Java Security —an Infrastructure for Secure Client-Server Communication

Thesis for the degree Master of Science

Lars-Helge Netland
Department of Informatics
University of Bergen

June 2005

The NoWires Research Group

http://www.nowires.org

Preface

Web application programmers tend to focus on services when developing networking applications. Numerous services are designed to communicate with each other and are made available through service-oriented architectures. The emphasis on functionality has taken attention away from the production of secure and reliable systems. In this thesis, approaches to secure client-server communication will be presented. The important step is the development of an underlying infrastructure that can provide basic security mechanisms. An example framework in Java will be refined and improved throughout the text to show readers how to implement the concepts discussed.

The intended audience for this thesis is students and IT professionals with an understanding of Public Key Infrastructures and Java programming. People with background in just one of these categories will have a harder time following the text, but are encouraged to continue reading. References are included throughout the document, enabling readers to go to the sources for an in-depth presentation.

Acknowledgments

First of all I would like to thank my supervisor, Professor Kjell Jørgen Hole, for brilliant guidance throughout this thesis. Our weekly meetings and daily discussions have been invaluable to the quality of my work. I also owe Yngve Espelid my deepest gratitude for his major involvement in reaching the bottom of Java security and the ongoing NIO/SSL project, and for valuable comments on this document. I also thank Thomas Tjøstheim and my uncle Johs for proof-reading the thesis. If I were to give these four a beer for every mistake and shortcoming they have corrected, I'd be broke. Both socially and academically, the members of The NoWires Research Group have been very important. Our monthly "excursions" to Onkel Lauritz have meant a lot to me. I would also like to thank my common-law spouse Kristine for her patience and great support. My finances over the last year have been taken care of my poker-playing brother, allowing me to concentrate on research. In closing, I thank my parents, Audun and Kari, for always pushing me beyond the limit of my capabilities.

Contents

List of Tables

List of Figures

Important acronyms

Acronym	First page occurrence
CA: Certification Authority	18
CRL: Certificate Revocation List	18
DN: Distinguished Name	18
DoS: Denial-of-Service	1
J2EE: Java 2 Enterprise Edition	3
J2SE: Java 2 Standard Edition	2
JAAS: Java Authentication and Authorization Service	34
JCA: Java Cryptography Architecture	32
JCE: Java Cryptography Extension	34
JSSE: Java Secure Sockets Extension	34
JVM: Java Virtual Machine	23
NIO: New Input/Output	4
PKI: Public Key Infrastructure	3
RA: Registration Authority	18
SSL: Secure Socket Layer	2
TLS: Transport Layer Security	35

Chapter 1

Introduction

Today, close to 900 million people use the Internet [59]. Over the last 5 years the Web's user base has expanded with 146.2% worldwide. Along with the rapid growth of this relatively new communication medium, a wealth of new services have been introduced. On-line you can now shop, gamble, date, pay bills, and find out how much your neighbor made last year. Not only do you have almost endless possibilities, but with the recent explosive addition of wireless networks, you are free to explore them from just about anywhere. And people are actively using the Web: A survey [46] from 1st quarter 2004, shows that 31% of the Norwegian population aged between 16-74 had ordered or purchased goods on-line over the last 3 months.

With the widespread use of the Internet, users have also experienced a drastic increase in Web related crime. Common felonies include theft of credit card information, deployment of computer worms and viruses, illegal access of confidential information, and Denial-of-Service (DoS) [11] attacks. According to Eurostat [47], 32.9% of Norwegian Internet users lost time or information due to computer viruses in 2004. The availability of the Web is complicating the investigation of cybercrime. Bringing perpetrators to justice often necessitates cooperation across national borders. Kazakhstani malicious hacker Oleg Zezev operated from his homeland when he in March 2000 broke into Bloomberg L.P.'s computer systems, and later tried to blackmail the company's founder Michael Bloomberg. The attacker demanded $200,000 or else he would destroy Bloomberg's systems. Zezev was apprehended in an undercover operation in London, a joint effort by the FBI and British intelligence, later that same year [31].

Evil-minded hackers can often obtain access to large networks of computers, from which they can launch powerful attacks. Internet Service Provider (ISP) Telenor discovered a network of approximately 10,000 "zombie" computers in September 2004 [22]. These networks are often called botnets. A botnet consists of computers that act like robots, communicating with each other and a central server. A skilled malicious hacker in possession of such an army of computers can accomplish almost anything on the Web.

1.1 Defining Security

The term security means different things to different people. Computer network people often talk about firewalls and intrusion detection systems when asked if their computer systems are secure. Folks in management sometimes will argue that their organization's systems are secure because they implement the Secure Socket Layer (SSL) protocol. Security is not a feature. It is a system property [5]. In building secure systems for the Internet, one has to realize that nothing can be considered 100% secure. Computer professionals actually need to take the opposite view: consider all client systems as fundamentally insecure [41]. As a consequence, security-minded developers apply best practices in order to build as robust systems as possible. In addition, security must be integrated into the software development life-cycle in order to craft reliable software. A security review at the end of the software process is not sufficient to produce good quality programs. Security needs to be considered in a much wider sense, as a set of non-functional goals. A thorough treatment of the matter includes procedures for prevention, traceability and auditing, monitoring, privacy and confidentiality, anonymity, authentication, and integrity [5].

Taking the necessary steps to secure your on-line systems is hardly a one-man show. To implement the procedures mentioned above, you need a team of security professionals. In this thesis, a subset of the total package of security services will be considered. In particular, the basic services of authentication, integrity, and confidentiality will be addressed. From this point on these three will also be referred to by the term *primary security services*.

The main goal is to explain in theory and through code examples how readers can construct an infrastructure that can be used to build a wide range of security services. But please keep in mind that a comprehensive security solution entails much more than these services.

1.2 Distributed Computing and Java

The security framework to be designed in this thesis is meant to be deployed in an Internet setting. As such, it is required to function in a hostile environment, meaning that the implementation language should be as secure as possible. Also, it is desirable to choose a platform that can keep up with the advances in technology. In particular, it seems certain that small devices such as cell phones will be deployed as fully capable client systems on the Internet.

Currently, given the factors above, Java is the best choice for your programming platform. Languages such as C and C++ were not designed with the Web in mind, and do not contain mechanisms to secure your applications from buffer overflows and arbitrary memory access. C# is a relatively new language that borrows heavily from Java. The biggest concern regarding C# is the lack of penetration in the mobile segment. Current cell phones contain limited computing capabilities, but do provide some support for Java development through the Java 2 Micro Edition platform.

In this thesis, programs will be developed using the Java 2 Standard Edition (J2SE) computing environment. It should be stressed that J2SE is a subset of the Java 2 Enterprise Edition (J2EE) platform, meaning that the security framework created can be exported to a J2EE setting. The most important libraries in Java that will be used include `java.nio`, `javax.net.ssl`, `java.net`, `java.security`, and `javax.net.ssl`. Also, a few classes from third party provider Bouncy Castle [54] will be used. For more information on the code, consult the author's web site [7].

1.2.1 Infrastructure

In terms of software, the universal tool platform Eclipse is used to develop programs. Most of the graphics are created in OmniGraffle Professional, except the figures in Section 3.2, which were made by Yngve Espelid in Visio. The main body of text will be written in LyX.

The code in chapters 2 and 4 has been developed on an Apple computer, a Powerbook with a 1.25 GHz G4 processor and 512 MB of RAM. The code in Chapter 5 has been developed on a Pentium 4 with a 2.8 GHz processor and 1 GB of RAM. All of the code can run on any system with J2SE version 5.0 installed.

1.3 Application Domains

As stated in Section 1.1, the main goal of the thesis is to develop a security framework capable of providing the primary security services. The true power of such a framework cannot be fully appreciated without an understanding of how it can be put into an operational context. An ideal candidate relying on the primary security services is Internet banks. They have high demands in terms of availability and security, making the development of them a difficult task. The possibility of getting access to lots of money, makes Internet banking systems attractive targets for evil-minded hackers. The challenges present in the development of these solutions make them attractive to security researchers as well. Also, a constellation called BankID [4] is currently working on a project that aims to improve Norwegian Internet banking systems. There is a bit of confusion surrounding this initiative, especially since they have been reluctant to give away details on their system. They have been very clear on one thing: their solution will include a Public Key Infrastructure (PKI).

The decision to implement the security constructs in an Internet banking scenario does not mean that the techniques can't be applied to different settings. Once you have a security infrastructure in place, you should be able to provide services to a whole range of different applications. The important step is to create the underlying infrastructure. The Internet banking system is included to show you how to make use of the services provided by the infrastructure. Other interesting application domains include electronic voting, e-commerce, and payment solutions with handheld devices.

1.4 Structure of Thesis

In this thesis, a security critical application developed for the Internet is considered. At first, the reader will be made familiar with a stripped down version of the system, a functioning prototype. In subsequent chapters, weaknesses in the initial system will be identified, and the application will undergo changes to meet various security requirements. The underlying theory will be explained before implementations are given. This way, readers are given theoretical background on security mechanisms and examples of how to get from theory to code. Each chapter ends with a short summary highlighting the most important parts of the text.

Chapter 2 sets up a communication framework to be used throughout the thesis. It relies on the client-server paradigm to exchange information. In the beginning, Internet banking is presented as a typical networking system. Later, a minimal prototype using the New Input/Output (NIO) library in Java is given.

Chapter 3 opens with a short introduction to PKI. Concepts that are necessary to understand when implementing a PKI will be presented. Readers should see texts such as [1, 15] for a more thorough understanding of the theoretical foundations of the infrastructure. Next, Java is discussed in a security context. A walk-through of the programming platform's approach to security is given. In closing, cryptographic capabilities in Java are presented.

Chapter 4 discusses current Internet banking systems. A discussion of next generation banking software is also included. Next, shortcomings in the initial communication framework are identified. These are later dealt with by applying the theory given in Chapter 3. After the development of a PKI for Internet banking, current challenges in PKI are discussed.

Chapter 5 addresses the widely used SSL protocol. First, current deployment of SSL in Internet banking systems are debated, and a simple attack is given. Next, a short introduction to the workings of SSL is given. Then, a prototype using SSL in Java will be implemented. Rounding off the chapter is an overview of SSL's role in a PKI.

Chapter 6 ends the thesis. First, a short summary highlights the main points of the text. Next, a number of conclusions are given. In closing, a list of ways to continue and build on the work in the thesis is presented.

Chapter 2

The Communication Framework

This chapter outlines the communication framework to be used throughout the thesis. First, Internet banking is presented to illustrate usage of network protocols. Second, the client-server paradigm is explained. The description is fairly superficial, the interested reader is strongly encouraged to read more about this topic in Sommerville's book on software engineering [49] or in undergraduate networking textbooks, such as "Computer Networking" by Kurose and Ross [34]. A Java implementation of a client-server protocol, featuring the NIO library [13] added in Java 2 Standard Edition (J2SE) version 1.4, ends the chapter.

2.1 Network Protocols and Internet Banking

Communication over a network is accomplished through the use of communication protocols. One such protocol is the Transport Control Protocol (TCP). Packages sent via TCP are guaranteed to arrive in order and in their entirety, but the sender has no guarantee regarding the time of arrival. An alternative to TCP is the User Datagram Protocol (UDP). The latter protocol is connectionless and makes no guarantees about delivery. The upside for UDP is that it introduces less overhead, allowing packets to propagate faster through the network. Programs in this thesis will use TCP, as alterations in the data sent can't be tolerated. Please consult Request For Comments (RFC) 793 [18] and 768 [17] for more information on TCP and UDP, respectively.

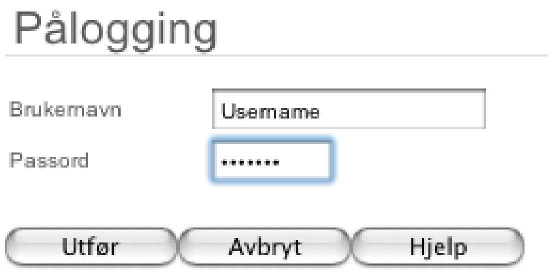
A typical scenario in which sensitive information is exchanged over a publicly accessible network is Internet banking. Customers log into the bank's servers to perform basic services, such as checking their balance, paying bills, or managing stocks and bonds. All of the above can be accomplished through the use of an Internet browser.[1]

[1]Popular ones include Internet Explorer, Mozilla, Opera, and Safari.

2.1.1 Real life example: Sparebanken Vest

Fig. 2.1 and Fig. 2.2 show screenshots of the login procedure in Sparebanken Vest's Internet banking system. The two fields in Fig. 2.1 prompt the user to input a username and the corresponding password. The format of these are specified as follows:

- The username is the full given name of the customer. All characters not being letters are removed, e.g., a person named Abdoulaye M'Baye would get the username: abdoulayembaye.
- The password is defined by the user and is verified by the bank. It must contain at least 6, but no more than 30 characters. The password is not case-sensitive and can include any character in the English alphabet. In addition, the bank recommends a mixture of letters and numbers.

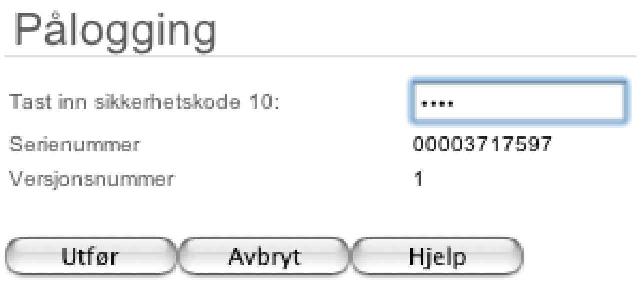

Pålogging

Brukernavn Username
Passord •••••••

Figure 2.1: Login Sparebanken Vest

In Fig. 2.2 the user is prompted for a four digit one-time Personal Identification Number (PIN), which is given by a table of 50 such PINs that is sent to the customer prior to usage of the system. After 35 logins the bank sends the user 50 new PINs.

Fig. 2.3 is a sequence diagram illustrating the flow of information when logging into Sparebanken Vest's Internet banking system:


Pålogging

Tast inn sikkerhetskode 10: ••••
Serienummer 00003717597
Versjonsnummer 1

Figure 2.2: Continued login Sparebanken Vest

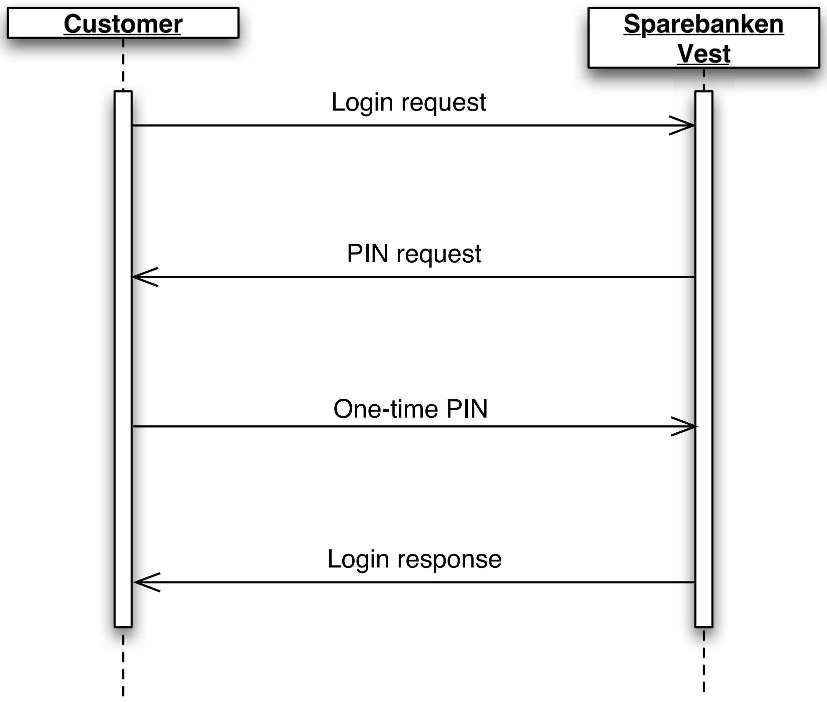


Figure 2.3: Sequence diagram: Login Sparebanken Vest

Main flow of events: First, the user sends his username and password to the bank, as indicated by the arrow labeled "Login request." Next, the financial institution responds by requiring the customer to send a one-time PIN. Upon looking up the required PIN, the user sends it off to Sparebanken Vest. The login procedure culminates in a response from the bank, granting the customer access to his/her accounts.

Exceptional flow of events: If the pair (username, password) fails to validate the customer, the bank prompts the user to resend these values up to a maximum of three times. Failing the last attempt closes the account. The customer must contact the bank via telephone or mail to re-open the account. If the user inputs an invalid one-time PIN, the scenario continues as if it were a mismatched (username, password) pair.

2.2 The Client-Server Paradigm

A widely used model for communication on the Internet is the *client-server model.* Merriam-Webster on-line dictionary defines a server [37] to be "a computer in a network that is used to provide services (...) to other computers in the network." Examples of services provided by a server are access to files or shared peripherals and routing of e-mail. A client is defined

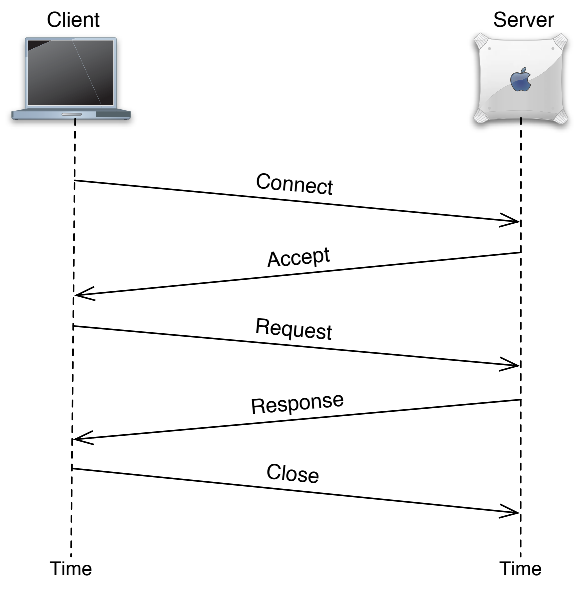


Figure 2.4: Client-server communication protocol

[36] to be "a computer in a network that uses the services (...) provided by a server." In this thesis, a server should be understood as a continuous provider of services, meaning that the server is able to accept and process multiple simultaneous connections. Such a scheme can be realized through the use of multiplexing I/O.

2.2.1 Communication model

In the following, we design a simple model for client-server communication, enabling the reader to get an easy start with both concepts and implementation issues. The communication protocol is depicted in Fig. 2.4.

The client initiates the communication by sending a connection request, indicated by the "connect" arrow in Fig. 2.4. The server responds with an "accept" message. Upon establishing the connection, the client then sends a service request, graphically represented in the figure by the arrow labeled "request." Next, the server sends a response. The communication ends with a "close" message sent from the client, indicating to the server that no additional services will be called for in the current session.

Looking back at the login procedure described in Section 2.1.1, notice that Sparebanken Vest uses a variant of the client-server paradigm to realize their Internet banking system. Customers use clients, disguised as Internet browsers, to connect to the bank's web server located at `www.spv.no`. In fact, except from the messages sent to initiate and close the communication,[2] Fig. 2.3 and Fig. 2.4 have the same structure, i.e., they are both request/response based. In terms of networking code, the text will rely exclusively on the

[2]Omitted in Fig. 2.3 in order to keep the figure simple.

client-server paradigm.

2.2.2 Client-server architecture

Having identified and described the interactions in an application, you are in position to design the system architecture. It can be useful to dissect the application layer into three layers:

1. The *presentation layer* presents information to the user and is responsible for all user interaction.
2. The *processing layer* is made up of all application logic details.
3. The *data management layer* is responsible for all database operations.

The exchange of information between the layers is portrayed in Fig. 2.5. The segmentation of the application layer enables us to distribute the different processes across multiple processors, possibly across a network. Please note that even if we regard these as separate processes, they don't have to be physically separated. The same computer could preform presentation, processing and data management.

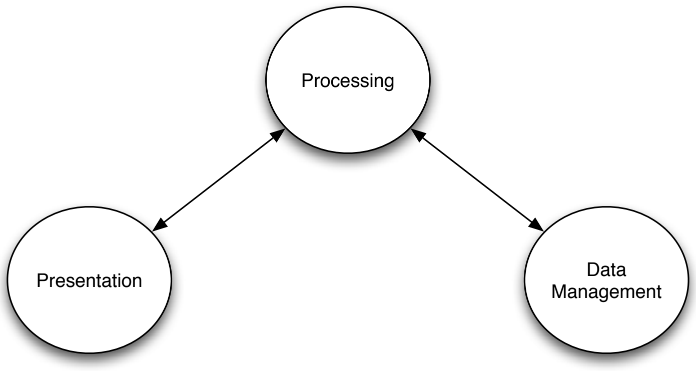


Figure 2.5: Application layers

In Sparebanken Vest's system, presentation is done in the individual client browser. Processing is handled by the bank's webservers. Sparebanken Vest's system is proprietary, so we can only make an educated guess when saying that they have an external data management system. Looking at the quantity of information that flows through the system, they would have to have a separate system for database operations.

A system involving a set of distributed components is known as a multi-tier system. This thesis will focus on three-tier client-server architectures, like the one displayed in Fig. 2.6. The three components portrayed are:

- client, responsible for presenting information to the user,
- webserver, conducts application processing, and
- database server, handles database transactions.

To ease the burden on the webserver, some of the processing can be done on the clients. This reduces the amount of traffic sent across the network, allowing higher throughput for the data moving between the processes. If clients are to do processing, you need relatively fast CPUs and large amounts of memory to do so efficiently. As of June 2005, this requirement forces an exclusion of smart phones, Personal Digital Assistants (PDAs) and smaller devices. Variants where mobile phones are used introduce workarounds like the one sketched in “Design and Implementation of a PKI-based End-to-End Secure Infrastructure for Mobile E-Commerce” [6]. A scheme involving client-side processing is known as a *thick-client* model.

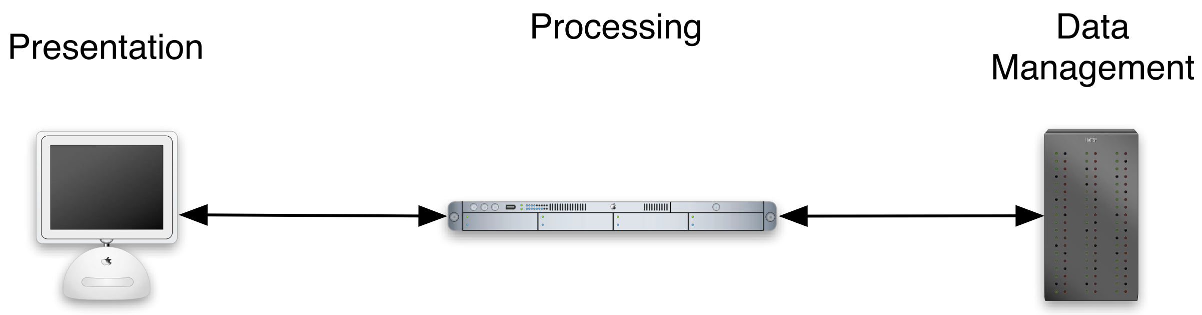


Figure 2.6: Three tier client-server architecture

2.3 Networking in Java

The following subsections show a possible implementation of the login procedure in Sparebanken Vest’s Internet banking system. The application is in no way meant to mimic the bank’s existing solution, but offers a view of how to implement a login scheme in Java using the NIO Application Programming Interface (API). I/O addresses transfer of data between processes and/or devices, e.g. from a web server to a client. Ron Hitchens gives a solid introduction to NIO in his book “Java NIO” [13]. Construction of highly scalable servers is addressed in [42].

2.3.1 NIO

Introduced in Java 1.4, the set of classes in the NIO library offers an enhanced approach in dealing with I/O issues. The genius of NIO is to rely on system specific calls in the underlying OS to handle I/O services. The interesting Java classes enabling advanced I/O are situated in the `java.nio`, `java.nio.channels`, and `java.nio.charset` packages. For an overview of the classes, consult Sun's on-line Java 1.4.2 API specification [24].

An important new feature is the ability to perform *readiness selection.* The `java.nio.channels.Selector` class encapsulates this concept. An instance of `Selector` can be set to monitor the traffic on a set of registered *channels.* A channel is used to transport data to and from processes. Whenever a channel is ready to perform some kind of I/O, the selector is notified and the event can be dealt with as needed. The result is that one thread can efficiently manage a large number of channels. Prior to NIO, the single thread scheme involved checking each channel manually for new activity.

2.3.2 The implementation

Segments of the implemented system are presented in this section. These are meant to highlight design choices and important NIO concepts used to realize the client-server scheme. A complete listing of the program code for Chapter 2 can be found on my web site [7].

Five classes make up the application: `Communicator.java`, `Client.java`, `Server.java`, `StartClient.java`, and `StartServer.java`. The latter two are engine classes, meaning that their sole purpose is to create instances of clients and servers. `Communicator.java` is an abstract class encapsulating common features of clients and servers. `Client.java` and `Server.java` define specific client and server behavior, respectively.

2.3.2.1 Server implementation

Before the communication can start, the server has to be bound to a specific port listening for incoming connections. Here is code for creating a non-blocking server:

```
// Open a new channel from the server
serverChannel = ServerSocketChannel.open();

// Get the serversocket associated with the channel
socket = serverChannel.socket();

// Bind the socket to the communication port given by SERVICE_PORT
socket.bind(new InetSocketAddress(SERVICE_PORT) );

// Instruct the channel to perform non-blocking I/O
serverChannel.configureBlocking(false);

// Register the serverChannel with the selector whose current
// interest is accepting new connections, indicated by constant:
// SelectionKey.OP_ACCEPT.
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

First a new channel is created. The socket associated with this channel is retrieved, allowing the service port to be set. Then the channel is made non-blocking. Finally, the channel is registered with a selector. The selector was created ahead of time by invoking the static `Selector.open()` method. When registering, an operation of interest is specified in the parameter list. At this point the server wants to accept new clients, which is indicated by the predefined variable `SelectionKey.OP_ACCEPT`.[3] Now the server is listening on the port given by `SERVICE_PORT` for new clients wanting to establish a connection.

Calling the `select()` method on a `Selector` returns the number of newly registered activities. A set of keys holding the underlying channels ready to perform I/O can be retrieved by invoking `selectedKeys().iterator()` on a given selector. The server-side code for getting and handling the previously mentioned key set is:

```
while( selector.select(15000) > 0) { //timeout set to 15 sec.
    Iterator keys = selector.selectedKeys().iterator();

    while (keys.hasNext() ) {
        SelectionKey selectKey = (SelectionKey) keys.next();

        // Removes from the underlying collection the last
        // key returned by the iterator
        keys.remove();

        // What type of key is this?
        if(selectKey.isAcceptable() ) {
            // set up new connection
            this.acceptClient(selectKey, selector);
        }
        else if(selectKey.isReadable() ) {
            // Read data from client
            String message = this.readData(
                                (SocketChannel)selectKey.channel() );
            this.processData(selectKey, message);
        }
    }
}
```

The server iterates through the keys, looking for incoming connections and data available for reading. If a client wants to connect, i.e. the call to `selectKey.isAcceptable()` returns true, a new non-blocking channel is created and registered with the selector.[4] The server wants to be notified whenever the new client writes a message to the channel, so

[3]Other valid operations include `OP_CONNECT`, `OP_READ`, and `OP_WRITE`.

[4]Future activities of interest in the new channel will be caught by a later `selector.select(15000)` invocation.

the channel is registered with the `SelectionKey.OP_READ` option. If there's data to be read from the channel, the `selectKey.isReadable()` method returns true. The data is then read and the resulting string processed in the `processData(SelectionKey, String)` method.

2.3.2.2 Client implementation

Looking back at the sequence diagram in Fig. 2.3, it is clear that the communication starts at the client. But before any messages in the protocol can be exchanged, the client has to connect to the server:

```
InetSocketAddress server = new InetSocketAddress(host, SERVICE_PORT);

channel = SocketChannel.open();

// Set non-blocking
channel.configureBlocking(false);

// Register channel with selector
channel.register(selector, SelectionKey.OP_READ);

// Connect to server
channel.connect(server);

while(!channel.finishConnect() ) {
    System.out.println("Waiting for server to accept connection...");
}
```

Host and service port are specified when creating an `InetSocketAddress` object. A channel is then created and registered with a selector. Current interest is set to the read operation: `SelectionKey.OP_READ`. The next step is to connect to the server. After issuing the connect command, the client waits for the connect operation to finish.

2.3.2.3 Client and server interactions

In order for the server and client(s) to fulfill the Sparebanken Vest login procedure, they have to agree on the format of the messages exchanged. The server is designed to recognize two types of requests:

1. A string on the form "usernamejohnjohnsen|passwordw3KpO9cQ" means that user 'johnjohnsen' with password 'w3KpO9cQ' wants to login.
2. A string on the form "PINxxxx" means that the user associated with the underlying channel has supplied a 4-digit PIN, where the string 'xxxx' contains the 4 digits.

Any other commands received at the server are discarded, and the corresponding channel is closed.

The client must recognize the following strings:

1. The string "PIN" indicates that the server is asking the client for a PIN.
2. "welcome!" indicates that the client was successfully authenticated and can be given access to the protected resources.

As previously mentioned, the client initiates the application logic in Sparebanken's login procedure. The user sends his username to the bank:

```
this.writeData(channel, "usernameabdoulayembaye|passwordrT7xX2Wb");
```

The server recognizes the message as a login request from a customer with username 'abdoulayembaye' and password 'rT7xX2Wb'. In order to distinguish communication sessions, the username and hash-code of the underlying channel are stored in a `Hashtable`:

```
if(message.startsWith("username") ) {
    // Get the username and store the request
    StringTokenizer st = new StringTokenizer(message, "|");

    // CAUTION can cause a NoSuchElementException if
    // the tokenizer is empty
    String username = st.nextToken();
    String password = st.nextToken();

    String user = username.substring(8);
    String passwd = password.substring(8);

    if(this.validateUser(user, passwd) ) {
        int hashCode = key.channel().hashCode();
        sessionInfo.put(new Integer(hashCode), user);

        // Prompt user for PIN
        this.writeData( (SocketChannel)key.channel(), "PIN");
    }
}
```

The username and password is extracted by using a `StringTokenizer` with delimiter '|'. In order to keep things simple, the supplied implementation always accepts the user, i.e. calling the `validateUser(...)` method returns true. Typically, the validation would include retrieving the (username, password) mapping from an external storage medium, such as a database.

The variable named `sessionInfo` is an instance of `Hashtable`, which is a collection of elements made up of (key, value) - pairs. At any given time, the program's hashtable contains the (username, hash-code of the underlying channel) pair of any clients that have sent the initial login request, but have not yet supplied a PIN. After storing the necessary information, the server demands a PIN in accordance with the predefined protocol:

```
this.writeData( (SocketChannel)key.channel(), "PIN");
```

The client recognizes the message and willingly obeys the order:

```
if (message.equals("PIN") ) {
    System.out.println("Server is asking for PIN");
    this.writeData( (SocketChannel)key.channel(), "PIN1234");
}
```

The PIN sent is 1234. Other clients would change these 4 digits as appropriate. On the receiving end, the server registers new activity on the channel corresponding to the before-mentioned client, and tries to validate the customer:

```
if (message.startsWith("PIN") ) {
    String tempPasswd = message.substring(3);
    String username = null;

    // Retrieve the username for this channel
    Object currChannel = key.channel();

    username = (String) sessionInfo.get(currChannel.hashCode() );
    sessionInfo.remove(currChannel.hashCode() );
~
    if (username == null) {
        System.out.println("Unable to recognize user.");
        key.channel().close();
    }
    else {
        if (this.validateUser(username, tempPasswd) ) {
            // Send welcome message to client
            this.writeData( (SocketChannel)key.channel(), "welcome!");
            key.channel().close();
        }
        else {
            // UNREACHABLE!
            key.channel().close();
            }
        }
    }
```

The hash-code of the underlying channel is used to retrieve the username corresponding to the PIN request. If the hashtable doesn't contain an entry with the given hash-code, the channel to the client is immediately closed. If instead a match is found, the external validation procedure is called. Upon successfully authenticating the customer, the server sends a welcome message to the client.

2.4 Summary

The **client-server paradigm** is a request-response based communication scheme suitable for exchanging messages in a large network, such as the Internet. Sparebanken Vest offers **Internet banking**, realized through an implementation of the previously mentioned paradigm. In this thesis, the client-server scheme will be used in the development of networking applications. In addition, the clients will be required to do processing. Such a scenario is called a **thick-client model**.

The NIO library in Java performs **readiness selection** to efficiently handle numerous clients in a single-threaded model. A lightweight communication framework addressing the login feature of Sparebanken Vest's Internet banking system, can be implemented using Java NIO. The implementation given in Section 2.3.2 suffers from quite a few shortcomings that will be addressed and dealt with in subsequent chapters.

Chapter 3

Cryptography

This chapter is meant to provide readers with the theoretical background needed to embark on a secure coding project using the cryptographic APIs in Java. Parts of the discussion is quite technical, in particular Section 3.2. An understanding of the concepts presented is needed to follow the evolution of the example framework given in Chapter 2.

The chapter opens with a short introduction to cryptography. In this opening section, cryptographic terminology used throughout the thesis is defined. A background in cryptography is required to follow the discussion. An understanding of public key cryptography is a necessity. Next, the security mechanisms in the Java programming language are presented. The chapter ends with a closer look on the cryptographic services and features available with the Java platform.

3.1 Introduction

The word cryptography is of Greek origin, and comes from the words *kryptos* meaning hidden and *graphein* meaning to write [9]. In short, cryptography is the science of encrypting and decrypting text. Simon Singh gives a publicly accessible introduction to cryptography in “The Code Book — The Secret History of Codes and Code-Breaking” [48]. For a more thorough mathematical treatment of the subject, consult “Cryptography—Theory and Practice” by Stinson [52].

The ideal security solution would be a framework capable of delivering services to a wide range of systems. In order to gain credibility, the infrastructure must facilitate analysis. Other desirable features are interoperability and extensibility. One promising infrastructure able to fulfill these requirements is known as the Public Key Infrastructure (PKI). It relies on public key cryptography to deliver security services. For an in-depth discussion on PKI see “Understanding PKI” [1] or “Planning for PKI” [15].

3.1.1 Defining PKI

A PKI can be set up to provide a wide range of services.[1] The implementation of an all-purpose PKI falls outside the scope of this thesis. The text will only deal with a handful of interesting and important services. The PKI that is to be implemented in Chapter 4 is required to offer the following:

Authentication: The proof of a binding of an identity to an identification tag. The binding is established by a trusted third party. An example of such a scenario in real life is the creation of passports, where the the person to whom the passport is issued represents the identity, and the passport itself is the identification tag. The trusted third party is the issuing government.

Integrity: An assurance to the involved parties that the data were not modified or tampered with in transit.

Confidentiality: Data travelling across a network should not be viewable or accessible for others than the sender and the intended recipient(s).

In order to meet these requirements, the infrastructure relies on a few central components and concepts to be discussed in the next sections:

- X509 public key certificate
- Certificate Revocation List (CRL)
- Certification Authority (CA)
- Registration Authority (RA)
- Repository
- Archive
- Certificate chain

3.1.1.1 X509 public key certificate

Authentication in a PKI is realized through the use of *public key certificates.* Many different formats for certificates exist, but the most widely deployed is *X509 version 3 public key certificates.* The X509 certificate structure is outlined in Table 3.1.

In a certificate, a binding between a public key and a Distinguished Name (DN) is created. A DN is a string on the following format [CN, OU, O, L, ST, C] that is meant to uniquely identify the owner:

[1] In principle that is. No current product offers comprehensive PKI support [1].

Field	**Description**
Version	X509 public key certificate version number. Three versions exist: 1, 2, and 3.
Serial Number	Unique identifier for the certificate within the CA.
Signature	Identifies the algorithm that was used to sign the certificate.
Issuer	The DN of the issuing CA.
Validity	Specifies two dates between which the certificate is considered valid, unless it has been otherwise canceled.
Subject	The DN of the owner of the certificate.
Subject Public Key Info	The public key and algorithm identifier for the owner.
Issuer Unique ID	Optional field assigning a unique ID to the CA issuing the certificate. Not recommended for use in RFC 3280.
Subject Unique ID	Optional field assigning a unique ID to the owner of the certificate. Not recommended for use in RFC 3280.
Extensions	Includes a set of optional extension fields. These can be marked as critical or noncritical. Failure in processing a critical extension forces the examiner to reject the certificate, whereas a noncritical extension can be skipped. Examples include *Certificate Policies*, *Policy Mappings*, and *Subject Alternative Name*.
Digital Signature	A signature on all the above fields by the issuing CA.

Table 3.1: X509 Version 3 public key certificates

- Common Name (CN): The full given name of the owner.
- Organizational Unit (OU): The unit the owner belongs to within her organization.
- Organization (O): The organization the owner is associated with.
- Locality (L): The location of the owner, i.e. city.
- State (ST): The state or province the owner resides in.
- Country (C): Two-letter country code for the owner.

To learn more about X509 certificates, see RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [21].

3.1.1.2 Certificate Revocation List (CRL)

The validity period of a certificate is determined by the field Validity presented in Table 3.1. In certain situations it's desirable to cancel the certificate before the end of that time window, e.g., in the case of theft or suspected compromise of the private key. The victim would then need to notify all other members of the PKI about the revocation. In practice, the user informs the CA who in turn will inform the rest of the user community. This can be accomplished by using a CRL, which is a listing of serial numbers belonging to certificates that have been canceled. X509 public key certificates contain an extension named 'CRL Distribution Point', which points to the location where revocation information for the given certificate can be obtained. Other techniques for revocation exist, such as on-line revocation solutions and schemes relying on reduced data structures [61], but a discussion of these fall outside the scope of this thesis. CRLs were mainly adopted because in terms of revocation, Java only includes CRL based solutions.

3.1.1.3 Certification Authority (CA)

In order to protect the integrity of a public key certificate, it is signed with the issuer's private key. The issuing party is called a CA. The CA is often an independent third party that all the communicating parties assumingly trust. When creating a certificate, the user must identify himself to the CA. The identification procedure varies with the intended use of the certificate. Consider a certificate that is to be used in the signing of multi-million dollar contracts. Identification would probably involve meeting with the CA in person, presenting your passport, birth certificate, and possibly other credentials. Getting a certificate intended for signing of e-mails is typically less cumbersome.

If Alice wants to communicate in a secure fashion with Bob, she first has to get his certificate. Lets assume they both trust a common CA. Alice would contact that authority for Bob's certificate. Upon receiving it, Alice is convinced that she's in possession of Bob's public key. This doesn't mean that he should be trusted, merely that the CA is vouching for the binding between Bob's identity credentials and the corresponding public key.

The CA is required to:

- **Register** new users,
- **issue** certificates and CRLs,
- **distribute** certificates and CRLs, and
- **store** certificate information.

A number of different providers of CA services exist. Two of the most recognized ones include Thawte [53] and Verisign [58].

3.1.1.4 Registration Authority (RA)

The CA is the cornerstone of any PKI. The tasks it performs are critical in terms of the PKI's overall functioning. The CAs responsibilities are often divided and distributed to other components in the architecture. The registration procedure can be handled by one or more RAs. Each of these are issued a certificate by the CA. In subsequent communications, the RA signs the customer information with its private key, ensuring the message integrity.

3.1.1.5 Repository

The task of distributing certificates and CRLs can be given to a *repository*. The CA creates and issues certificates and CRLs, which in turn are signed by the authority and sent to the repository. The address and protocol of the repository is announced to make the system available to clients. The integrity of the information is achieved through the CA's signature, allowing the repository to focus on availability and performance issues.

3.1.1.6 Archive

An archive in a PKI context is a tamper-proof long-term storage facility that holds certificate information generated by the CA. The important problem this construct needs to solve is whether or not a certificate and its corresponding private key was valid at a given point in time. By answering that question the archive can solve disputes involving the validity of a digital signature. As an example, consider a contract digitally signed by two companies. A while after the signing, one of the organizations deny any involvement in the business agreement. Archive data on the certificates of the two parties can help solve the situation. This property is more commonly referred to as *non-repudiation.*

3.1.1.7 Certificate chain

Consider a scenario where Alice has obtained Bob's certificate. She does not trust Bob directly, so Alice needs to create a path from Bob's certificate to an entity that she trusts. First, Alice obtains the certificate for the entity that has signed Bob's certificate. Then, continuing recursively, she continues to add the signer's certificate until she reaches an issuer she already trusts or a *self-signed* certificate. In the former case, Alice has a complete *certificate chain* and can start validating the individual entries top-down. A self-signed certificate is signed by the owner herself, which means that the issuer and subject fields are identical. The term self-signed is used interchangeably with *root*. If Alice arrives at a root certificate not known to her, she needs to establish trust in the issuing party in order to communicate with Bob.

After accumulating the complete path, Alice needs to validate each entry. The process of validation involves:

1. Verifying the digital signature,
2. checking that the DN of the certificate corresponds to the identity Bob,
3. examining the validity period, and
4. checking revocation status.

The procedure starts with the entity she trusts, possibly a root CA, then the subject of that starting certificate, continuing all the way to Bob's certificate.

In Chapter 4 the various components are put together to form a more robust Internet banking system.

3.2 Java Security Fundamentals

The following section describes the underlying security foundation of the Java platform. For a more thorough presentation see "Java Security" [40] by Scott Oaks.

3.2.1 The Java language rules

The basic building blocks in the Java security model is a set of language specific rules. Their primary purpose is to deny access to or modification of random locations in the memory of the hosting machine. If you're interested in the reasoning behind the rules, see Oaks[40]. The rules are

1. Access levels are strictly enforced.
2. Code cannot access arbitrary memory locations.
3. Entities marked with the `final` identifier cannot be changed.
4. Variables may not be used before they have been initialized.
5. You cannot access data outside your initial data set. E.g., attempts to access an array index that does not exist will result in an `ArrayIndexOutOfBounds`.
6. Objects cannot be arbitrarily cast into other objects.

3.2.1.1 Enforcement of the Java language rules

The constructs responsible for enforcing these rules are the *compiler*, *byte-code verifier*, and the *Java Virtual Machine* (JVM*)*. The first line of defence is the compiler. During compilation, every rule but 5 and 6 is checked; the compiler cannot enforce checking of array bounds or all cases of illegal casts. These checks will be completed at runtime. The problem with casting arises when two objects are not known to be unrelated:

```
Object maybeCar = myVector.elementAt(0);
Car ferrari = (Car) maybeCar;
```

There is no way for the compiler to know whether the object returned from the vector is a car, or just something posing as a car.

When classes are loaded in Java, the byte-code verifier provides a way to check the rules on the list above. In addition to the four first entries on the list above, the byte-code verifier also makes sure that:

- The format of the class file is correct.
- Every class has a single superclass.[2]
- There are no operand stack overflows or underflows.

The compiler and the byte-code verifier have overlapping responsibilities: they perform some of the same checks. The double-checking is necessary when dealing with code that has been compiled by somebody else, whom you possibly don't trust. To use the byte-code verifier you run your program with the `-verify` option from the command line. In future releases of the Java platform, the byte-code verifier will most likely be running by default. But for now, using the `-verify` option is highly recommended when running code compiled by others.

The JVM is responsible for checking of array bounds and the validity of object casts. Non-compliance with the former check results in a `java.lang.ArrayIndexOutOfBoundsException`. Likewise, illegal object casts end with the throwing of a `java.lang.ClassCastException` at runtime.

3.2.2 The sandbox

The security model in Java centers around the idea of a sandbox. In executing code you don't fully trust, you confine the environment in which it runs. There are two aspects of protection to think of:

1. Protecting resources outside the sandbox from the code itself, and

[2]multiple inheritance is illegal in Java

2. making sure that elements living on the outside can't manipulate the code on the inside.

Stand-alone programs in Java are by default running without the sandbox enabled. You have to turn this mechanism on explicitly by executing your application with the command-line option: `-Djava.security.manager`

Conceptually, you need the following elements to set up a sandbox in Java:

- A predefined set of sensitive actions code can perform,
- a way of binding these actions to specific segments of code, and
- a control center responsible for allowing or refusing code to perform sensitive operations.

3.2.2.1 Permissions

The abstract class `java.security.Permission` defines a permission in Java. Every class is associated with a set of permissions. By default, classes in the core Java API can perform any action. A class able to do anything is associated with the special `java.security.` \[3] `AllPermission` class. You can choose from a list of predefined permissions, such as `java.io.FilePermission` and `java.net.SocketPermission`, or you can define your own by extending the `Permission` class, or more commonly the `java.security.` \ `BasicPermission` class. A permission consists of three attributes:[4]

Type, the name of the particular Java class implementing the permission. This attribute is required.

Name, based on the type of permission. A name associated with a permission to a file is the name of the target directory or file. Many permissions don't have a name entry, e.g. instances of the `AllPermission` class.

Actions, an optional list of entries describing what can be done to the target. A file permission may specify that a file can be read, written or deleted.

Permission to read files in the directory "/Users/public/shared/" would be specified as:

```
permission java.security.FilePermission
"/Users/public/shared", "read";
```

The given permission is of *type* `java.security.FilePermission`, has *name* `/Users/` \ `public/shared`, and *action* `read`.

[3]The \ will be used throughout the thesis to indicate splitting of characters that should be considered one word.

[4]The naming of the attributes is a bit confusing and awkward. The *type* is really a name and the *name* is a type.

3.2.2.2 Code sources

A code source encapsulates information about where a class was loaded from and who signed the class. Both entries are optional. The location is specified as a Universal Resource Locater (URL), and is called the code base. In Java, the `java.security.CodeSource` class defines code sources. You may specify the URL using the `java.net.URL` class. Information about signers are encapsulated in a table of `java.security.cert.Certificate` instances. Certificate objects make a binding of an identity's credentials to a public key.

3.2.2.3 Protection domains

A *protection domain* defines a mapping between permissions and code sources, i.e. it contains information about what a code source is allowed to do. This construct glues together the ability to perform sensitive operations with specific segments of code. In terms of Java, a protection domain is an instance of the `java.security.ProtectionDomain` class. Each class can only be associated with one protection domain, and classes in the core API belong to the special system domain.

3.2.2.4 Policy files

The description of protection domains is done through the use of *policy files.* A policy file relates permissions to code sources. The JVM can use any number of policy files, but two are used by default:

Global policy file, located in `$JREHOME/lib/security/java.policy`,[5] and a

user-specific policy file, located in `${user.home}/.java.policy`.[6]

An excerpt from the global policy file:

```
grant codeBase "file:${java.home}/lib/ext/*" {[7]
        permission java.security.AllPermission;
};
```

The result is that the code source `$java.home/lib/ext` is associated with the special `AllPermission`, meaning that code from this directory can do whatever it wants.

Programmatically, an implementation of the abstract class `java.security.Policy` encapsulates the policy files in use. This is known as the system's security policy. There can only be one instance of the policy class in the JVM at any given time, but it can be replaced if the given code has permission to do so. You can specify which policy files your system should use in the `$JREHOME/lib/security/java.security` file. You can also specify the policy files you want to use directly on the command-line by using the option:

[5]$JREHOME indicates the root catalog for your Java runtime installation.

[6]${user.home} indicates the home catalog of the current user.

[7]${java.home} points to the root catalog for your Java installation.

```
Djava.security.policy=myPolicy.policy
```

The resulting policy uses the global, user-specific, and command-line specified policy files. Use two equal signs to only enable the one given on the command-line.

In summary, we now have a way to explicitly state what specific classes in Java are allowed to do through the use of permissions and code sources, and we can set up our own protection domains by creating policy files. Please note that giving code permission to perform some action doesn't necessarily mean that the environment on the executing host will allow that action, e.g., deleting critical system files will typically not be permitted, unless the user has administrator privileges.

3.2.2.5 Access controller

In determining if a class should be allowed to carry out a sensitive operation, the underlying OS and the class' policy files must be consulted. This is the job of the *access controller*, it controls the security policy of an application. Prior to the Java 2 release, this responsibility fell on the *security manager*. This mechanism still exists, in order to accommodate programs developed before the introduction of the Java 2 platform. The common scenario now, is for the security manager to defer its actions to the access controller. The controller was added to enable a more fine-grained way to customize the security policy of an application: that process is simply too hard to implement with the security manager.

The access controller is represented by the `java.security.AccessController` class. It cannot be instantiated, but contains static methods that query the state of the security policy in effect. The following method checks to see if a particular permission should be granted:

```
public static void checkPermission (Permission p)
```

An `AccessControlException` is thrown if the operation isn't permitted, otherwise the method silently returns, allowing the execution of code to continue. This decision depends on the set of protection domains on the *stack*. The stack holds references to the active classes in a thread. It's created on a per-thread basis, meaning that each thread in an application has its own stack. Fig. 3.1 illustrates the structure of the stack in place when the `checkPermission (p)` method of the `AccessController` class is called. The current permissions in effect are given by the intersection of the permissions associated with each protection domain. This association is denoted by a solid yellow arrow in the figure.

Since the system domain is an identifier for classes loaded from the core API, which can perform any operation, the important permissions are those associated with classes loaded elsewhere. To generalize, we label these as belonging the the non-system domain. Upon invocation of the static `checkPermission (p)` method, the stack is traversed from the top down, checking the protection domains in effect, as indicated by the numbered arrows in Fig. 3.1. The intersection of the protection domains determines the current security policy.

Fig. 3.2 shows the relationship between important classes in the `java.security` package when it comes to explaining how the access controller works. The `java.security.` \

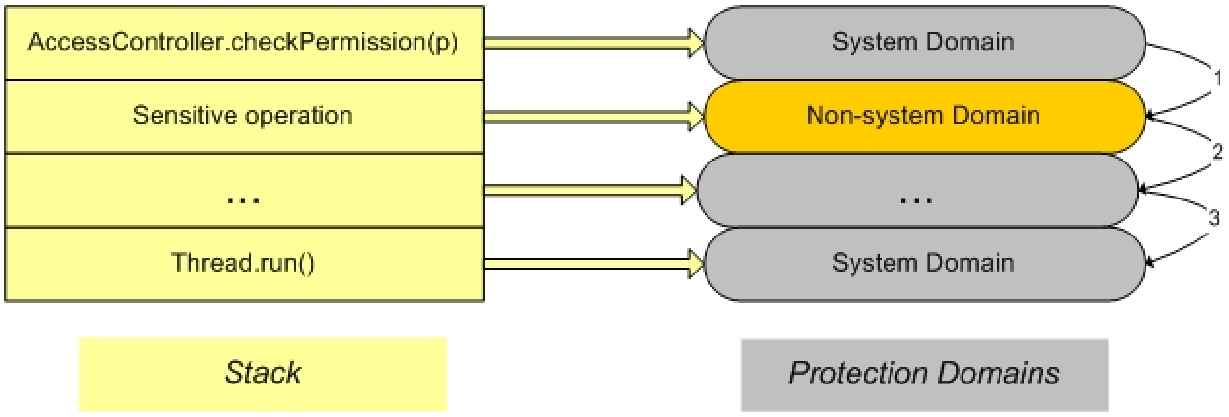


Figure 3.1: The stack and protection domains of a method

`AccessController` object holds a reference to the system's current security policy, which is encapsulated in a `java.security.Policy` object. At any given time, there's only one policy in effect for an application. The policy can be retrieved by a call to the static `getPolicy ()` method of the `Policy` class. This class specifies which permissions are available for code from various sources, i.e. it contains the different protection domains in the program. This information is held by a `java.security.ProtectionDomain` object. The result is that the access controller can find out through the system policy, which in turn consults the needed protection domain that holds information about the permissions assigned to a specific code source, if a sensitive operation can be performed. To illustrate the procedure depicted in Fig. 3.2, consider the following scenario: A programmer wants to find out if his program will be allowed to use a socket to try to connect to an arbitrary host on the Internet. The following code segment will do the trick:

```
SocketPermission perm = new SocketPermission("*:1-", "connect");
AccessController.checkPermission(perm);
```

The strings passed to the constructor of `SocketPermission` indicates ([hostname]:[port range], [action]). In our example the wildcard "*" means all possible hostnames, the character ":" separates hostname from port range, and "1-" specifies all ports from 1 and up. The value "connect" indicates that our application should be allowed to establish connections to hosts. After the call has been made to `checkPermission (perm)`, the `implies (ProtectionDomain, perm)` method of the `Policy` class is consulted to see if the given permission is contained in the current system policy. To determine if it is, the protection domain's `implies (perm)` method is called. Finally, the `PermissionCollection` object, which can hold a variety of `Permission` instances, is asked whether the initially specified permission can be matched with a permission in the collection of permissions. If yes, true is returned back through the chain of calls and the application continues silently, otherwise false is returned, ultimately causing an exception to be thrown.

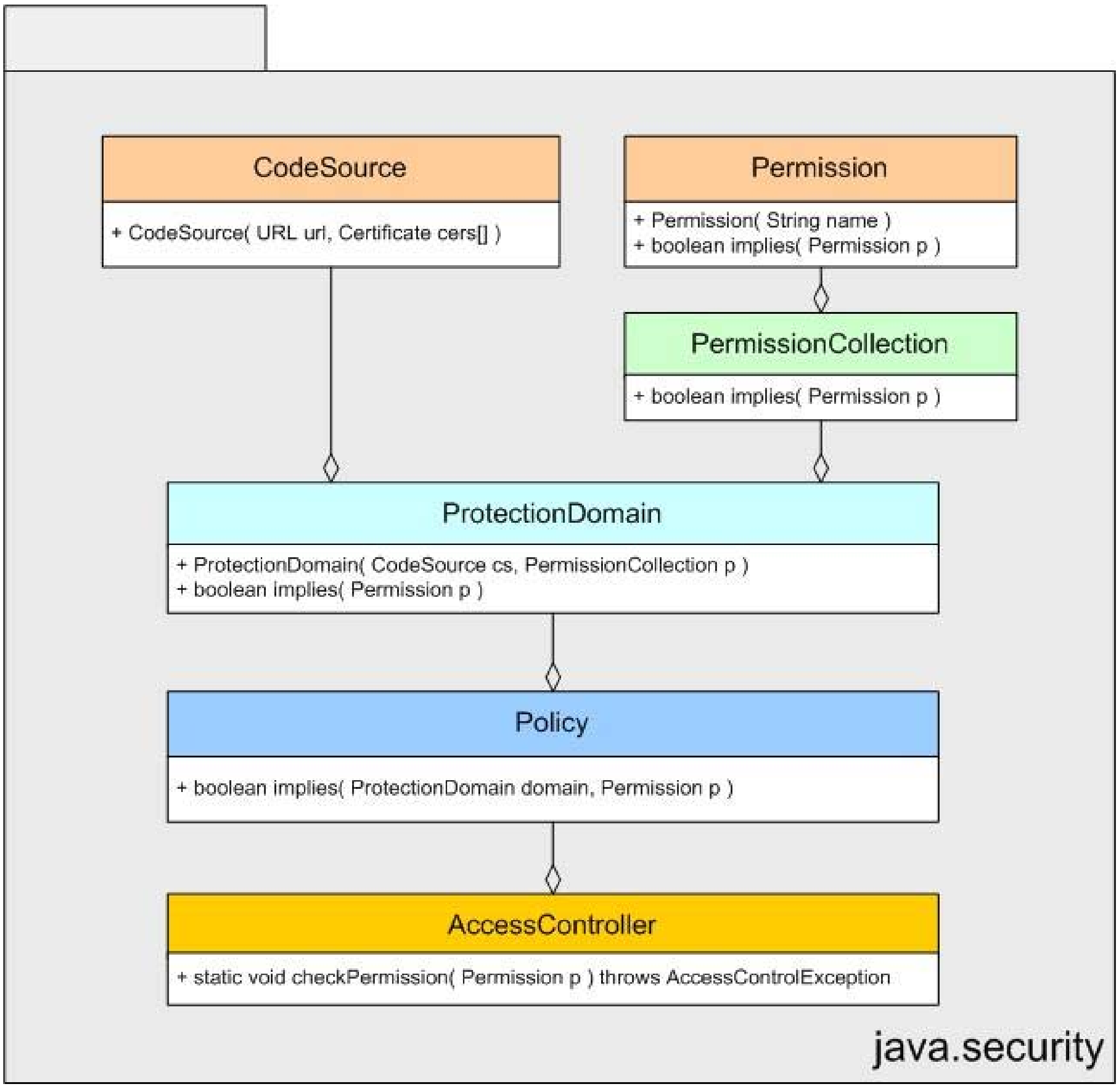

CodeSource
+ CodeSource(URL url, Certificate cers[])
Permission
+ Permission(String name)
+ boolean implies(Permission p)
PermissionCollection
+ boolean implies(Permission p)
ProtectionDomain
+ ProtectionDomain(CodeSource cs, PermissionCollection p)
+ boolean implies(Permission p)
Policy
+ boolean implies(ProtectionDomain domain, Permission p)
AccessController
+ static void checkPermission(Permission p) throws AccessControlException
java.security

Figure 3.2: AccessController class hierarchy

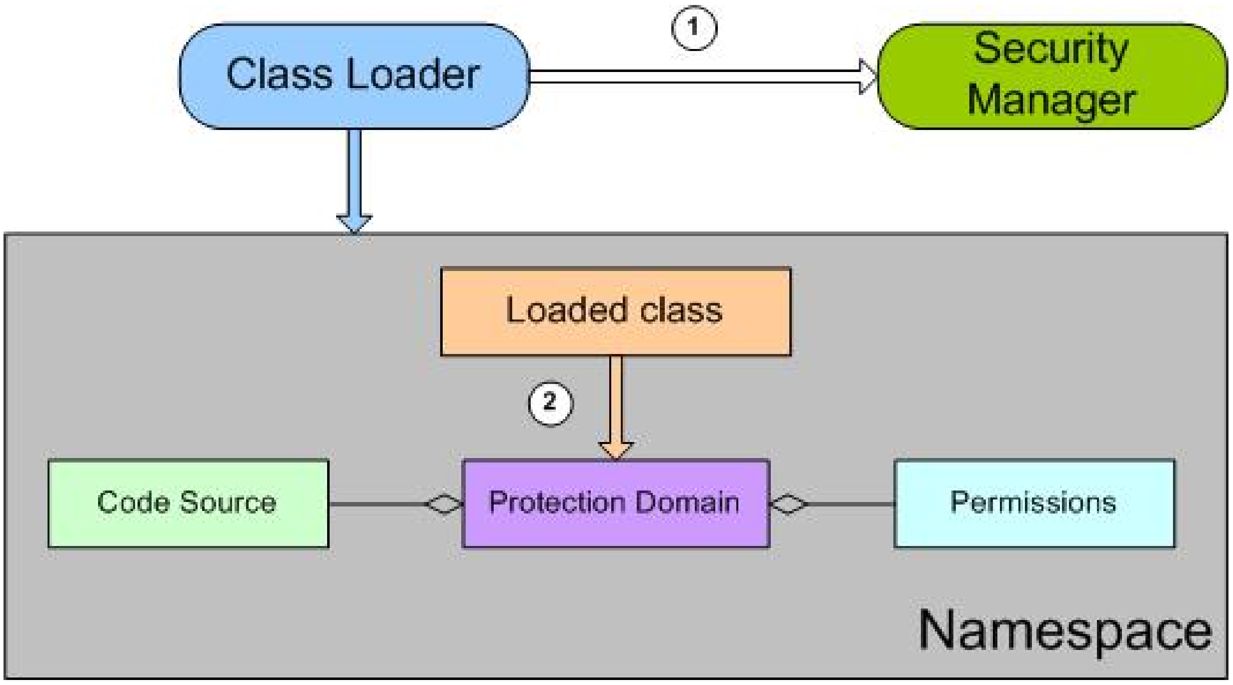


Figure 3.3: Java class loader responsibilities

3.2.2.6 Class loading

The mechanism responsible for converting Java byte-code into Java class files is the *class loader*. This operation takes place inside the JVM, and is carried out simultaneously with byte-code verification. From a security standpoint, the class loader is important because of its three duties:

1. It defines *namespaces* in co-operation with the JVM.
2. It calls the security manager to see if a particular class should be allowed to access or define classes.
3. It sets up a mapping of permissions to class objects.

Every class in Java has a unique namespace based on the package it belongs to and the name of that particular class. The Math class' namespace is `java.lang.Math`. The definition of namespaces is crucial in terms of protecting the built-in security features of the Java platform. Imagine a scenario where the JVM was unable to distinguish namespaces from each other. You could then confuse the virtual machine by supplying your own implementations of core Java API classes, potentially causing mayhem. In Java, this is resolved by requiring that code from different code bases are loaded by different instances of class loaders. This way, the security manager knows which class loader to contact for the correct versions of a given namespace.

Fig. 3.3 depicts the responsibilities of a class loader. When prompted to load a class, the class loader must consult the security manager to find out if it has the necessary permissions to carry out the operation. Failure to present valid access properties causes an exception to be thrown. This interaction is labeled 1 in Fig. 3.3. The `defineClassInPackage` and `accessClassInPackage` permissions carry the attributes needed to define or access classes.

When a class is loaded, the class loader creates the appropriate protection domain, which is a mapping of a specific code source to one or more permissions. This operation is summarized under 2 in Fig. 3.3. The necessary information needed to create protection domains is usually located in the policy files. The previously mentioned mapping lets the access controller know what classes have which permissions. This way, you can define your own security policy by writing a custom class loader, which is considered less cumbersome than implementing the `java.security.Policy` class.

Class loaders are organized in a hierarchy. The mother of all class loaders is called the system class loader,[8] and loads all the core Java API classes. It has one or more descendants, at least a URL class loader responsible for loading classes from the classpath. Class loaders should always consult their parents when prompted to load a class. This operation continuous recursively, making the primordial class loader the first in line to provide the class in question. If it isn't able to find the class, the task falls back down through the chain of calls.

Class loaders in Java are extensions of the abstract class `java.lang.ClassLoader`. However, the preferred basis for developing class loaders is the `java.security.` \ `SecureClassLoader` class. It provides additional support for defining classes with associated code sources and permissions. The `java.net.URLClassLoader` provides a complete definition of a class loader aimed at loading classes from a filesystem or an HTTP server.

3.2.3 The Java security model

Fig. 3.4 shows an overview of the Java security model. Source files come in three flavors: *core API*, *local*, and *remote*. Files belonging to the core API are converted to byte-code by a compiler at Sun Microsystems and shipped with releases of the Java Runtime Environment (JRE). This code is considered trusted and is therefore not subject to byte-code verification. Code created locally is compiled by a local compiler.[9] Remotely loaded resources are usually class files, meaning that the source files have been compiled by a remote, unknown and possibly not trustworthy compiler. To ensure that this code follows the rules of the Java programming language, these files are checked by the byte-code verifier as they're loaded into the JVM. Byte-code verification of local class files is optional, but highly recommended. The arrow from the remote to local class files in Fig. 3.4 indicates scenarios where the user downloads remote class files and stores them on a local disk, making byte-code verification an absolute necessity, unless you trust the source 100 percent. Looking again at Fig. 3.4, core API class files are loaded into the JVM by the system class loader.

[8]The system class loader also goes by the names primordial and null class loader.

[9]Distributed with the Java platform.

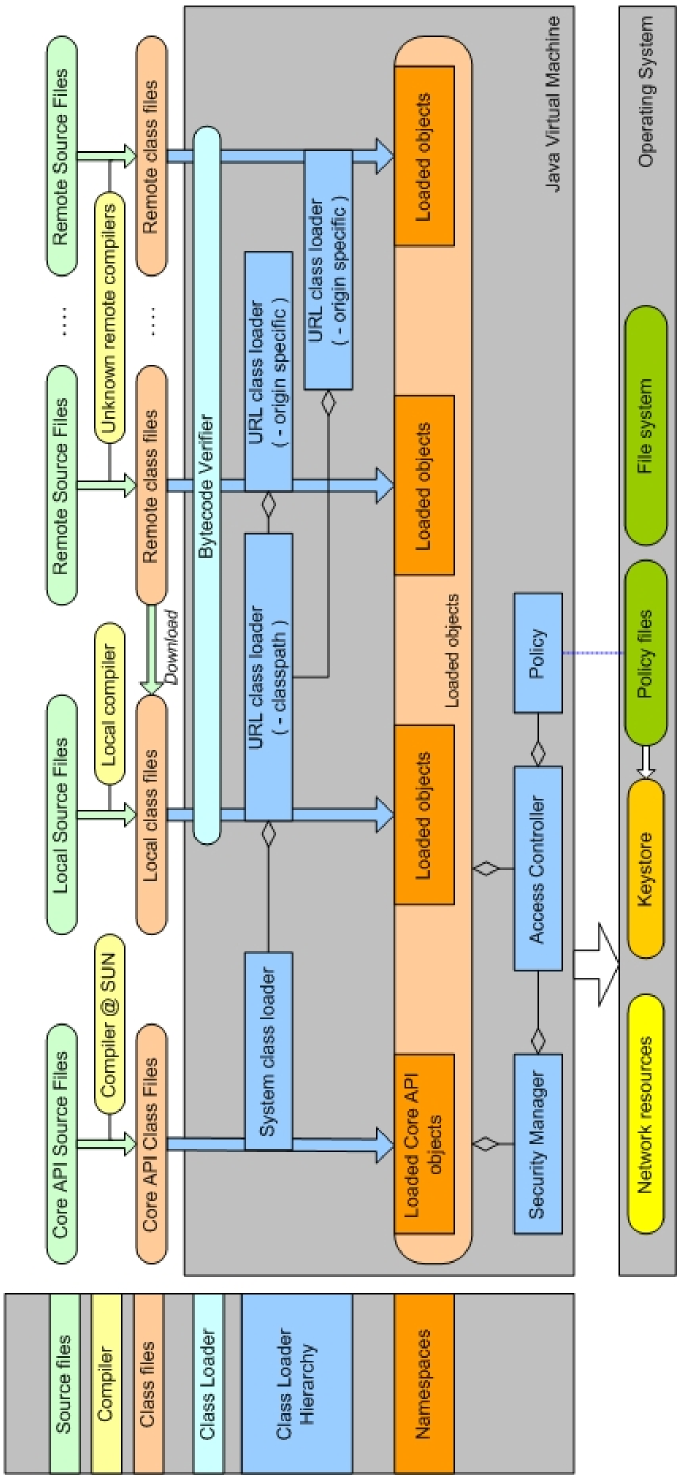


Figure 3.4: The Java security model

All other class files are loaded by class loaders further down in the hierarchy. The different loaders collaborate in the design of unique namespaces for classes that are instantiated and loaded into the virtual machine.

As the JVM is started, a security manager and an access controller are created. The policy files specified in the `$JREHOME/lib/security/java.security` file are consulted during creation of the system policy. The arrow between the JVM and the OS illustrates that the virtual machine can access resources controlled by the underlying host. This access is restricted to the rules defined for the user executing the code, meaning that someone logged into the OS with administrator privileges will be able to do whatever she pleases.

3.3 Cryptography in Java

Cryptographic services in Java are made available through the *Java Cryptography Architecture* (JCA). For an in-depth presentation of the architecture, see "Java Cryptography Architecture: API Specification & Reference" [29].

3.3.1 The framework for cryptographic services

Support for cryptographic services is realized by *engine classes* in Java. An engine is an abstraction of a particular cryptographic concept, e.g., the signature engine encapsulates the mechanism for managing digital signatures. A complete listing of the engines in Java 5.0 is given in Table 3.2.

Each engine has a corresponding Java class with the same name. In general, to use any of the engines you instantiate the given cryptographic service with a call to its `getInstance ()` method. A certain engine can be associated with numerous *algorithms*. An algorithm in this context is a "recipe" of how to create an instance of an engine. A message digest can be realized using, e.g., the MD5, SHA-1, or MD2 algorithms.

Assume that a developer wants to create a message digest using the SHA-1 algorithm. All that's needed is the following code segment:

```
MessageDigest shaDigest = MessageDigest.getInstance("SHA-1");
```

The actual implementation of the algorithm is done by a *provider.* The Java 5.0 release comes with a variety of different providers pre-installed: SUN, SunRsaSign, SunJSSE, SunJCE, SunJGSS, and SunSASL. If you have a preferred message digest provider you simply invoke `MessageDigest.getInstance(SHA-1, myProvider)`, where myProvider is your favourite provider. If you don't supply a provider, like we did in the example above, the system tries to find one for you. The file `$JREHOME/lib/security/java.security` contains a 1-based ranking of the installed providers. Excerpt from the security file:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
```

Engine	**Description**
Signature	Provides the functionality required to create a digital signature, which in turn can be used for authentication or data assurance.
CertStore	Encapsulates functionality to retrieve certificates and CRLs from a repository.
CertPathValidator	Tests the validity of certificate chains.
CertPathBuilder	Builds certificate paths.
MessageDigest	Enables message digests, which are secure one-way hash functions that take an arbitrary length input and produce a fixed length output.
AlgorithmParameterGenerator	Can be used to specify a set of parameters to be used in a certain algorithm.
SecureRandom	Provides a cryptographically strong Random Number Generator (RNG). An implementation must comply to the random tests defined by FIPS 140-2, Security Requirements for Cryptographic Modules [10].
KeyStore	Used to store cryptographic keys and certificates. More specifically: private keys, symmetric keys, and trusted certificates.
KeyFactory	Can convert opaque key objects into key specifications, and vice versa.
AlgorithmParameters	Representation of cryptographic parameters.
CertificateFactory	Factory used to generate certificates, certificate chains, and CRLs from their encodings.
KeyPairGenerator	Used to generate asymmetric key pairs, i.e. public and private key pairs.
TrustManagerFactory	Factory to create trust managers based on trusted material from e.g. a keystore.
KeyManagerFactory	Factory to create key managers based on key material from e.g. a keystore.
SSLContext	Used to specify and retrieve SSL implementations.
MAC	Provides the functionality of a Message Authentication Code (MAC).
KeyGenerator	Used for generation of symmetric keys.
Cipher	Provides the functionality to create cryptographic ciphers that can encrypt and decrypt data.
SecretKeyFactory	Factory for secret keys that can convert opaque secret key objects into key specifications, and vice versa.
KeyAgreement	Encapsulates the functionality of a key exchange protocol.
SaslClientFactory	Factory to create Simple Authentication Security Layer (SASL) clients.
SaslServerFactory	Factory to create SASL servers.

Table 3.2: Engines in J2SE 5.0

```
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
```

Assume that you invoke the static `getInstance ()` of the `MessageDigest` class without naming a provider. The system would then try to find an implementation of SHA-1 by `security.provider.1`. If the effort fails, provider number 2 is consulted, then number 3 and so on. If no provider has an implementation of the given algorithm, the system throws a `NoSuchAlgorithmException`.

The provider architecture can be extended with your own or third party implementations. If you need additional providers, see "How to Implement a Provider for the Java Cryptography Architecture" [16].

3.3.2 Cryptography APIs

In the following, the cryptographic libraries explored in the thesis are introduced. The most up-to-date resource for information on Java APIs is Sun Microsystems' Java webpages [24].

Historically, cryptography came into Java in the 1.1 release, with the introduction of the JCA. At first, it contained functionality to create and manage digital signatures and message digests. In later revisions more services were added, and presently the JCA includes support for all the engines listed in Table 3.2. More cryptographic features were added in the *Java Cryptography Extension* (JCE). The JCE API was included in version 1.4 of the Java platform. In addition, the *Java Secure Sockets Extension* (JSSE) and the *Java Authentication and Authorization Service* (JAAS) libraries rely on cryptography to offer services such as secure transport using SSL, authentication, and authorization.

3.3.2.1 Java Cryptography Architecture (JCA)

The JCA defines a generic, pluggable, and extensible framework for cryptography in Java. It was designed around two principles:

1. Algorithm independence, and
2. implementation independence.

The first is achieved through the definition of engine classes. As described in Section 3.3.1, the engine classes encapsulate the generics of some specific cryptographic service. The actual algorithm and implementation used is decided by the programmer. She has three choices: a) Do all the work herself, b) use the cryptographic implementations that come bundled with the JDK, i.e. software supplied by the installed providers, c) import third party software into the Java environment.

Implementation independence is accomplished through the provider architecture.

3.3.2.2 Java Cryptography Extension (JCE)

The JCE is an extension to JCA built upon the same provider architecture. The JCE defines a framework for advanced cryptographic services and an implementation of these. The implementation is made available through the SunJCE provider that comes pre-installed with current releases of the Java platform. The JCE addresses the following cryptographic concepts:

Encryption and decryption, encryption is the process of making data unintelligible, while decryption is the opposite task.

Password-based encryption, encryption where a key derived from a password is used.

Cipher, used to encrypt and decrypt data according to a particular algorithm.

Key Agreement, protocol to securely agree on a set of keys to be used in subsequent communications.

Message Authentication Code (MAC), a keyed hash function used to assure data integrity.

Various techniques from the JCE will be used as needed throughout the remainder of the thesis. For a detailed description of the workings of JCE, see “Java Cryptography Extension: Reference Guide” [27].

3.3.2.3 Java Secure Sockets Extension (JSSE)

The SSL protocol is the most widely used mechanism to secure transmissions of data over the Internet. It was designed by Netscape Communications Corporation in the mid-90s, but its development and maintenance was taken over by the Internet Engineering Task Force in 1996. Now in version 3.1, SSL has been renamed to Transport Layer Security (TLS). Every reference to SSL in this thesis should be understood as SSL version 3.1. As the name indicates, the protocol offers secure transport of data, and can be used in conjunction with application level protocols, e.g. the Hypertext Transfer Protocol (HTTP), or the File Transfer Protocol (FTP). TLS provides the following services:

Mutual Authentication, the server and the client can establish trust through the use of digital certificates.

Data Encryption, the data passed between the communicating parties is made unintelligible by using an encryption algorithm. This ensures privacy and confidentiality.

Data Integrity, changes in the data sent over the communication channel are detected. This service also guards against replays of old messages.

JSSE is a framework for implementation of SSL in Java. JSSE supports SSL v2.0, v3.0, and TLS v1.0. Due to the security problems identified in the earlier versions of SSL, this thesis will only focus on TLS.

JSSE will be used in Chapter 5. For more information on SSL, see "Java Secure Sockets Extension (JSSE): Reference Guide" [28].

3.3.2.4 Java Authentication and Authorization Service (JAAS)

JAAS provides subject authentication and authorization. In JAAS, the subject (user or service) running the application is authenticated. Authentication is based on a Java implementation of the Pluggable Authentication Module framework, which separates the application code from the underlying authentication. Replacing authentication technologies boils down to plugging in the desired solution. The application code itself remains unchanged.

Section 3.2.2.5 stated that access control is determined by the code source of the executing code. More specifically, the location and the signer of the code. In JAAS, upon completion of the authentication procedure, the authenticated subjects are associated with code sources. Any subsequent authorization is determined by the permissions belonging to the requesting subject. So instead of checking the code source of the application code, the code source of an authenticated subject is consulted.

For more information, please see "User Authentication and Authorization in the Java Platform" [57].

3.3.3 Java key and certificate management

Proper management of keys and certificates is vital for the proper functioning of PKI schemes. The class `java.security.KeyStore` encapsulates the concept of a keystore. Its purpose is to hold cryptographic keys and public key certificates. The store distinguishes between three types of entries:

1. Private key entry,
2. symmetric key entry, and
3. trusted certificate entry.

Private keys are stored together with the corresponding certificate chain. Both private and secret keys can be stored in encrypted form. Trusted entries are meant to hold certificates belonging to entities you have authenticated, i.e. people or organizations you trust.

The keystore provides functionality to store the entries through the `store(OutputStream stream, char[] password)` method. Note that the store doesn't dictate a specific form of storage, it is left to the developer to decide which medium to use. Access to the keystore is protected by a password. Yet another password can optionally be applied to safeguard private and secret keys in the store. Compromise of a private or secret key is critical for the victim using PKI services, and secure storage of these items is therefore extremely important.

3.4 Summary

The **Public Key Infrastructure** relies on cryptographic primitives to provide **authentication**, **integrity**, and **confidentiality**. Built-in features in the Java platform provide a safe computing environment for developers. Central components include the **access controller**, **byte-code verifier**, and **class loader**. In Java, a variety of **engines** provide access to cryptographic services. The **JAAS** and **JSSE** APIs enable use of authorization procedures and **TLS**, respectively. Access to certificates and keys is given through a **keystore**, that can hold **private key**, **symmetric key**, and **trusted certificate entries**.

Armed with the theoretical background on PKI and an overview of the security mechanisms of the Java platform, we're in position to weld the two together and bring security-minded development to Internet banking.

Chapter 4

Towards More Secure Internet Banking

In this chapter, present Internet banking solutions are discussed with an emphasis on security issues. Next, PKI is debated in the context of banking systems. Current initiatives and challenges are presented. The chapter continues with an expansion of the application code given in Chapter 2, where PKI is used to provide a framework for security services. The main focus of the chapter is to familiarize readers with important basic operations in a PKI. These include generation of keys, certificates and CRLs, and distribution of the created material. An example of how to realize mutual authentication using the output from the basic operations is also included. In closing, present PKI challenges are debated.

4.1 Security in Today's Internet Banking Systems

Selmersenteret, a subdivision of Department of Informatics at University of Bergen, has evaluated the security in some of the largest Norwegian Internet banks. In the resulting report [14], the research group recommends that a number of banks should change their procedures for customer login. The problem with current practices is the authentication of clients through publicly available usernames, such as social security numbers, in combination with short PINs. In particular, these systems seem to be very vulnerable to a distributed DoS attack joined with a brute-force PIN cracking scheme.

4.1.1 Improving authentication procedures

One way to improve security against brute-force attacks in Internet banking is to introduce longer PINs. Each new digit added reduces an attackers chance of success by a factor of 1/10. On the downside, users will have a harder time remembering longer PINs. A more serious threat, independent of the length of the PIN, is that as the bank's customer base expands, the odds are changing in favor of the hacker. This is due to the nature of the attack outlined in the before-mentioned report [14].

Another improvement, which also would help against DoS attacks aimed at individual account holders, would be to conceal usernames of clients. One way to achieve this is to have account holders pick their own credentials. However, customers tend to choose usernames that are easy to memorize, which often translates to easy to guess for an adversary. A variant of the scheme is to have the bank generate random login names. The approach forces users to remember two pieces of information, which isn't optimal as clients then become more likely to write down and store their credentials. For a discussion on choosing good passwords, see "Password Memorability and Security: Empirical Results" [3].

4.1.2 The banking security paradigm

Norwegian banks' approach to security is that of total secrecy. No breaches or attempted attacks against their systems are reported to customers. In fact, clients are told that their financial institution's security system is infallible. In "Why Cryptosystems Fail" [2], Ross Anderson describes several cases of poorly designed and implemented security solutions in British banks. To assume that the problems identified in Anderson's article are limited to the English banking industry would be at best naive. The study conducted by researchers at Selmersenteret mentioned above suggests the contrary.

A shift of paradigm seems to be underway in the banking industry [2]. Traditionally, services have been developed in a military fashion, minimizing communication and learning even between banks, the result being that successful attacks against one system could be conducted with the same outcome at a later time against a different institution. In 1993, several trends indicated an upcoming fusion of security and software engineering [2]. Total secrecy would be replaced by an iterative development process with input from other financial institutions and the public, ultimately leading to the production of more robust systems.

Modern cryptographers believe that "there is no security through obscurity." The history of the discipline repeatedly shows how secret systems have been broken due to unknown vulnerabilities in the design. This goes all the way back to the earliest substitution ciphers that were easily broken by exploiting the relative frequency of letters. New cryptographic protocols are introduced with a detailed description of their inner workings, promoting public review and analysis. The issuance of new standards is carried out by organizations such as the National Institute of Standards and Technology (NIST) and the Internet Engineering Task Force (IETF).

4.1.3 BankID

The Norwegian banking industry is currently working on a PKI project named BankID [4]. Its goal is to deliver authentication and digital signature services to bank customers. As of March 2005, the only publicly available technical information on the project is a 38 page long document on the system's policy for personal certificates [38]. In the introduction it is explicitly stated that descriptions of security and technical details are restricted. Later in the text, readers are informed that permission to read further documentation can be given on a "need-to-know" basis. It turns out that individuals can get more information by signing a non-disclosure agreement, making them legally incapable of reporting their findings.

While experts in the crypto community insist that developers publish their security protocols, the BankID team has chosen not to involve independent security researchers in the process. This is alarming due to a number of reasons:

- According to research conducted by Ross Anderson [2], most failures in cryptosystems are caused by management and implementation errors. The introduction of software engineering processes similar to those used in the design of safety critical systems can increase learning between projects and result in more robust applications.
- Systems designed for the Internet leak information. Hackers can often gather insight about web applications by using the software in ways not intended by the system designers. Most likely, attackers learn less about the application from a public review than others, as they would get a great deal of this information through various information retrieval techniques anyway.
- BankID aims to support non-repudiation. This is a complex and difficult service to provide. In order for it to function properly, you would need a secure time server and a secure data storage facility approved by all members of the PKI. Non-repudiation can't be handled in software alone, a human presence is needed in solving disputes. At the very least, the BankID team must inform its users on how secure time stamping is achieved, how information is stored securely, and clarify who that is to decide the outcome of future disagreements.
- A central part of any PKI is trust. Key components in the infrastructure must be recognized by all members of the PKI. BankID does not include an explicit trust model.
- In an interview with Dagbladet [8], the coordinator of BankID is quoted saying that BankID is introduced to offer more services, and is not intended to strengthen the security of current Internet banking solutions. A PKI is primarily a security infrastructure, that when employed correctly can be used to deliver a wide range of services to its members. The security foundation has proved to be very difficult and complex to implement, some common pitfalls are pointed out through case studies in "Planning for PKI" [15]. BankID's service-oriented approach could very well end up weakening Internet banking security, through sacrificing security to user friendliness.

4.2 Shortcomings in the Initial Internet Banking Scheme

The code developed in Section 2.3.2 provides a communication framework which can be used for Internet banking. A specialized protocol was designed to show how users can supposedly authenticate themselves to the bank. By itself this scheme fails to deliver the primary security services. Data in transit from an end user to the bank, can easily be read, altered, or intercepted by anyone controlling an intermediate node in the network.

This is also a recognized fact in the banking community. Their solution is to use the SSL protocol with server-side authentication. Typically, the bank purchases a certificate from Verisign,[1] that it uses to authenticate their web-site in subsequent sessions with customers. The underlying assumption is that everyone should trust Verisign, since the company is a large commercial provider of certificates. It should be noted that Skandiabanken has implemented a solution that includes client certificates. The problem with their system is the distribution of these certificates. Users get these through weak client authentication procedures. So instead of strengthening security, Skandiabanken has included a new layer of complexity.

In current Internet banking solutions it is assumed that only a legitimate user can obtain the credentials asked for at login. This assumption isn't correct, a fact established in Section 4.1. Employing SSL with server-side authentication does not improve the weak client authentication. An additional concern is the growing amount and sophistication of computer viruses that are spread throughout the Internet. Malicious software can monitor legitimate bank users' computers, and use the acquired information to access customers' accounts.

A step in the right direction would be to introduce strong client-side authentication in today's Internet banking systems. This solution, in combination with a tamper-proof smartcard-reader setup, can effectively strengthen login procedures and thwart viral attacks. The key here is realizing that sensitive operations should not be performed on a device that is connected to the Internet. Private key computations should be carried out on an offline device, and the results transferred to an on-line computer. A way to make this transfer is to make the user manually input the computed values. Section 5.4.2 contains a discussion of the suggested scenario. Union Bank of Switzerland (UBS) started testing a system based on the above-mentioned approach in 2004. More details on the design can be found in "Secure Internet Banking Authentication" [12]. Mutual authentication requires each client to have their own certificate and corresponding private key, which introduces a key and certificate management problem. An underlying infrastructure, such as the PKI, can deal with the new administrative challenge.

4.3 PKI for Internet Banking

The following subsection describes a PKI implementation in Java. Important methods and segments of code needed to realize a PKI are presented, and tips and hints are given to developers of PKI software. The framework will later be used to offer mutual authentication of clients and servers.

[1]The largest Norwegian Internet banks all use certificates from Verisign.

4.3.1 The scalable solution

Fig. 4.1 presents the textbook solution in designing a PKI. Different tasks are distributed among components first introduced in Section 3.1.1 to facilitate analysis and management. To become a member of the PKI you need to register with the RA. This process is exemplified in Fig. 4.1 by the arrows from the customer and bank entities to the RA. The information supplied is then verified and signed by the RA and sent to the CA, which in turn produces the actual certificates. Status information on issued certificates is kept in the archive. Any updates regarding this status are time stamped and sent for long-term storage as indicated by the arrow from the CA to the archive in Fig. 4.1. Next, the certificates are signed by the CA and sent to a repository for further distribution. The signing procedure is denoted $\mathrm{KR}_{\mathrm{signer}}(\mathrm{Cert}_{\mathrm{entity}})$, where KR is the private key belonging to the *signer* and Cert is the certificate issued to *entity*. As noted in the figure, the repository sends the final certificates to users. In addition, the repository can also be queried to obtain certificates of other users in the PKI.

4.3.2 Java Implementation

4.3.2.1 Assumptions

The CA is an independent third party provider, meaning that neither the bank nor any customers control the PKI architecture. In the eyes of the CA, the bank and customers are treated as equal members of the infrastructure. This setup is somewhat controversial, as current practice in Internet banking is to let the bank control the CA. A PKI with an independent CA has the benefit of being useful in other settings than banking. All members can use the infrastructure to communicate with each other securely. If the bank is running the CA you should only use the PKI for services you trust the bank to perform. An independent CA can be used to support services for society at large.

The question is whether or not it is realistic to assume the existence of a CA that everyone can trust. Maybe governments can be trusted to issue electronic IDs to its citizens and companies? You could argue that it would be just like issuing passports and business enterprise organization numbers, a job governments around the world are trusted to perform. As a cautionary note, keep in mind that a forged handwritten signature is fundamentally different from a forged electronic signature in that the latter is identical to a valid signature [52]. This is important to realize because someone in possession of your private key will be able to do a virtually undetectable impersonation of you. Therefore, any chosen CA should not be trusted with keys that are to be used for electronic signatures. Also, there's the question of cost. Someone has to finance the operation of the PKI. This theme is debated in Section 4.4.

The discussion above leads us to the next assumption: users must generate their own keys. Please see Section 5.4.1 for a discussion on management of keys and certificates.

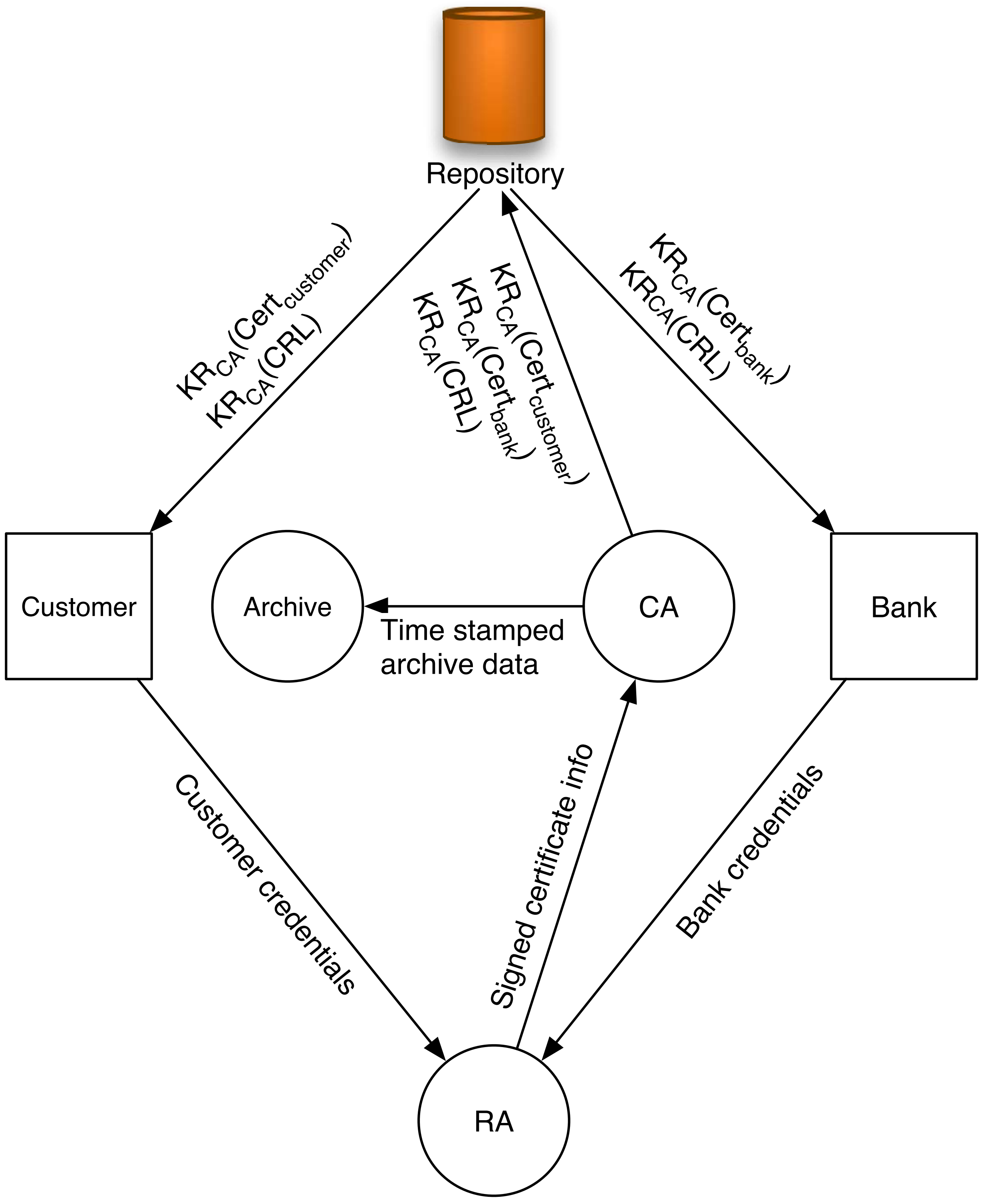


Figure 4.1: PKI architecture

4.3.2.2 Configuration details

In the following, a strict hierarchy of CA's will be used as the underlying trust model. In fact, the PKI contains only one CA, given by the DN: {CN=Certification Authority, OU=NoWires, O=UiB, L=Bergen, ST=Hordaland, C=NO}. All members of the PKI trust this single authority, and will only accept certificates issued by that entity.

The responsibilities of the repository and the registration authority will be fulfilled by the CA. The size of the PKI we're about to implement is so small that there's no reason to distribute the workload to more than one component. The archive will not be considered, as it is not an essential part in providing the primary security services. The operations that will be addressed are

- Key generation,
- X509 certificate generation,
- CRL generation,
- distribution of certificates and CRLs, and
- mutual authentication between members of the PKI.

These services are built on top of the communication framework presented in Chapter 2. A complete listing of the added source code is given on the author's web site [7]. The code is meant to run under a security manager, enabled on the command-line at startup by including the option `-Djava.security.manager`.

4.3.2.3 The Legion of the Bouncy Castle

The current version of Java, J2SE 5.0, does not support generation of X509 certificates and CRLs. They can both be instantiated from a data source, but cannot be created from scratch. The Legion of the Bouncy Castle [54] has created a lightweight crypto API that enables generation of certificates and CRLs. The software package can be used, copied, and modified free of charge. On the downside, it is extremely poorly documented. This basically renders the code useless in the development of secure systems. If it were to be used in such a setting, one would have to put time and effort into examining and documenting Bouncy Castle's source code, which can be downloaded from their site. Currently, the API consists of about 800 Java classes. Turning the whole package into production quality code is a very time consuming task, but the Bouncy Castle API can be used as a reference in developing your own crypto libraries. The makings of such a library falls outside the scope of this thesis, instead we'll use Bouncy Castle's APIs as needed. It should be stressed that this is highly discouraged in a business context. Usage of the Bouncy Castle API will be marked explicitly in code segments and pointed out in the text.

4.3.2.4 Key generation

The first step towards becoming a member of the PKI is to register with the CA. In addition to proving your identity, one must supply a public key to be included in the certificate. The applicant must create a keypair prior to the registration process. You should not trust somebody else to generate keys for you, as it is of utmost importance that nobody else learns your private key. Failure to do so can potentially result in others misusing your credentials and privileges in the PKI.

Java provides two ways to generate keys:

Keytool, a command-line utility distributed with the JDK [32]. Primarily designed to ease key and certificate management.

Programmatically, using the `java.security.KeyPairGenerator` class.

Example key generation using *keytool*:

```
keytool -genkey -alias alice -keystore alice -keyalg rsa

Enter keystore password:  o5BIwt

What is your first and last name?  [Unknown]:  Alice Johnson
What is the name of your organizational unit? [Unknown]:  N/A
What is the name of your organization?  [Unknown]:  N/A
What is the name of your City or Locality?  [Unknown]:  Wonderland
What is the name of your State or Province?  [Unknown]:  IL
What is the two-letter country code for this unit?  [Unknown]:  WO

Is CN=Alice Johnson, OU=N/A, O=N/A, L=Wonderland, ST=IL, C=WO \
correct?  [no]:  yes

Enter key password for <alice>        (RETURN if same as keystore \
password):
```

The first line starts the program in key generation mode, as indicated by the option `-genkey`. For an explanation of keytool parameters, please consult the manual [32]. The user is then prompted to select a password for the newly generated keystore. Next, keytool collects the necessary information to construct a DN. The last step is to set a password for the private key, meaning that the security of the key relies on two user-supplied passwords. They consist of at least 6 characters. The designers of keytool recognize the difficulty of remembering 2 passwords, and therefore suggest that you use the same password both for the keystore and the private key. It should be noted that 6 alphanumeric characters can be brute-forced in 2^{36}computations, meaning that a longer password is desirable. In an environment where multiple users share the same keystore, the private keys should be protected by different passwords.

Key generation with the `java.security.KeyPairGenerator` class:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024, SecureRandom.getInstance("sha1prng") );
KeyPair kp = kpg.generateKeyPair();

PublicKey pubKey = kp.getPublic();
PrivateKey privKey = kp.getPrivate();
```

First, the key generation algorithm is specified through the static `getInstance(String algorithm)` method of the `java.security.KeyPairGenerator`. Next, the newly generated object is initialized with a key size and source of randomness. The key pair is generated by a call to the `generateKeyPair()` method. After completing this step, the programmer can extract the individual keys by calls to the `getPublic()` and `getPrivate()` methods of the `java.security.KeyPair` class.

4.3.2.5 X509 certificate generation

Upon confirming the identity of the applicant and receiving the corresponding public key, the CA can issue a new certificate. X509 version 3 certificates can be created using the `X509V3CertificateGenerator` class provided by Bouncy Castle. The following code describes a method to generate certificates:

```
public X509Certificate createCertificate (X509Name DN,
            PublicKey publicKey, PrivateKey privateKey,
            BigInteger serial) {

    X509Certificate cert = null;

    // Bouncy Castle class
    X509V3CertificateGenerator certGen = new
                                 X509V3CertificateGenerator();

    // Bouncy Castle class
    certGen.setIssuerDN(new X509Name(
                    "CN=Certification Authority," +
                    "OU=NoWires, O=UiB, L=Bergen, ST=Hordaland," +
                    "C=NO"));

    // Expires in approx. 10 years
    Calendar cal = Calendar.getInstance();
    cal.set(2015, 6, 30);
    certGen.setNotAfter(cal.getTime() );
    certGen.setNotBefore(new Date() );
```

```
        certGen.setPublicKey(publicKey);
        certGen.setSerialNumber(serial);
        certGen.setSignatureAlgorithm("sha1withrsa");
        certGen.setSubjectDN(DN);

        cert = certGen.generateX509Certificate(
                    privateKey, SecureRandom.getInstance("sha1prng") );

        return cert;
    }
```

Prior to instantiation of the certificate, a number of fields have to be set in different set methods. The programmer specifies a DN for the issuer, time-window for certificate validity, the subject's public key, a unique serial number, the key signature algorithm, and the DN of the certificate owner. The two DN strings are constructed by using Bouncy Castle's `X509Name` class. A call to the `generateX509Certificate(PrivateKey key, SecureRandom random)` method of the `X509V3CertificateGenerator` creates the actual certificate. The private key used belongs to the CA issuing the certificate, and creates a digital signature protecting the integrity of the certificate.

4.3.2.6 CRL generation

The CA must also create CRLs, which can be accomplished through Bouncy Castle's `X509V2CRLGenerator` class. Initially, the CA creates an empty CRL:

```
    // Bouncy Castle class
    X509V2CRLGenerator crlGen = new X509V2CRLGenerator();

    // Bouncy Castle class
    crlGen.setIssuerDN(new X509Name(
                    "CN=Certification Authority," +
                    "OU=NoWires, O=UiB, L=Bergen, ST=Hordaland, C=NO"));
    crlGen.setThisUpdate(new Date() );
    crlGen.setSignatureAlgorithm("sha1withrsa");

    X509CRL crl = crlGen.generateX509CRL(CertificationAuthority.privKey);
```

Before producing the list, the issuer DN, the current date, and the signature algorithm fields are set. The CA's private key, indicated in the code by the class variable `Certification` \ `Authority.privKey`, is used to sign the CRL.

Revoking a certificate involves creating a new list. The contents of the current revocation list must be copied onto the new CRL, along with the new entry. The following method enables revocation, where the current CRL is assumed to be contained in a class-variable named `crl`:

```
private void revokeCertificate (BigInteger serial) {
    // Retrieve the current list and add new entry
    Iterator revokedCerts = crl.getRevokedCertificates().iterator();

    // Bouncy Castle class
    X509V2CRLGenerator gen = new X509V2CRLGenerator();
    gen.setIssuerDN(CertificationAuthority.CA);

    while(revokedCerts.hasNext() ) {
        BigInteger revokedSerial = ( (X509Certificate)
        revokedCerts.next() ).getSerialNumber();
        gen.addCRLEntry(revokedSerial,
        crl.getRevokedCertificate(revokedSerial).getRevocationDate(), 0);
    }
~
    gen.addCRLEntry(serial, new Date(), 0);

    // Replace current revocation list with the newly generated
    crl = gen.generateX509CRL(CertificationAuthority.privKey);
}
```

The certificates are distinguished through their unique serial number within the CA. In the first part of the method, the current CRL is retrieved and put into an iterator. The while loop goes through the iterator, copying the old CRL into a new list. Next, the new entry is added. The method `addCRLEntry(BigInteger certificateSerialNumber, Date revocationDate, int reason)` takes as input the serial number of the certificate to be revoked, the time of revocation, and a reason for the revocation. The last element is ignored in the implementation, indicated in the code by passage of the 0 value. The new CRL is then generated and signed with the CA's private key.

Since adding an entry to the CRL means that you have to create a new list, the complexity of the operation is $O(n)$. Also, the signature operation can be time consuming. In a PKI where the CRL is required to be as fresh and up-to-date as possible, and with a large community of users, the workload in managing the CRL can become substantial.

4.3.2.7 Distribution of certificates and CRLs

A number of protocols can be used to successfully distribute certificates and CRLs to customers. Examples include the X500 Directory, the Lightweight Directory Access Protocol (LDAP), FTP, HTTP, and electronic mail. These have different strengths and weaknesses. The selection of a specific protocol depends upon what you want to achieve. It's important to realize that in most cases you won't need a trusted repository. The certificates and CRLs are both signed with the private key of the issuing CA, meaning that the integrity of the information is protected. No one can modify or insert valid certificates and/or CRLs in the repository without access to the CA's private key. An attacker can achieve denial of service by modifying or inserting bogus data in the repository, but cannot make users trust invalid information.

The implementation given on the author's web site [7] loads certificates and CRLs from the local filesystem. The code executes under close attention of a security manager, meaning that any potentially sensitive operation must be authorized in a policy file. Lets assume that Alice and Bob want to communicate. Alice has stored the following information:

- The CA's certificate in keystore `/PKI/CA.ks`,
- Bob's certificate in keystore `/PKI/Bob.ks`,
- a recent CRL in file `/PKI/crl`, and
- her own certificate in keystore `/PKI/Alice.ks`.

She would now need to specify this set of permissions:

```
permission java.io.FilePermission "/PKI/CA.ks", "read";
permission java.io.FilePermission "/PKI/Bob.ks", "read";
permission java.io.FilePermission "/PKI/crl", "read";
permission java.io.FilePermission "/PKI/Alice.ks", "read";
permission java.net.SocketPermission "*:1024-", "accept, connect,
                                      listen, resolve";
```

The `SocketPermission` allows Alice to set up a communication link with Bob.

4.3.2.8 Mutual authentication between members of the PKI

In the following we assume that a bank customer and a bank server have obtained their own certificates through a common CA, the CA's public key, and a recent CRL. These will all be loaded from file on the local system. Certificates are loaded from a file-based keystore. The CA's public key must be obtained in a secure fashion, e.g., meeting with the CA. The grade of security depends on the services you require from the PKI. Some may be comfortable with fetching the public key from the CA's Internet site. When you have acquired the key, certificates and CRLs can be gathered using one or more of the protocols mentioned in Section 4.3.2.7.

We're now in position to show how two members of the PKI can achieve mutual authentication. The client-server implementation from Section 2.3.2 is still used, but the application logic is replaced with the following protocol:

1. The customer sends its certificate to the bank server.
2. The bank tries to verify the customer certificate:
   (a) If successful, it sends its certificate to the customer.

   (b) If unsuccessful, it terminates the current session.

3. The customer tries to verify the bank certificate:
   (a) If successful, the customer can initiate secure communication using public key cryptography.
   (b) If unsuccessful, the bank certificate is discarded and the connection closed.

The protocol is essentially symmetric: the server and client both have to do the same operations. In Section 2.3.2, common features and methods for the client and server were put into the abstract `Communicator` class. We continue this practice and therefore add functionality to load CRLs from file, load certificates from a file-based keystore, and verify certificates to `Communicator.java`.

Instantiating CRLs from file

CRLs can be instantiated by a `CertificateFactory`. Through its static `getInstance(String type)` method, a factory supporting X509 CRLs can be created. Currently, X509 is the only certificate type implemented by the pre-installed providers in Java. So if you need a different type you're on your own. The following code segment loads a CRL from file:

```
CertificateFactory cf = CertificateFactory.getInstance("X509");
FileInputStream fis = new FileInputStream(filename);
X509CRL crl = (X509CRL)cf.generateCRL(fis);
```

The variable `filename` identifies the file holding the CRL data. The key method to retrieve the CRL is `generateCRL(InputStream inStream)` of the `CertificateFactory` class.

Loading certificates from a file-based keystore

A keystore is created through the static `KeyStore.getInstace(String type)` method. The default type is JKS and can be used to store asymmetric keys and certificate entries. If you need to store symmetric keys, specify JCEKS instead. Before you can extract data from the store, you have to initialize it with location and password. This is done by invoking the `load(InputStream stream, char[] password)` method of the `KeyStore` class. The code below instantiates a certificate from a file-based keystore:

```
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream(keyStore), pass);
Certificate[] certs = ks.getCertificateChain(alias);
X509Certificate cert = (X509Certificate) certs[0];
```

The certificates are stored in a certificate chain, represented as a table of type `Certificate`, in the store. The different chains are uniquely identified by aliases. In our PKI model, where the users trust the same CA, we only have certificate chains of length 1. The certificate is therefore acquired from the first (and only) index in the table.

Verification of certificates

The certificate presented by another user in the PKI must be verified before it can be used as a basis for secure communication. Specifically, the user must inspect the time-window, a recent CRL, the binding between the DN and the certificate, and verify the digital signature of the CRL and certificate. In the following, the bond between the DN and actual certificate will not be considered. The bank will probably have a list containing a mapping of DNs to certificates. The client is asssumed to obtain the bank's certificate when opening an Internet banking account. The repository can be queried to discover the identity credentials belonging to a given certificate. The method `checkCertificate(X509Certificate certificate)` implements the verification steps:

```
protected boolean checkCertificate(X509Certificate certificate) {
    boolean validity = true;
    try {
        certificate.checkValidity();

        X509CRL crl = loadCRL("/PKI/crl");

        if(crl.isRevoked(certificate) ) {
            validity = false;
        }
        // Verify that the certificate and CRL both were signed by
        // the CA
        X509Certificate caCert = loadCertificate(
                                   "/PKI/CA.ks",
                                   "alias", "password".toCharArray() );
        PublicKey caPubKey = caCert.getPublicKey();
        certificate.verify(caPubKey);
        crl.verify(caPubKey);
    } catch (Exception e) {
        validity = false;
        e.printStackTrace();
    }
    return validity;
}
```

The boolean flag `validity` indicates the current status of the verification process, where true represents successful verification. To find out if the certificate is currently within the time-window set when issued, the `checkValidity()` method is used. If outside the time-interval, this method throws a `CertificateExpiredException` or a `CertificateNotYet` \ `ValidException`. In either case the `validity` flag is set to false. The catch clause in the code above shows this technique, where the mother of all exception classes is used to catch any exception thrown. The application code on the author's web site [7] handles exceptions individually, not presented above in order to spare the reader from boring technicalities. Next, the CRL is loaded and inspected to determine whether or not the certificate has been revoked. The last step in the verification procedure is to check the digital signature on the certificate and the CRL. The `verify(PublicKey key)` method of the `Certificate` and `X509CRL` classes check the integrity of the data. These methods throw exceptions, in the same manner as with the `X509CRL.isrevoked(X509Certificate certificate)` method, if they're unable to verify the signature.

Upon successful completion of the protocol, both parties have acquired a valid certificate of the entity identified in the subject field. However, neither of them can be sure that this is the same entity as the one they got the certificate from. A challenge/response protocol can establish such a binding.

Proving possession of the corresponding private key: challenge/response

Anyone can obtain a valid certificate from a repository and use it to initiate a session with the bank. It's also possible to masquerade as the bank by intercepting connections. To solve this problem you can use a challenge/response protocol. This is a common basic authentication technique, and a more thorough treatment can be found in Stallings [51] or Mao [35]. The idea is to send a random challenge to the party with whom you are communicating, and have him digitally sign the challenge. This way you can prove possession of the private key matching the given certificate. The challenge must be random to prevent replay-attacks, i.e. someone replaying messages exchanged in previous communications.

After finishing the initial certificate exchange, the following challenge/response protocol is run:

1. The customer sends a random 20-byte challenge to the server.
2. The bank generates its own 20-byte challenge, and signs the challenge issued by the customer. Both values are sent to the customer.
3. The client attempts to verify the bank's signature:
   (a) If successful, it signs the server-challenge and sends it back to the server.
   (b) If unsuccessful, it terminates the session.
4. The server tries to verify the client's signature:
   (a) If successful, both parties have proven possession of the alleged private key and secure communication can start.
   (b) If unsuccessful, the bank terminates the session.

Implementation details are given on the author's web site [7]. Messages can now be exchanged securely through asymmetric encryption/decryption. If the client wants to send a message to the server, he encrypts the plaintext with the bank's public key. The resulting ciphertext can only be decrypted with the corresponding private key, which is only known to the bank. Asymmetric operations are time consuming, a faster approach is for the parties to exchange a symmetric key. Designs of key agreement protocols can be found in Stallings' [51] textbook on cryptography.

4.4 PKI Challenges

A comprehensive PKI can provide a wide range of security services. In terms of new opportunities, the service of non-repudiation is very interesting. Employed correctly, this mechanism effectively binds a public key, which in turn is bound to a identity's credentials, to an action. The nature of the bond is such that the owner of the public key cannot later deny involvement in the given action. Non-repudiation has the potential to enable legally binding electronic signatures.

The attractiveness of non-repudiation is often used to market new PKIs. BankID has been marketed with such capabilities. The service of non-repudiation is difficult to implement. It necessitates a way to determine the order in which various events occurred. I.e., given three actions A, B, and C, it must be possible to sort out the actions relative order. One approach is to use time to order the elements. A working group under the IETF tried to extend the Network Time Protocol (NTP) to provide authenticated distribution of time [45]. The group's goal was to lay the foundation for a new RFC. The work has now been concluded, but their suggestions has not been turned into a standards-track document as of April 2005. In addition to decide the order in which events happened, it must be possible to prove that the archive was working properly at the time of conflict [30]. This necessitates the definition of a set of standard protocols and procedures that the archive should follow.

Java 5.0 does not contain APIs to design a comprehensive PKI. Basic features such as the ability to generate certificates and CRLs are not addressed by the current version. Third party vendors provide extensions that can fill in the missing pieces. In Section 4.3.2.3, Bouncy Castle's crypto software was presented. Present PKI support in the mobile segment is very limited. Currently, even generation of cryptographic keys is infeasible to do in software on a smartphone. Java Specification Request (JSR) 177 [26] is intended to bring better PKI capabilities to mobile clients.

The widespread use of PKI is not only being stalled by technological problems, political and economical issues must also be resolved. In terms of politics, the notion of trust is at the centre of attention. Who should be responsible for the deployment and day-to-day operation of the infrastructure? If the PKI is to be used for a single service, such as Internet banking, it can be perfectly reasonable to let the bank operate the infrastructure. As soon as you intend to use the PKI for a wider range of services, you cannot allow a provider of one service to operate the entire machinery. A bank should not be given the possibility to access sensitive medical records. Financially, the biggest obstacle is to make the PKI cost effective. Developing and maintaining a PKI is very expensive. The BankID PKI has already cost more than NOK 100 million [60]. It should also be noted that this figure only covers an initial investment, the PKI still hasn't been set in operation. Once in production, the infrastructure is estimated to cost NOK 20-30 million a year [60]. Several solutions have been sketched to give the BankID constellation a return on their investment. The alternatives include billing the end user for each digital signature, or for each CRL download.

4.5 Summary

Current Internet banking solutions employ **weak client authentication**. New systems are on the way, such as **BankID**, but they seem to emphasis service needs instead of solving present security problems. **Strong client authentication** procedures can be implemented in Java with the help of a PKI and third party libraries. A PKI can offer increased security, but technological, political, and economical challenges must be overcome to ensure widespread use.

In Chapter 5, the underlying infrastructure is used as a basis for the **SSL protocol**.

Chapter 5

Transport Security

The widely deployed SSL protocol offers authentication, integrity, and confidentiality services. In order to work, SSL relies on an underlying infrastructure to provide server and client certificates. In the following, the PKI developed in Chapter 4 is the foundation in an implementation of the SSL protocol. Various techniques and information presented throughout the thesis, including NIO, Java security fundamentals, the JSSE API, and basic operations in a PKI, make up the core ingredients in the development of an SSL prototype.

The chapter starts with a look at a current deployment of SSL. Next, a quick glance at the internal workings of the protocol is provided. The last part of the chapter describes the prototype.

More information on SSL can be found in "SSL and TLS essentials—Securing the Web" [56] and RFC 2246 [20]. For those interested in the protocol in a Java context, the "Java Secure Sockets Extension (JSSE) Reference Guide" [28] is priceless.

5.1 Background

The application code in Section 4.3.2.8 shows how a PKI can be used to realize mutual authentication between communicating peers. The purpose of the example was to illustrate how an underlying PKI can be used to offer security services. Once you have the infrastructure in place, you can build your own protocols that best fit the needs of your system. When it comes to the primary security services, people have been experimenting with different approaches for quite a while. At present, developers rely heavily on SSL to implement these services. With an industry-wide approval of a standard that has been tested and updated by software makers since the mid-90s, there's no need to create your own protocol. In fact, designing security protocols is a difficult task. The common scenario for new protocols is to go through several revisions before they are recognized as production quality software. The history of SSL is no different, the protocol is now in version 3.1.

Almost all shopping on the Internet is done using SSL. The seller acquires a certificate from an allegedly trustworthy CA, and uses it to set up a secure communication channel with customers. The merchant does not care who you are, only that you can present a valid credit card.[1] The buyer has two main reasons to feel comfortable trading on-line:

1. She can authenticate the seller (=web server) by examining the certificate used to create the SSL session. Currently, the domain name of the server is bound to a public key by an assumingly trusted third party.
2. SSL sets up a secure communication channel preventing others from viewing or altering the sensitive information you exchange.

While it may be satisfactory for both parties to trade goods using the above-mentioned approach, some agreements cannot be effectively handled by the scheme. Any form of contract signing or access to sensitive information such as medical records necessitates authentication of the client. Also, today's practice in on-line shopping, where possession of credit card information enables anyone to bill the owner of the card is far from ideal. A better solution would be to require a digital signature on each purchase. This way an identity is bound to the transaction, making the settlement of any subsequent disputes easier.

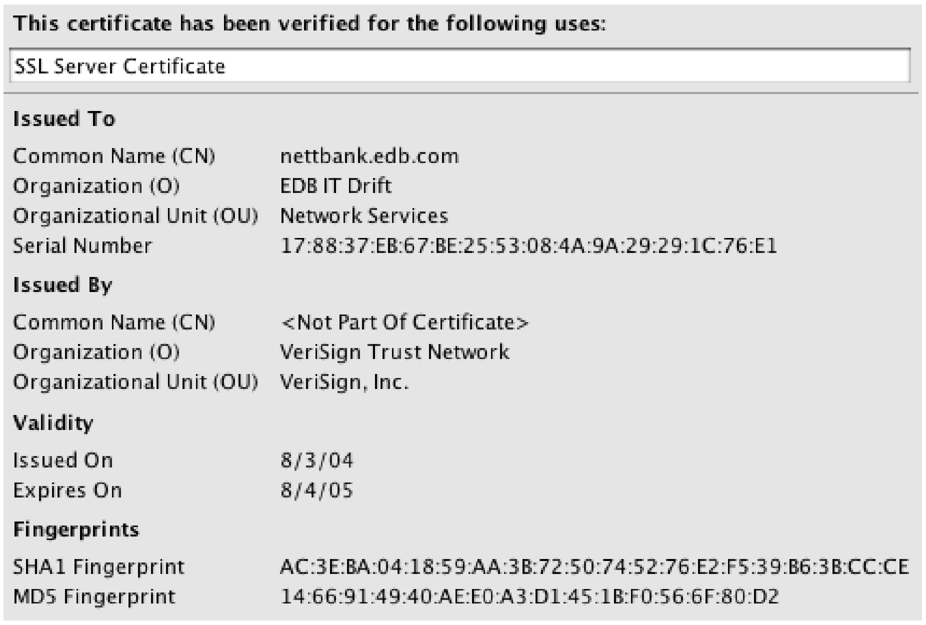
As an example of current usage of SSL among providers of Internet banking, we shall look at Sparebanken Vest's on-line banking system. From their main site [50], you can navigate a link to a login site for Internet banking. This link sends you to the server `nettbank.edb.com`, which communicates with clients over HTTPS. An SSL session is established over HTTP, where server side authentication is enabled. As of April 3rd 2005, the server uses the certificate depicted in Fig. 5.1. Current browser technology verifies the certificate after completing 3 checks:

1. Current date must be within the validity time-interval of the certificate,
2. the DN of the certificate must match the hostname of the server, and
3. the issuing CA must be recognized by the browser so that the certificate's signature can be verified.

The certificate in Fig. 5.1 is within the time-interval, and was issued by Verisign. Since this particular Verisign certificate is trusted by the most popular browsers, the SSL connection is silently established. The client can now communicate securely with `nettbank.edb.com`.

Lets assume that you trust Verisign. The certificate in Fig. 5.1 creates a binding between the domain name `nettbank.edb.com` and a public key. There's no reason for you to trust the identity `nettbank.edb.com`, you should only be convinced that you're presently communicating with `nettbank.edb.com`. Should you give this server your username and password? Put in a similar way: do you trust every server on the Internet that has a certificate issued by Verisign with your login credentials? The obvious answer is no.

[1]Validity is determined based on credit card number, expiration date, and occasionally a security number known as CV2.



This certificate has been verified for the following uses:

SSL Server Certificate

Issued To

Common Name (CN)	nettbank.edb.com
Organization (O)	EDB IT Drift
Organizational Unit (OU)	Network Services
Serial Number	17:88:37:EB:67:BE:25:53:08:4A:9A:29:29:1C:76:E1

Issued By

Common Name (CN)	<Not Part Of Certificate>
Organization (O)	VeriSign Trust Network
Organizational Unit (OU)	VeriSign, Inc.

Validity

Issued On	8/3/04
Expires On	8/4/05

Fingerprints

SHA1 Fingerprint	AC:3E:BA:04:18:59:AA:3B:72:50:74:52:76:E2:F5:39:B6:3B:CC:CE
MD5 Fingerprint	14:66:91:49:40:AE:E0:A3:D1:45:1B:F0:56:6F:80:D2

Figure 5.1: EDB Business Partner ASA's server certificate

Sparebanken Vest believes you should trust this certificate because it belongs to one of their business associates: EDB Business Partner ASA.

An adversary can exploit the bank's SSL practice by intercepting redirects to `nettbank.edb.com`. Instead users can be referred to the attacker's newly registered `nettbank.ed6.com` site, holding a certificate from Verisign. A set of mock-up web pages can be created to simulate the real customer login. After the potential victim has given the adversary his username, password, and one-time PIN, the user can be sent to an error page advising him to try another login later. Meanwhile the attacker can log into the user's account through the real login site.

5.2 The SSL Protocol

A full description of SSL is outside the scope of this thesis, but a few aspects of the protocol must be understood to follow the code to be described in Section 5.3. Especially important is the initial procedure of setting up a secure communication channel between the client and server, better known as the *Handshake Protocol.* Depending on the mode of operation, a predefined set of messages are exchanged in this phase. We want both server and client authentication, which require the messages shown in Fig. 5.2 to be transmitted.

In short, the Handshake Protocol enables the parties to:

- Verify each others certificates,
- decide which algorithms to use for encryption and MAC, and

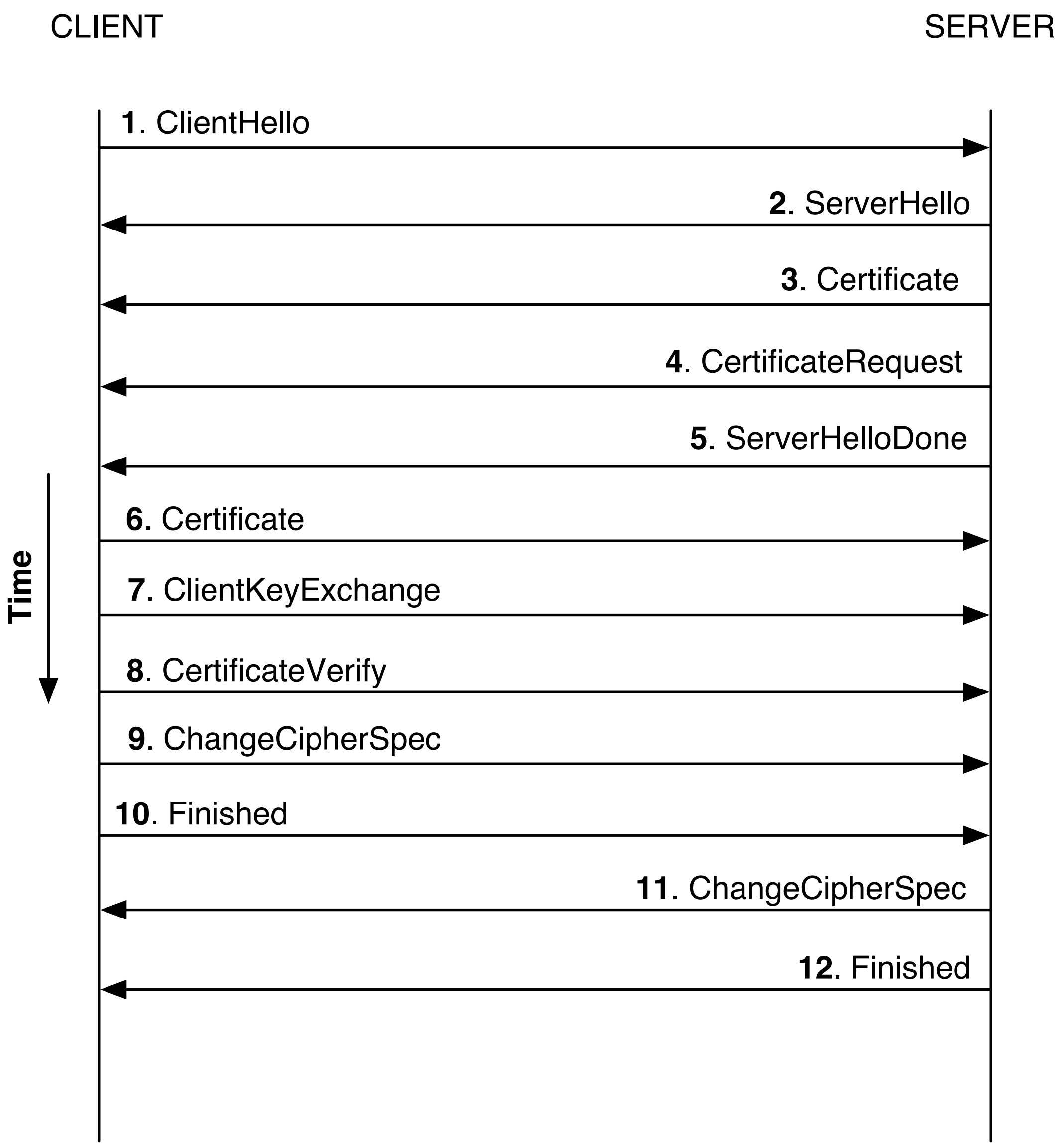


Figure 5.2: SSL Handshake Protocol

- generate secret cryptographic keys.

Upon successful completion of the Handshake Protocol, the client and server can securely exchange application data.

5.3 JSSE

SSL functionality in Java is provided through the JSSE API. It was first introduced in version 1.4 of the Java platform, but has been drastically changed in the 5.0 release. The problem with the initial JSSE implementation was its dependency on stream-oriented I/O, i.e., SSL in Java was too closely coupled with the transport mechanism. The problem caught everyone's attention with the introduction of the NIO API, which uses a different approach to I/O. Prior to the latest major release of Java it was almost impossible to get JSSE to work with NIO. In Java 5.0 the problem was fixed by making JSSE independent of the underlying transport. The price developers have to pay for this new flexibility is increased complexity in the JSSE API.

5.3.1 SSL in Java 1.4

Originally, the idea was to make SSL sockets as easy to use as standard sockets. If you had previously developed a communication model using the ordinary socket API, you should then be able to make it secure by replacing the sockets. In practice, you also had to specify the keystore from which to load your own key material and a *truststore* holding certificates of the CAs you recognize. These are both loaded from keystores. The information in the truststore is later used to verify certificates of others you decide to communicate with.

Assuming that the certificate of a CA you trust is located in the file `/Keystores/CA.ks` and the password to the keystore is `wHEh8eXp`, the following code initializes the trust material:

```
char[] caStorePass = "wHEh8eXp".toCharArray();
KeyStore caKs = KeyStore.getInstance("JKS");
caKs.load(new FileInputStream("/Keystores/CA.ks"), caStorePass);

TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(caKs);
```

Management and trust decisions are handled by `TrustManagers`. These can be created by subclassing the `TrustManager` class or through a `TrustManagerFactory`. The example above illustrates the last option. The trust material is coupled with the manager through the `init(Keystore ks)` method of the factory. Initializing your own key material is done in the same manner using the `KeyManagerFactory` class. The only difference is that you need to supply the password protecting the private key in the keystore when you initialize the manager.

The connection between the trust and key material, and the SSL protocol is established in an `SSLContext`. Two lines of code create that bond:

```
SSLContext sslCon = SSLContext.getInstance("TLS");
sslCon.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
```

You must specify which version of the SSL protocol to use in the `getInstance(String algorithm)` method of the `SSLContext` class. Next, the context is initialized with key and trust managers, and possibly a source of randomness. The latter is used when setting up new SSL sessions with this `SSLContext`. In the code above, `null` is passed instead of an instance of the `SecureRandom` class, causing the default implementation to be used. Sun's cryptographically strong pseudo random number generator [24] complies to FIPS 140-2 [10] and RFC 1750 [19].

After successfully initializing the `SSLContext`, you can get an SSLSocketFactory to create SSL sockets by invoking the context's `getSocketFactory()` and `getServerSocket \ Factory()` methods. The rest of the programming involved in setting up the SSL communication channel mimic that of setting up a scheme not using SSL, and is left to the reader.

In summary, developers didn't need to know anything at all about the actual workings of the SSL protocol to get it to work in Java 1.4. All the details were left up to providers to implement. One of the providers distributed with releases of the Java platform implements SSL: `com.sun.net.ssl.internal.ssl.Provider`. For more information on providers, please see Section 3.3.1.

5.3.2 The New SSL API

With the release of Java SDK 5.0, Sun introduced a new approach to SSL. Key and trust material must still be initialized with an `SSLContext`, as described in the previous section. The most important new addition is the `SSLEngine` class which encapsulates a state-machine for SSL sessions. Depending on the current state, an instance of this class produces and handles SSL specific data. It is up to the developer to transport data to and from the engine, making it transport-independent. Santos gives a brief introduction to the new API in "Using SSL with Non-Blocking IO" [43]. A more thorough treatment can be found in the JSSE reference guide [28]. If you're planning to write your own scalable application using SSL, sample code for a NIO-based HTTPS server is distributed with the Java 5.0 JDK. The code can be found in `JDK_HOME/sample/nio/server`, where `JDK_HOME` points to the installation directory of your JDK.

5.3.2.1 The Handshake Protocol

Assuming that both communicating parties have initialized an `SSLContext`, the next step is to perform the Handshake Protocol. All the data necessary to execute the protocol is generated by an `SSLEngine`. This class is retrieved from an `SSLContext`:

```
SSLEngine sslEngine = sslc.createSSLEngine();
```

In the `setUseClientMode(Boolean)` method you specify if you're setting up the client or server side of the Handshake Protocol. Client side authentication also needs to be explicitly set:

```
sslEngine.setNeedClientAuth(true);
```

The workings of the `SSLEngine` is presented in Fig. 5.3. The data produced and consumed by the engine is handled by 4 `ByteBuffers`. Data coming from the network is read into the `inNet` buffer. The `outNet` buffer holds information that should be transported over the communication channel. Application data coming from the network or ready to be dispatched over the network is held by the `inApp` and `outApp` buffers, respectively. All 4 buffers are managed by the `SSLEngine` through the `wrap` and `unwrap` methods. An invocation of `wrap(outApp, outNet)` causes the state machine to put handshake data, corresponding to the current state of the protocol, and application data contained in the `outApp` buffer into the `outNet` buffer. If we're handshaking, there's usually no need to send any application data along with the handshake data. In the same manner, invoking `unwrap(inNet, inApp)` will feed any handshake data to the `SSLEngine` and application data is filled in the `inApp` buffer.

Each call to `wrap` and `unwrap` returns an `SSLEngineResult`. This class encapsulates two indicators on `SSLEngine` status:

1. Overall `SSLEngine` status, represented by the nested class `SSLEngineResult.Status`.
2. Handshake status, represented by the nested class `SSLEngineResult.HandshakeStatus`.

The overall status can be any of the following:

- BUFFER_OVERFLOW, indicates that the `SSLEngine` was unable to perform an operation, because the destination buffer is too small to hold the data produced.
- BUFFER_UNDERFLOW, happens when the engine receives a partial SSL message. Necessitates reading of more data from the network.
- CLOSED, the SSLEngine is closed and can no longer be used.
- OK, the operation was completed successfully. The engine is now ready to process more calls.

The handshake status can be

- FINISHED, indicates that the engine has **just** completed handshaking.
- NEED_TASK, one or more tasks have to be run before handshaking can proceed. These tasks are retrieved by an invocation of the `getDelegatedTasks()` method of the `SSLEngine`.

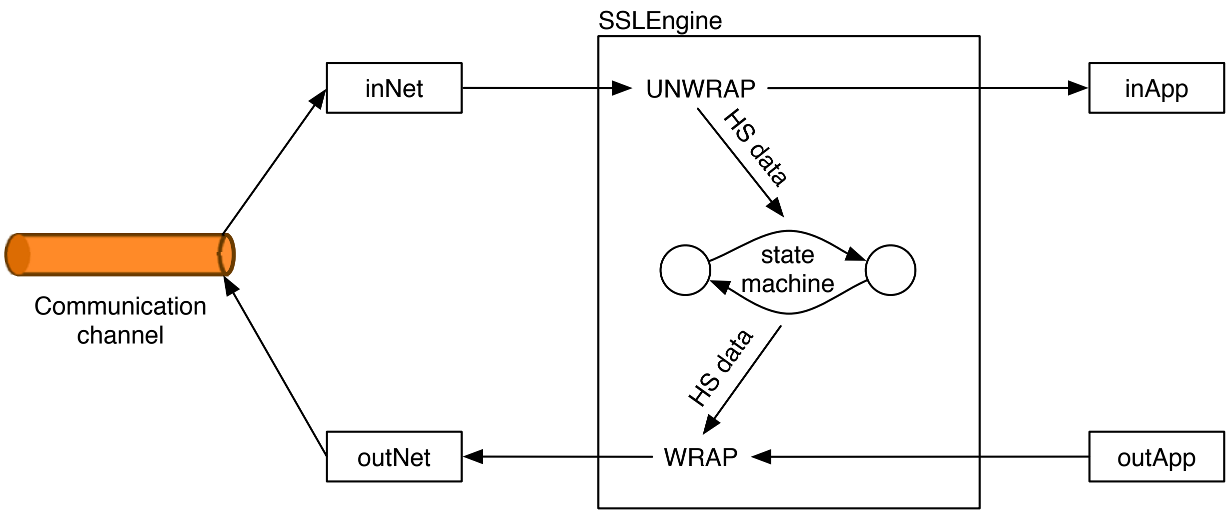


Figure 5.3: JSSE communication model

- NEED_WRAP, the engine needs to send handshake data over the network.
- NEED_UNWRAP, the engine is expecting handshake data from the network.
- NOT_HANDSHAKING, indicates that this instance of SSLEngine is not currently handshaking. It does not state whether the handshake has been completed or not.

Looking again at Fig. 5.3, lets assume that you are implementing the server side of the Handshake Protocol. According to Fig. 5.2, the first message from the client is `'ClientHello,'` which is read into the `inNet` buffer. At this time the handshake status of the server side should be either `NOT_HANDSHAKING` or `NEED_UNWRAP`, which can be resolved by invoking the `getHandshakeStatus()` method of the engine. In the first case, you need to start the handshake by calling the `beginHandshake()` method, which updates the handshake status to `NEED_UNWRAP`. Next, you need to unwrap:

```
SSLEngineResult result = sslEngine.unwrap(inNet, inApp);
```

Ideally, the call feeds the hello message from the client to the engine, and any application data is sent to the `inApp` buffer. Both the overall and handshake status of the engine is updated, and can be queried with these calls:

```
SSLEngineResult.Status overallStatus = result.getStatus();
SSLEngineResult.HandshakeStatus hsStatus= result.getHandshakeStatus();
```

If everything executed nicely, the overall status should now be `OK` and the handshake status should be `NEED_WRAP`. These results indicate that the `SSLEngine` has received the `'ClientHello'` message and needs to send data to the client. This is achieved through a wrap:

```
SSLEngineResult result = sslEngine.wrap(outApp, outNet);
```

Any data contained in the `outApp` buffer is copied to the `outNet` buffer, along with handshake data from the `SSLEngine`. From Fig. 5.2 we can see that the handshake data consists of the 4 messages: `'ServerHello,'` `'Certificate,'` `'Certificate Request,'` and `'ServerHelloDone.'` After invoking `wrap`, the developer must query the `result` to see what to do next. In most cases the `SSLEngine` will now be waiting for data from the client, meaning that the handshake status is `NEED_UNWRAP`.

The client side of the Handshake Protocol is implemented in the same way: testing on the overall and handshake status of the client's `SSLEngine` and creating appropriate code to act on the various states.

5.3.2.2 Exchanging application data

After the handshake has been completed, application data can be exchanged in a secure fashion. Secure meaning that the underlying SSL connection provides the primary security services. Lets assume that a client and server have established an SSL session, and have finished the Handshake Protocol. They are now ready to send and receive application data. The following steps are required to send a message:

1. The data must be transferred to a `ByteBuffer`. E.g., the information you want to send is contained in the `String` variable `sensitiveIntel`. The data can be put into a buffer by invoking: `ByteBuffer.wrap(sensitiveIntel.getBytes() )`
2. Feed the data to the previously initialized `SSLEngine`. E.g., assuming the result of the call in step 1 was assigned to the `ByteBuffer` named `outAppBB`, you invoke `SSLEngineResult result = sslEngine.wrap(outAppBB, outNetBB)`.
3. The overall status of the engine must be consulted to see if further handling is required. E.g., in case of a `BUFFER_OVERFLOW`, the `outNetBB` buffer is too small to hold the encrypted data, and must be resized.
4. The data contained in `outNetBB` must be sent over the network. E.g., by invoking `write(outNetBB)` on a `SocketChannel` until the buffer is emptied.

Reading data from a communication peer involves reading from the network, and using the `SSLEngine` to unwrap data to the `inAppBB` buffer:

```
SSLEngineResult result = sslEngine.unwrap(inNetBB, inAppBB)
```

If the overall status of the operation is `OK`, then data can be retrieved from `inAppBB`.

5.3.2.3 Pitfalls

Following the guidelines above should get you started in developing your own applications using the new SSL API. However, there are a few pitfalls that are likely to cause headache and frustration. The biggest obstacle is to get the `unwrap` method of the SSLEngine to work properly. As illustrated in Fig. 5.3, any application data is placed in the `inAppBB ByteBuffer`. The catch is that this buffer must be empty before performing the unwrap. If it contains any data when an unwrap is attempted, the overall status of the engine will be set to `BUFFER_OVERFLOW`, indicating that there is not enough space in the `inAppBB` buffer. Normally, you would proceed by making the buffer larger. This will not solve the problem. As long as `inAppBB` hasn't been cleared, a call to `unwrap(inNetBB, inAppBB)` will update the `result` variable to indicate `BUFFER_OVERFLOW`. The bottom line is that application data must be drained from the `inAppBB` buffer after each unwrap that returns status `OK`. After retrieving the information, the buffer must be cleared by invoking `inAppBB.clear()`. As it turns out, the problem was identified on Sun's java-security forum [25] prior to the JDK 5.0 release. It hasn't been fixed yet, meaning that developers either must consume application data on the fly or create an additional buffer to hold the intermediary data.

It is also important to realize that data from the network are often read in chunks, meaning that many SSL packets can be read into the `inNetBB` buffer in one invocation of `read(inNetBB)`. Each call to `unwrap(inNetBB, inAppBB)` will only handle a single SSL packet. Therefore, one read can necessitate many unwraps.

5.3.3 The Prototype

In March 2005, the author and Yngve Espelid[2] started a project whose goal was to combine SSL and NIO in order to make a security framework in Java. We have now completed a prototype that successfully runs the SSL Handshake Protocol. Upon completion of the protocol, users can securely exchange application data. It should be stressed that the project is a work in progress. It can be followed from Espelid's web site [44]. Currently, the prototype successfully combines NIO, PKI and SSL to secure client-server communications. The intersection of these technologies has the potential to strengthen for instance Internet banking. Please note that the prototype encapsulates limited functionality and that it is merely a start towards a fully functional framework. No effort has been made to deal with abrupt termination of the SSL protocol, the code has not been tested, no benchmarking against stream-oriented Java applications has been done, and no CRL checking is performed during validation of certificates. These are all issues that should be dealt with later.

5.4 SSL and PKI

SSL is the single-most important protocol when it comes to securing e-commerce. Despite its widespread use, SSL is not a complete security solution in itself. In fact, the protocol

[2]Yngve Espelid is a Ph.D. student in The NoWires Research Group.

needs an underlying PKI to provide authentication, integrity, and confidentiality services. If you're developing a PKI where other services are needed, other protocols must also be implemented. The BankID team has decided to not use the SSL protocol at all. Their PKI is implemented with proprietary protocols [60]. The result is that a BankID client can only communicate with a BankID server. The BankID PKI is therefore a closed infrastructure, limiting its application domain to banking services.

The relationship between a PKI and the SSL protocol is important to understand. A security solution should not be trusted merely because it uses SSL. Trust must come from the underlying PKI. Today's software makers focus on crafting good-looking and easy-to-use applications in order to capture market shares. Security issues are sacrificed to ensure usability. In Sparebanken Vest's Internet Banking system, important trust decisions are automated in software. As described in Section 5.1, the browser automatically accepts the server `nettbank.edb.com`'s certificate. The key point is that while SSL provides its services for data in transit, the protocol cannot be used directly to deduce trust.

5.4.1 Side-effects of client side authentication

Requiring clients to authenticate to the server forces each client to manage keys and certificates. In particular, a client's private key should not be disclosed to anyone. This means that users must be educated in protecting sensitive information. In today's Internet banking schemes, users are already accustomed to keeping information secret. Sparebanken Vest have equipped their customers with passwords and cards with one-time PINs. Current research, such as the paper mentioned in Section 4.2 [12], recommends that the private key should be stored on a smart card. Through using a card reader, the private key can be loaded into the memory of a computer. If the bank develops software to handle the process of using the private key in establishing the SSL connection, users are not bothered with any cryptographic details. So from a usability standpoint, a PKI solution doesn't seem to impose a greater burden on the end user.

Client side authentication can become a substantial threat to client privacy. Using the techniques applied throughout the thesis to develop a large-scale PKI, will make the infrastructure a powerful surveillance tool in some settings. Imagine a PKI operated by a government. Assume that citizens are encouraged to use the infrastructure to perform tasks they traditionally have accomplished through meetings in person with different governmental departments. Examples include filing tax returns, ordering passports, change home address, and getting a new drivers license. By collecting this information, the government can follow the inhabitants closely. Not only can it do so, the process can also be automated in software. As more and more services are added, the government will get more information and control over its citizens. To counter this effect, the PKI can offer *anonymous* certificates, where there is no binding between the certificate and an identity. Canada's E-government initiative uses anonymous certificates indexed by a Meaningless But Unique Number (MBUN) [30]. No association between the certificate and the owner's identity is made before enrolling with a particular department or service. During enrollment, the MBUN is mapped to a Social Insurance Number.[3] Individuals are free to re-use the certificates or throw them away after each communication session. One-time use provides citizens with the highest degree of privacy, but results in more overhead as you would have to register and enroll every time.

5.4.2 SSL on a compromised host

Some people argue that a user's computing environment must be assumed to be compromised, meaning that the private key cannot be read into the memory of the host. In "Pocket device for authentication and data integrity on Internet banking applications" [41], the authors argue the need for an offline trusted computing device. The main idea is that the private key should never be read into the memory of a device connected to the Internet. An offline device with a keyboard and screen is suggested to replace the computer when it comes to cryptographic operations involving the private key. When logging into the bank, the user must input a challenge on the device, and then feed the computed result back to the on-line computer. This approach will effectively thwart viral attacks, but at the expense of degraded usability.

Using the technique above in conjunction with the SSL and NIO project, impacts the `CertificateVerify` message in Fig. 5.2, which is a digital signature on a hash constructed using the key information and all previously exchanged SSL messages. If an RSA certificate is used, the length of the signed hash information is 36 bytes [56], which translates to way too many characters for a user to input on the offline device and the computer. Such a scheme is clearly not viable in an Internet banking setting. In the before-mentioned SSL and NIO project [44], we are currently investigating other possible solutions.

5.5 Summary

Through the use of an underlying PKI, the **SSL protocol** can be used to provide the **primary security services** for data in transit over a network. Current deployment of **SSL in e-commerce** is mostly set up to authenticate servers to customers, but the protocol can be configured to **authenticate clients** as well. Sparebanken Vest's Internet banking solution uses SSL to authenticate an **outsourced server** to clients, which reveals a **highly questionable trust model**. The **JSSE API** was used in Section 5.3 to show readers how to implement **SSL in Java**. The code enables mutual authentication, integrity, and confidentiality of data.

[3]A Social Insurance Number is Canada's equivalent to a US Social Security Number.

Chapter 6

Summary and Conclusions

6.1 Summary

A thick-client version of the client-server paradigm can be used to create a communication framework for Internet banking. With the introduction of the NIO API, it is now possible to implement highly scalable applications in Java. As a programming language, Java is a good choice when it comes to the development of secure systems. The Java security model encapsulates functionality to take proactive measures against malicious hackers. A number of APIs bring advanced security mechanisms to the programming platform. In particular, J2SE contains PKI capabilities.

Safeguarding information on the Web is a particularly challenging task due to the availability requirements the software must meet. In the banking industry the production of good-quality code is further hindered by the development paradigm, which prohibits software makers to publish algorithms and protocols. The financial industry is clearly practicing security through obscurity.

Current Internet banking systems employ weak security mechanisms. Clients are authenticated based on credentials that can be easily generated by a powerful adversary. Servers are authenticated to users through certificates, as an intermediary step in the SSL protocol. In Sparebanken Vest's system, the identity credentials in the certificate do not appear to be the bank itself. Presently, an initiative by the name BankID aims to bring a complete PKI solution to the Norwegian Internet banking industry. The project's outspoken goal is not to improve security, but to offer customers a wider range of services.

Through the use of core Java classes and the third party provider known as Bouncy Castle, a PKI can be set up and managed in Java. In addition to showing readers how to set up such an infrastructure, an example on how to enable mutual authentication between PKI members was given in Section 4.3.2.8.

SSL is a well-suited protocol to run on top of a PKI to provide the primary security services. Current usage of SSL in on-line banking is for servers to authenticate themselves to clients, but SSL can also be configured to achieve mutual authentication. By combining the NIO and JSSE libraries, a scalable and secure server can be implemented in Java.

6.2 Academic Progression and Uniqueness

This section outlines the author's academic development throughout the work with the thesis. The discussion includes both theoretical and technological progress. Also, a brief comment on the uniqueness of the work is given.

6.2.1 Progression

Considerable time went into understanding the inner workings of the Java security model. While it is easy to give an elegant overview of the security principles in Java, which usually entails a presentation of the sandbox analogy, a thorough treatment of the matter is challenging. Sun's choice to make every new version of Java compatible with earlier releases has introduced redundancy and awkward workarounds into the security model. An example is the existence of both an access controller and a security manager. The best way to learn more about Java security is through programming.

A number of APIs have been investigated, ranging from client-server communication in NIO to SSL specific code in JSSE. In addition, PKI capabilities in the `java.security.cert` package and a handful of classes from Bouncy Castle were studied.

A great deal of effort has gone into researching PKI. The theoretical background given in the thesis was intentionally cut to the bone in order to concentrate on the primary security services. The translation from theory to practice is not trivial. The technology, Java in this case, dictates the design of the PKI. This means that developers must know about the programming language's shortcomings prior to the initial design.

The last part of the project focused on SSL. Before J2SE 5.0, setting up an SSL communication scheme was a walk in the park. The expense coders had to pay in earlier releases was lost flexibility. An example is that NIO and SSL could not be combined. Version 5.0 gave programmers more flexibility, but introduced a significant amount of complexity. The JSSE API is now an advanced API, requiring a good understanding of both Java programming and SSL. The complexity is further increased by the PKI component.

6.2.2 Contributions

The uniqueness of the project is the combination of client-server communication (NIO), a security infrastructure (PKI), and a protocol using the security services provided by the infrastructure (SSL). This thesis also introduces strong client side authentication in SSL as a way to strengthen Internet banking. In terms of that technique, it is important to understand the close relationship between the PKI and SSL. A well-functioning foundation encompassing procedures to administer certificates is an absolute necessity for the scheme to work. On top of an explanation of the theory behind the technology is a step-by-step implementation of the concepts in Java.

6.3 Conclusions

Today's Norwegian on-line banking systems fail to provide basic security services. The existence of very simple attacks, such as the one described in Selmersenteret's report [14] and the design-flaw identified in Section 5.1, indicate that security was not a high priority for the parties responsible for developing these applications. The production of secure and robust software is a challenging problem. The Internet banking industry could benefit from disclosing details on protocols and algorithms used in their programs to independent security researchers. Emphasis must also be shifted from service-minded development to a security-minded approach.

PKI is a security infrastructure that appears suitable to strengthen current Internet banking systems. It should be primarily viewed as a foundation that can provide various security mechanisms to a system.

The application code developed in this thesis demonstrate how PKI can be used to improve security in current Internet banking systems. The PKI architecture given in the text is not locked to a specific scenario, and can easily be used for other purposes than on-line banking. Care should be taken when deciding whether or not to use Java to implement a PKI in a business setting. The limited support for PKI in core Java libraries force programmers to rely on Bouncy Castle's APIs or to implement the missing pieces themselves. Both alternatives constitute a great deal of work.

6.4 Further Work

On top of the to-do list is a continuation of the NIO and SSL project. As of May 2005, no extensive evaluation of the two technologies combined has been published. By refining the prototype we can possibly create a competitive and more secure alternative to the PIN-SSL solutions used in current Internet banking applications. It also remains to see how well the system scales. Benchmarking the application on a multi-processor architecture will answer that question. The test results can be compared to existing open-source and commercial servers.

In terms of PKI, the services of authorization and non-repudiation are interesting. Neither of them were treated in any depth in the thesis. Authorization is a necessity in an on-line banking setting, and non-repudiation has potential to revolutionize the way we sign information. Besides adding new services to the infrastructure, an expansion of end-user functionality would be interesting. In a large-scale PKI with numerous providers of functionality, privacy issues would become very important.

With the expected growth in the smartphone market, an approach to replace desktop clients with high end cell phones could be very rewarding. The limited computation capabilities in current products, makes an offloading of responsibilities to the server side very likely. PKI in combination with resource-constrained devices is a new research field, both in terms of theory and technology. In particular, the ongoing project of The NoWires Research Group [55] looks very interesting.

A closer look at the BankID project would be of great interest. Due to the scarce information available on the system, an extensive analysis cannot be performed at the time being. Such an effort will have to wait until a full documentation of the system is released, or more realistically: when the project is set into production.

Bibliography

[1] Carlisle Adams and Steve Lloyd, *Understanding PKI—Concepts, Standards, and Deployment Considerations*. Pearson Education, Inc, second edition 2003.

[2] Ross Anderson, "Why Cryptosystems Fail," ACM 1st Conf.- Computer and Comm. Security 1993.

[3] Ross Anderson, Alan Blackwell, Alashdair Grant, and Jeff Yan, "Password Memorability and Security—Empirical Results," *IEEE Security and Privacy*, September/October 2004.

[4] BankID.no. Retrieved January 21, 2005, from bankid.no. `http://www.bankid.no/`

[5] Building Secure Software—Best practices in software security. Retrieved May 16, 2005, from Cigital.
`http://www.cigital.com/presentations/roots/bss05/index_files/frame.html`

[6] Tin-Wo Cheung and Samuel T. Chanson, "Design and Implementation of a PKI-based End-to-End Secure Infrastructure for Mobile E-Commerce", *World Wide Web Journal*, Volume 4, No. 4, pp. 235-254, 2001.

[7] Code. Retrieved April 15, 2005, from ii.uib.no.
`http://www.ii.uib.no/~larshn/static/code.html`

[8] Dagbladet.no. *Den digitale bankhverdagen*. Retrieved March 6, 2005, from db.no.
`http://www.dagbladet.no/dinside/2005/02/27/424662.html`

[9] Encyclopædia Britannica. Cryptology. Retrieved May 3, 2004, from Encyclopædia Britannica Online. `http://www.search.eb.com/eb/article?eu=117762`

[10] FIPS PUB 140-2 Security Requirements for Cryptographic Modules. Retrieved November 1, 2004, from Computer Security Division, NIST.
`http://csrc.nist.gov/cryptval/140-2.htm`

[11] Stuart McClure, Joel Scambray, and George Kurtz, *Hacking Exposed—Network Security Secrets & Solutions*. McGraw-Hill, fourth edition 2003.

[12] Alain Hiltgen, Thorsten Kramp, and Thomas Weigold, "Secure Internet Banking Authentication," submitted to *IEEE Security & Privacy*, Nov., 2004.

[13] Ron Hitchens, *Java NIO.* O'Reilly, first edition 2002.

[14] Kjell J. Hole, Vebjørn Moen, Thomas Tjøstheim, "Security in Norwegian Internet Banks," submitted to *IEEE Security & Privacy*, April, 2005.

[15] Russ Housley and Tim Polk, *Planning for PKI—Best Practices Guide for Deploying Public Key Infrastructure.* John Wiley & Sons, Inc., 2001.

[16] How to Implement a Provider for the Java Cryptography Architecture. Retrieved November 25, 2004, from sun.com.
`http://java.sun.com/j2se/1.5.0/docs/guide/security/HowToImplAProvider.html`

[17] Internet RFC/FYI/STD/BCP Archives. RFC 768. Retrieved August 27, 2004, from Internet FAQ Archives - online education.
`http://www.faqs.org/rfcs/rfc768.html`

[18] Internet RFC/FYI/STD/BCP Archives. RFC 793. Retrieved August 27, 2004, from Internet FAQ Archives - online education.
`http://www.faqs.org/rfcs/rfc793.html`

[19] Internet RFC/FYI/STD/BCP Archives. RFC 1750. Retrieved April 28, 2005, from Internet FAQ Archives - online education.
`http://www.faqs.org/rfcs/rfc1750.html`

[20] Internet RFC/FYI/STD/BCP Archives. RFC 2246. Retrieved March 24, 2005, from Internet FAQ Archives - online education.
`http://www.faqs.org/rfcs/rfc2246.html`

[21] Internet RFC/FYI/STD/BCP Archives. RFC 3280. Retrieved November 9, 2004, from Internet FAQ Archives - online education.
`http://www.faqs.org/rfcs/rfc3280.html`

[22] ISP Telenor cripples zombie PC network. Retrieved May 16, 2005, from Computerworld. `http://www.computerworld.com/printthis/2004/0,4814,95847,00.html`

[23] Java 2 Microedition security. Retrieved August 30, 2004, from Java 2 microedition security. `http://www.kenti.org/nowires/`

[24] Java 2 Platform, Standard Edition, v 1.5 API Specification. Retrieved October 18, 2004, from sun.com.
`http://java.sun.com/j2se/1.5/docs/api/index.html`

[25] Java-Security@sun.com archives. Retrieved April 4, 2005, from archives.java.sun.com. `http://archives.java.sun.com/cgi-bin/wa?A2=ind0403&L=java-security&F=&S=&P=4272`

[26] Java Community Process—JCP Home. Retrieved April 12, 2005, from www.jcp.org. `http://www.jcp.org/en/jsr/detail?id=177`

[27] Java Cryptography Extension. Retrieved November 29, 2004, from sun.com. `http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html`

[28] Java Secure Sockets Extension (JSSE)—Reference Guide. Retrieved November 28, 2004, from sun.com. `http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html`

[29] Java Cryptography Architecture—API Specification & Reference. Retrieved November 23, 2004, from sun.com. `http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html`

[30] Mike Just and Danielle Rosmarin, "Meeting the Challenges of Canada's Secure Delivery of E-Government Services," in Pre-Proceedings to 4th Annual PKI R&D Workshop: Multiple Paths to Trust, Gaithersburg, MD, April 2005.

[31] Kazakhstan Hacker Sentenced to Four Years Prison for Breaking into Bloomberg Systems and Attempting Extortion. Retrieved May 16, 2005, from U.S. Department of Justice. `http://www.usdoj.gov/criminal/cybercrime/zezevSent.htm`

[32] Keytool - Key and Certificate Management Tool. Retrieved February 3, 2005, from sun.com. `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html`

[33] Pankaj Kumar, *J2EE Security—For Servlets EJBs, and Web Services*. Prentice Hall PTR, first edition 2004.

[34] James F. Kurose and Keith Ross, *Computer Networking—A Top-Down Approach Featuring the Internet*. Pearson Education, Inc., second edition 2003.

[35] Wenbo Mao, *Modern Cryptography—Theory & Practice*. Prentice Hall PTR, first edition 2004.

[36] Merriam-Webster OnLine. Client. Retrieved August 6, 2004, from Merriam Webster OnLine. `http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=client&x=15&y=15`

[37] Merriam-Webster OnLine. Server. Retrieved August 6, 2004, from Merriam Webster OnLine. `http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=server&x=15&y=15`

[38] *Norsk BankID sertifikat policy for sertifikater til personkunder, v 1.0.* Bankenes Standardiseringskontor, July 2004.

[39] Nowires.org. *Hvor sikre er norske nettbanker?* Retrieved January 19, 2005, from Nowires.org. `http://www.nowires.org/nettbanker/index.html`

[40] Scott Oaks, *Java Security.* O'Reilly, second edition 2001.

[41] F. de la Puente, J.D. Sandoval, and P. Hernandez, "Pocket device for authentication and data integrity on Internet banking applications," in proceedings for IEEE 37th annual 2003 International Carnahan Conference, 14-16 Oct. 2003, pps: 43 - 50.

[42] Nuno Santos. Building Highly Scalable Servers with Java NIO. Retrieved October 5, 2004, from onjava.com.
`http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html`

[43] Nuno Santos. Using SSL with Non-Blocking IO. Retrieved March 10, 2005, from onjava.com. `http://www.onjava.com/pub/a/onjava/2004/11/03/ssl-nio.html`

[44] Secure and scalable client-server communication using SSL and NIO. Retrieved April 15, 2005, from ii.uib.no. `http://www.ii.uib.no/~yngvee/?sslnio`

[45] Secure Network Time Protocol (stime). Retrieved April 14, 2005, from the Internet Engineering Task Force. `http://ietf.org/html.charters/OLD/stime-charter.html`

[46] Share of individuals having ordered/bought goods or services for private use over the Internet in the last three months. Retrieved May 16, 2005, from Eurostat. `http://epp.eurostat.cec.eu.int/portal/page?_pageid=1996,39140985&_dad=portal&_schema=PORTAL&screen=detailref&language=en&product=EU_yearlies&root=EU_yearlies/yearlies/I/I5/ecb15632`

[47] Share of individuals with Internet access having encountered security problems. Retrieved May 16, 2005, from Eurostat. `http://epp.eurostat.cec.eu.int/portal/page?_pageid=1996,39140985&_dad=portal&_schema=PORTAL&screen=detailref&language=en&product=Yearlies_new_industry&root=Yearlies_new_industry/D/D7/ecb16656`

[48] Simon Singh, *The Code Book—The Secret History of Codes and Code-Breaking.* Doubleday, first edition 1999.

[49] Ian Sommerville, *Software Engineering.* Addison-Wesley, sixth edition 2001.

[50] Sparebanken Vest. Retrieved April 3, 2005, from spv.no. `http://spv.no`

[51] William Stallings, *Cryptography and Network Security—Principles and Practices.* Prentice Hall, third edition 2003.

[52] Douglas R. Stinson, *Cryptography—Theory and Practice.* Chapman & Hall/CRC, second edition 2002.

[53] Thawte. Retrieved November 19, 2004, from thawte.com. `www.thawte.com`

[54] The Legion of the Bouncy Castle. Retrieved January 23, 2005, from bouncycastle.org. `http://www.bouncycastle.org/index.html`

[55] The SWAP Project. Retrieved May 26, 2005, from nowires.org. `http://www.nowires.org/SWAP/Intro.html`

[56] Stephen Thomas. *SSL and TLS Essentials—Securing the Web.* John Wiley & Sons, Inc., first edition 2000.

[57] User Authentication and Authorization in the Java Platform. Retrieved November 28, 2004, from sun.com. `http://java.sun.com/security/jaas/doc/acsac.html`

[58] Verisign. Retrieved November 19, 2004, from verisign.com. `www.verisign.com`

[59] World Internet Usage Statistics and Population Stats. Retrieved May 16, 2005, from internetworldstats.com. `http://www.internetworldstats.com/stats.htm`

[60] Jon Ølnes, "Status for PKI i Norge ved utgangen av 2004," IBM, January 2005. `www.norid.no/pki/pki-norge2004.pdf`

[61] Andre Årnes, Mike Just, Svein J. Knappskog, Steve Lloyd, Henk Meijer, "Selecting Revocation Solutions for PKI," in Proceedings NORDSEC 2000 Fifth Nordic Workshop on Secure IT Systems, Reykjavik, 2000.