Teknisk Ukeblad*, 2, 34–35. http://www.tu.no/multimedia/archive/00027/Teknisk_Ukeblad_0206_27673a.pdf (Visited May 2006).

[66] SunMicrosystems. Java speech api 2.0 (jsapi) jsr 113. http://www.jcp.org/en/jsr/detail?id=113 (Visited Apr. 2006).

[67] Johansson, J. M. October 2004. The great debates: Pass phrases vs. passwords. part 1-3. http://www.microsoft.com/technet/community/columns/secmgmt/sm1004.mspx (Visited May 2006).

[68] Hills, R. Common vpn security flaws. http://www.netsc.ch/ressources/VPN-Flaws-Whitepaper.pdf (Visited Apr. 2006), January 2005.

[69] SafeSecret. http://www.compactbyte.com/safesecret/ (Visited Nov. 2005).

[70] Bunnell, J., Podd, J., Henderson, R., Napier, R., & Moffat, J. K. 1997. Cognitive, associative and conventional passwords: Recall and guessing rates. *Computers & Security*, 16(7), 645–657.

[71] MozillaZine. November 2005. Master password. http://kb.mozillazine.org/Master_password (Visited Dec. 2005).

[72] Staikos, G. & Watts, L. June 2005. The kwallet handbook. http://docs.kde.org/stable/en/kdeutils/kwallet/ (Visited Dec. 2005). Password manager.

[73] Pilhofer, F. April 2005. Password gorilla, a cross-platform password manager. http://www.fpx.de/fp/Software/Gorilla/ (Visited Dec. 2005). Password manager.

[74] Reichl, D. Keepass password safe. http://keepass.sourceforge.net/index.php (Visited Dec. 2005). Password Manager.

[75] Jedlinski, M. 2003. Oubliette. http://www.tranglos.com/free/oubliette_main.html (Visited Dec. 2005). Open source password manager.

[76] OperaSoftware. Opera mini. http://www.opera.com/products/mobile/operamini/phones/ (Visited Apr. 2006). Web browser for mobile phones.

[77] SunMicrosystems. Midp 2.0 api. http://www.sun.com/software/communitysource/j2me/midp/download20.xml (Visited Mar. 2006).

[78] Stokes, J. H. Understanding moore's law. ars technica. http://arstechnica.com/articles/paedia/cpu/moore.ars/1 (Visited Apr. 2006), February 2003.

A Thesis proposal

Single Sign On : secure password store

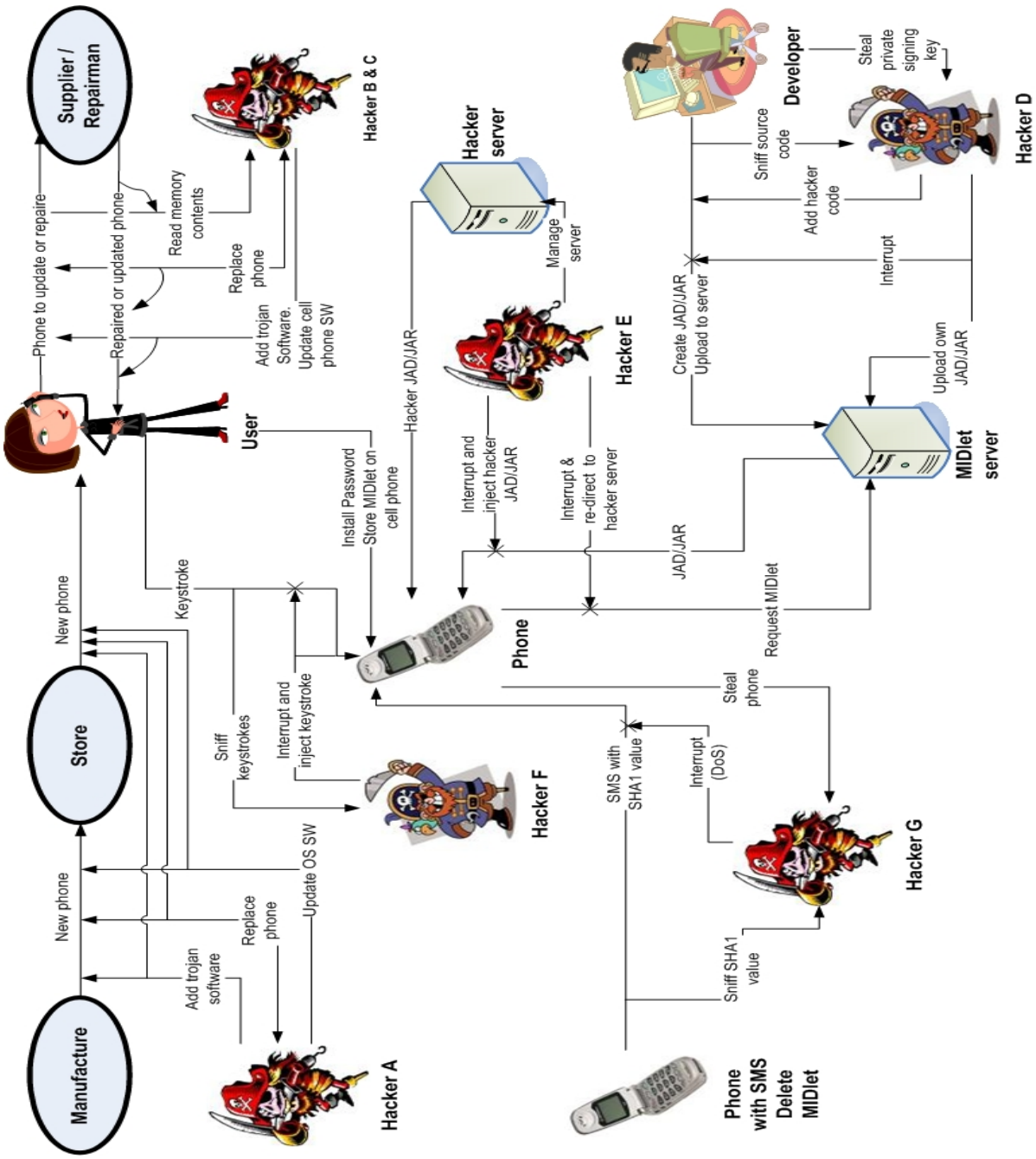
Background:
Most users have to keep track of a large number of passwords. For most users it is difficult to remember a large number of passwords without having to write them down or use the same password or PIN in many applications. There are several SSO products available, but they all seem to be less than perfect. NISlab has developed a Bluetooth based device that fits in the USB port and can be configured as a wireless keyboard.

Objective:
The student is to implement a JAVA MIDLET based password store and management system and an associated Bluetooth interface. The research challenge is to design a secure password store that is as close to the theoretically perfect user interface (e.g. number of keys to be pushed, training requirements, installation activities, etc).

Faculty contact:
Professor Einar Snekkenes
Email: einar.snekkenes@hig.no
Phone: +47 982 11 222
Office: A124I

http://www.hig.no/imt/index.php?id=270

B Complete DFD



C Tools Used

Different tools have been helping me in the work on this thesis. Especially has the LaTeX environment been of great help. LaTeX gives you the opportunities to automatically generate bibliography and table of contents, and will also handle references, pagination and the overall structure in a simple and predictable fashion. Download a version of MiKTEX [1], to start writing documents in LaTeX.

Many people will however struggle with image handling in LaTeX since eps is the basic image format. Regular JPEG images can on the other hand also be used, if you make use of the *pdflatex* compiler. The tex file will then automatically generate a pdf document of the tex file, which also will support JPEG images. The only thing you need to make sure, is not to include the *usepackage[dvips]hyperref* in the preamble.

Images can then easily be included in the text by the following code:

```
\begin{figure}[h] % Placement. h=here, t=top, b=bottom, p=float.
                  % A ! will force the chosen placement. E.g. !h will
                  % force the image to be placed at the given position
                  % in the tex file.
      \centering
      \includegraphics[scale=0.75]{yourJPEGimage.jpg}
      \caption[Short label]{Text assosiated with the image.}
      \label{fig:Image Reference Name}
\end{figure}
```

LaTeXEditor (LED) [2] is another nice software, which may give a new LaTeX user a flying start. You will have a lot of tools at your hands with this editor. With code completion, compiler, viewer, symbol choosers et cetera.

JabRef [3] has been used to organize and enter referenced literature. With this software, you can easily generate BibTex files, search through the entered information, associate the references with web pages or pdf documents for fast access among many other nice features. This software has indeed been to great help, especially when the amount of references increased.

Microsoft Office Visio 2003 has been the tool used to create the different data flow diagrams used in this thesis. Large illustrations can easily be generated with this tool, which also has got a lot of pre-installed images and illustrations that can be used.

[1] MiKTEX can be downloaded from `http://www.miktex.org`
[2] LaTeXEditor can be downloaded from `http://www.latexeditor.org`
[3] JabRef reference manager, can be downloaded from `http://jabref.sourceforge.net`

D Stealing MIDlet, source code

```
/*
 * SealingMIDlet.java
 * Created on 13. mars 2006, 13:16
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
import org.bouncycastle.crypto.Digest;
import org.bouncycastle.crypto.digests.SHA1Digest;

/**
 * @author Tommy
 */
public class StealingMIDlet extends MIDlet implements CommandListener {
    private Form form1;
    private Command exitCommand;
    private TextField textField1;
    private Command okCommand;
    private Alert alerten;
    private StringItem stringItem1;

    /** Creates a new instance of StealingMIDlet*/
    public StealingMIDlet() {
        initialize();
    }

    /** Called by the system to indicate that a command has been invoked
     *  on a particular displayable.
     *  @param command the Command that was invoked
     *  @param displayable the Displayable on which the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        if (displayable == form1) {
            if (command == okCommand) {
                findRMSname();
            } else if (command == exitCommand) {
                exitMIDlet();
            }
        }
    }
```

```
/**
 * This method initializes UI of the application.
 */
private void initialize() {
    getDisplay().setCurrent(get_form1());
}

/**
 * This method should return an instance of the display.
 */
public Display getDisplay() {
    return Display.getDisplay(this);
}

/**
 * This method should exit the midlet.
 */
public void exitMIDlet() {
    getDisplay().setCurrent(null);
    destroyApp(true);
    notifyDestroyed();
}

/** This method returns instance for form1 component and should be
 *  called instead of accessing form1 field directly.
 *  @return Instance for form1 component
 */
public Form get_form1() {
    if (form1 == null) {
        form1 = new Form(null, new Item[] {
            get_textField1(),
            get_stringItem1()
        });
        form1.addCommand(get_exitCommand());
        form1.addCommand(get_okCommand());
        form1.setCommandListener(this);
    }
    return form1;
}

/** This method returns instance for exitCommand component and should
 *  be called instead of accessing exitCommand field directly.
 *  @return Instance for exitCommand component
 */
public Command get_exitCommand() {
```

```
        if (exitCommand == null) {
            exitCommand = new Command("Exit", Command.EXIT, 1);
        }
        return exitCommand;
    }

    /** This method returns instance for textField1 component and should
     *  be called instead of accessing textField1 field directly.
     *  @return Instance for textField1 component
     */
    public TextField get_textField1() {
        if (textField1 == null) {
            textField1 = new TextField("Enter RS name \nto view content",
"", 120, TextField.ANY);
        }
        return textField1;
    }

    /** This method returns instance for okCommand component and should
     *  be called instead of accessing okCommand field directly.
     *  @return Instance for okCommand component
     */
    public Command get_okCommand() {
        if (okCommand == null) {
            okCommand = new Command("Ok", Command.OK, 1);
        }
        return okCommand;
    }

    /** This method returns instance for alerten component and should
     *  be called instead of accessing alerten field directly.
     *  @return Instance for alerten component
     */
    public Alert get_alerten() {
        if (alerten == null) {
            alerten = new Alert(null, "<Enter Text>", null, AlertType.INFO);
            alerten.setTimeout(-2);
        }
        return alerten;
    }

    /** This method returns instance for stringItem1 component and should
     *  be called instead of accessing stringItem1 field directly.
     *  @return Instance for stringItem1 component
     */
    public StringItem get_stringItem1() {
```

```
        if (stringItem1 == null) {
            stringItem1 = new StringItem("Record Stores in " +
                          getAppProperty("MIDlet-Name") + ":", "");
            String [] recorder = RecordStore.listRecordStores();
            String tmp = "";
            for(int i = 0; i<recorder.length;i++) {
                tmp += "\n" + recorder[i];
            }
            stringItem1.setText(tmp);
        }
        return stringItem1;
    }

    public void startApp() {
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void findRMSname() {
        String data = "";
        try {
            RecordStore rs = RecordStore.openRecordStore(get_textField1().getString(),
                                                         false);
            RecordEnumeration renum = rs.enumerateRecords(null,null,false);
            while( renum.hasNextElement() ) {
                byte [] recordData = renum.nextRecord();
                data += "\n" + new String(recordData);
            }
            rs.closeRecordStore();
        } catch (RecordStoreException rse) { data = "Fant ikke RS\n" +
                                             get_textField1().getString(); }
        get_alerten().setTimeout(Alert.FOREVER);
        get_alerten().setTitle("RecordStore");
        get_alerten().setString(data);
        get_alerten().setType(null);
        getDisplay().setCurrent(get_alerten());
    }
}
```