MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Advanced authentication in Java applications using Kerberos protocol

MASTER'S THESIS

Bc. Tomáš Král

Brno, spring 2011

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked on myself. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Mgr. Pavel Tuček

Acknowledgement

I would like to thank my supervisor Mgr. Pavel Tuček for his guidance and support during writing of this thesis, especially for commenting and suggesting improvements of the text. I would also like to thank him for encouraging me to write the text in English and for solving related formal issues.

Also I would like to thank RNDr. Petr Švenda, Ph.D. for his support and answering of my Java Cards related questions.

I also would like to thank the company Y Soft for publishing this topic of master's thesis, which gave me opportunity to learn something more about authentication protocols and Java Card technology.

Last but not least, I would like to thank my family, especially my mother, for support during all the years of my studies.

Abstract

This thesis deals with authentication and authorization in Java applications using Kerberos protocol. Especially with the possibility of saving a service ticket on a flash drive or on a smart card and later using it on another machine to make an authorized service request to a third party application.

Demonstration applications showing a working solution for both — saving of a ticket and using of a previously saved ticket through a negotiation mechanism *SPNEGO,* were created and are also described on the end of the thesis.

Keywords

Java, authentication, authorization, JAAS, GSS–API, Kerberos, SPNEGO

Contents

1 **Introduction** . . . 1
2 **Analysis** . . . 3
2.1 Requirements analysis . . . 3
2.2 Options of authentication and authorization in Java . . . 4
2.2.1 Java Authentication and Authorization Service . . . 4
2.2.2 Generic Security Service Application Program Interface . . . 8
2.3 Usable protocols . . . 12
2.3.1 The Kerberos protocol . . . 13
2.3.2 Simple and Protected Negotiation Mechanism . . . 16
2.4 Java Card . . . 18
3 **Design** . . . 22
3.1 Service Ticket Authorization Library . . . 22
3.2 Applet for a smart card . . . 30
3.3 Service Ticket Saver . . . 31
3.4 Card Terminal Demo . . . 32
4 **Solving the task** . . . 33
4.1 Configuration of systems . . . 33
4.1.1 Windows server . . . 33
4.1.2 Windows client . . . 34
4.1.3 Linux machine . . . 35
4.2 Work process . . . 36
4.3 Arisen problems . . . 37
5 **Created applications** . . . 42
5.1 Applications . . . 42
5.1.1 Service Ticket Saver . . . 42
5.1.2 SPNEGO Client Demo Application . . . 43
5.1.3 Card Terminal Demo Application . . . 43
5.2 Library . . . 43
5.2.1 StorageApplet . . . 43
6 **Conclusion** . . . 45
Bibliography . . . 47
A **Browsers with SPNEGO support** . . . 50
B **Print screens of created applications** . . . 52
C **Licence** . . . 56

D **Contents of the attached CD** 57

Chapter 1

Introduction

Information technologies are widely used in our society. Companies store important business data in various kinds of information systems. To protect information stored in these systems, only authorized employees should have access to them. Many companies also use more than one information system, so an employee may need to work with several applications during the workday. Therefore a requirement of common authentication to these systems has appeared, so that employees do not have to remember as many passwords as many applications they are using. In these days, there are many Single Sign–On[1] (*SSO* in short) solutions out there solving this problem. They allow users to log-in once, when they come to work, and then work with all the applications they need, without being prompted for other authentication.

Companies do not have only applications and information systems, they need to protect. They have also other resources they need to protect from being misused by unauthorized people, or employees for their personal use. Printers/scanners or any other equipment can serve as an example. Only certain employees should have access to those devices. There are solutions for this purpose too, but they usually require some kind of it's own infrastructure to be working. Many companies already have some infrastructure that could be used for this purpose, so the cost of the solution might be lower if their infrastructure could be used.

Why not to combine all these requirements together and have an environment, where a user comes to work, logs-in once and obtains all the authorizations he/she needs at once? So when he/she needs to use some application, system or even a device, he/she has access to it, because authorization has already took place.

To perform all the possible authorizations at once would neither be practical, nor safe, but users actually do not care when the authorization takes place. They only perceive how many times, they have to enter their password to be able to do their job. So when they log-in, only an authentication can take place and they might obtain some kind of a token, and the authorization can be performed when needed. The authentication token can be used for the authorization request to authenticate the user, instead of prompting for user's password. Also an authorization token may be retrieved for the requested service and then used all day to authorize user's access to the service.

---

1. Single Sign–On <http://en.wikipedia.org/wiki/Single_sign-on>

Company Y Soft[2] is one of the leading companies in printing solutions with their *SafeQ*. Their customers often run *Windows Server*[3] with *Active Directory*[4] (*AD* in short), which is used for centralized management of users and computers. AD also supports the Kerberos[5] authentication protocol. The Kerberos protocol can be used for authentication to third party applications and will be described later in the text.

Clients of Y Soft raised a question, whether it could be possible to use the infrastructure, they already have, for authorization of printing or access to some other devices. Such a solution would be cheaper, because it would use the infrastructure, companies have already invested into. Y Soft has therefore published a topic of master's thesis **Advanced authentication in Java applications using the Kerberos protocol**.

The goal of the thesis is to elaborate the possibility and difficulty of such a solution, where a user logs-in to his/her workstation and in case he/she needs, a ticket for some service is saved on a flash drive. The ticket can be later used for authorization for this service on another PC or device. For example when printing a document, user's ticket for the printer would be saved on a flash drive and then used by the printer to grant access to it. Two applications or utilities should be created to demonstrate the solution, that can be used. One application, which will allow saving of the ticket for some service and second, which will demonstrate usage of the ticket to access the service.

In the first part, the requirements of Y Soft and their expectations will be discussed. Then possibilities of authentication and authorization in Java applications will be shown. Possible approaches to this issue will be presented together with their advantages and limitations. Then the protocols which can be used to accomplish the task, will be presented. The next chapter will be focused on the design of the solution. It will be followed by a chapter focusing on the process of the work and eventual pitfalls, which showed up. The fifth chapter deals with the applications which were created for the purpose of this thesis. The last chapter then summarises whole thesis.

2. Y Soft, Ltd. <http://www.ysoft.com/>
3. Windows Server 2008: Overview <http://www.microsoft.com/windowsserver2008/en/us/overview.aspx>
4. Active Directory Architecture <http://technet.microsoft.com/en-us/library/bb727030.aspx>
5. Kerberos: The Network Authentication Protocol <http://web.mit.edu/Kerberos/>

Chapter 2

Analysis

This chapter deals with the requirements of Y Soft and how they evolved. Then possible approaches to authentication and authorization in Java applications will be presented. Namely *Java Authentication and Authorization Service* and *Generic Security Service Application Program Interface*, that can be used in the solution, will be introduced. Their properties will be described together with their advantages and limitations. Some sample codes will also be provided to give a better picture of how they can be used. Finally protocols that might be useful and could be used in the solution will be described. Especially the Kerberos protocol, which is the base for this thesis.

2.1 Requirements analysis

At first the assignment was focussing on saving of a ticket for the requested service and the intended use of the saved ticket was not clearly specified. It was required to use the Kerberos protocol, which is supported in *Active Directory* in *Windows Servers* from *Microsoft*. This support in AD allows to request a ticket for a logged-in user without prompting the user for his/her password, because the user has already been authenticated in AD. It was required to have an application or utility which could be used to save user's ticket for the requested service on a flash drive or a smart card. After the ticket would be saved, the second application should be able to validate the ticket (check whether the ticket is valid).

After a discussion with Y Soft, the intention of using the ticket for authentication to third party applications has appeared. So instead of an application that validates the ticket, an application demonstrating how can be the saved ticket used to authorize a request to a third party application, is required. As an example of a third party application, *Internet Information Services*[1] (*IIS* in short) shipped with *Windows Server*s was selected, because it is widely used by customers. Also it should be possible to store more than one ticket for a user on a flash drive or a smart card.

In the end, two applications should be created. The first one should allow a user to save a Kerberos ticket for the selected service on a flash drive or a smart card, if a smart card reader is attached. That means also an applet for smart cards needs to be created to allow saving of tickets on smart cards. And the second application that will demonstrate using of the saved ticket for authorizing user's request to IIS.

---

1. Overview: The Official Microsoft IIS Site <http://www.iis.net/overview>

Later, last two requirements appeared. The first one was to support saving of a Kerberos ticket on a contactless smart cards. And the second one was to create a library, that will provide the core functionality of ticket loading and saving.

To accomplish the task, authentication and authorization support in Java should be used. Otherwise, all already existing functionality would have to be implemented again. It would be very unwise trying to implement all the Kerberos related communications, if it is already implemented well and provided for free with Java. To have a picture of possible alternatives of using existing libraries, options of authentication and authorization in Java will be presented in the following section.

2.2 Options of authentication and authorization in Java applications

Of course it would be possible to implement all the protocol communications from scratch. But it would be too time-consuming and unnecessary, because there is already implementation of the Kerberos protocol available in the standard edition of Java. It is not even necessary to work with the protocol itself, there is also an authentication and authorization service encapsulating protocol specifics, available in Java. The service is called *Java Authentication and Authorization Service* (*JAAS*) and will be described in this section. Also a generic application program interface providing generic access to security protocols is available in Java. The interface is referred to as a *Generic Security Service Application Program Interface* (*GSS–API*) and will be also introduced in this section.

2.2.1 Java Authentication and Authorization Service

Java Authentication and Authorization Service (*JAAS* in short) is a package, which can be used to reliably and securely determine who is executing the Java code and ensure only users with proper permissions can execute it. It has been available in Java™ 2 SDK since version 1.3 (J2SDK v. 1.3) as an optional extension package. Since version 1.4 it has been available as a standard package in *Java Standard Edition*. This section is based on the information from the *Java™ Authentication and Authorization Service (JAAS) Reference Guide* [1].

Subject

The most important class in JAAS is the class `javax.security.auth.Subject`, which encapsulates information about a single entity (e.g. a user or a service). It groups together entity's Principals, private credentials and public credentials. An entity may have more than one `Principal`, which identifies the entity, for example a student may have a name `Principal("John Doe")` and a faculty login `Principal("doe2")`.

Credentials are security related attributes, that might be contained in the `Subject`. They are divided into two groups, private credentials and public credentials. Private credentials are treated in more secure way because they contain sensitive data that needs to be

protected, such as private cryptographic keys of the entity. Public credentials represents publicly available informations, such as public cryptographic keys of the entity.

A `Principal` is represented through the interface `java.security.Principal`, both private and public credentials may be any Java object. However it is recommended for classes representing credentials to implement `javax.security.auth.Refreshable` and `javax.security.auth.Destroyable` interfaces.

There are two constructors for `Subject` class:

```
public Subject();
```

The first constructor creates an instance of `Subject` with empty sets of principals, public and private credentials. This constructor can be used to construct an instance, when principals and credentials are not known yet. It is used for example to initialize a login context in `javax.security.auth.login.LoginContext` for unknown Subject. `LoginContext` internally instantiates a new empty `Subject`, if no `Subject` instance is passed to it.

```
public Subject(boolean readOnly, Set principals,
               Set pubCredentials, Set privCredentials);
```

The second constructor creates an instance of `Subject` with preset principals and credentials. It can also construct an instance of `Subject` with immutable principals and credentials sets, when `readOnly` parameter is `true`. It can be used when all the information about the entity should be already known and we want to prevent any modifications of them in the future.

To run a thread as a Subject and to ensure the thread will only be run by authorized Subjects, the class `Subject` has two static methods: `doAs` and `doAsPrivileged`. Both methods come in two variations — with and without throwing a checked exception.

The first method `doAs` has the following two forms:

```
public static Object doAs(final Subject subject,
                          final PrivilegedAction action);

public static Object doAs(final Subject subject,
                          final PrivilegedExceptionAction action)
    throws PrivilegedActionException;
```

The method `doAS` associates the `Subject` with the current thread's access control context and then execute the action. The action can return any Java object, that is then returned by the `doAs` method to the caller. During the execution of the action an exception may be raised. The first method allows only runtime exceptions to be thrown in the action, the second one allows to throw any exception from the action code. The exception will be wrapped as `PrivilegedActionException`.

The second method `doAsPrivileged` has the following two forms:

```
public static Object doAsPrivileged(
                        final Subject subject,
                        final PrivilegedAction action,
                        final AccessControlContext acc);

public static Object doAsPrivileged(
                        final Subject subject,
                        final PrivilegedExceptionAction action,
                        final AccessControlContext acc)
   throws PrivilegedActionException;
```

The method `doAsPrivileged` is same as the `doAs` method except it associates the `Subject` with the access control context passed as a parameter and not the current thread's access control context.

When the `doAs` or `doAsPrivileged` method is invoked, the `Subject` is associated with the access control context and parameterless method `run()` of the action is called. Whatever the method returns is returned by the calling method `doAs` or `doAsPrivileged`.

LoginContext

The class `javax.security.auth.login.LoginContext` is used to authenticate a subject in the application. All the `LoginContext` constructors require a name as an index into configuration represented by the class `javax.security.auth.login.Configuration`. After instantiating a `LoginContext`, the `LoginContext` loads all the configured login modules, which implement `javax.security.auth.spi.LoginModule` interface. When `login()` method is invoked, all `LoginModules` attempt to authenticate the subject and associate `Principals` and credentials with the `Subject`, if the authentication is successful. The authentication status is then returned to the application and `Subject` may be retrieved by the application from the `LoginContext`.

```
// instantiate a~LoginContext
LoginContext lc = new LoginContext("configurationKey");

try {
    // try to authenticate the Subject
    lc.login();

    // get the authenticated Subject
    Subject subject = lc.getSubject();

    // use the Subject and do the work
    ...
```

```
    // logout when done
    lc.logout();
} catch (LoginException le) {
    // report an error
    System.err.println("Authentication failed: " + le.getMessage());
}
```

An example usage of a `LoginContext`

Configuration options

Which `LoginModules` should be used to authenticate the subject is based on the configuration file. Path to the file is set through a system property, so it can be changed when running a Java program by a command line parameter:

```
java -Djava.security.auth.login.config==jaas.config ...
```

This is very useful, when an application should be running with various configurations. The only thing that needs to be changed is the value of the parameter in the command launching the application. Other option is to set the system property in the application itself by invoking:

```
System.setProperty("java.security.auth.login.config",
                   "./jaas.conf");
```

This approach is useful when the application will be always executed with the same configuration file, so there is no need to specify it when launching the application.

The configuration file may contain configuration for more applications, each application uses different name in the configuration file. For every application multiple login modules can be configured with a different flag and module options. The flag determines whether authentication by the module is required/requisite/sufficient or optional. Module options are different for every `LoginModule` and can be found in the documentation of each module. For example for `Krb5LoginModule`[2], there are options that specify whether a ticket cache should be used to obtain tickets, what file should be used as a ticket cache and what principal should be used as the default one. Such a configuration file may look like this:

```
ServiceTicketSaver {
        com.sun.security.auth.module.Krb5LoginModule
                required
                useTicketCache=true;
};
```

2. Krb5LoginModule (Java Authentication and Authorization Service) <http://download.oracle.com/javase/6/docs/jre/api/security/jaas/spec/com/sun/security/auth/module/Krb5LoginModule.html>

```
AnotherApp1 {
        sample.SampleLoginModule required;
        com.sun.security.auth.module.NTLoginModule required;
};

AnotherApp2 {
        sample.SampleLoginModule sufficient;
        com.sun.security.auth.module.NTLoginModule sufficient;
};
```

Such a configuration file would configure the application using the key "ServiceTicketSaver" to require authentication using the module `Krb5LoginModule` and allowing to retrieve credentials from the default cache of the system. The difference between configuration entries for "AnotherApp1" and "AnotherApp2" is that in case of "AnotherApp1", both of the modules need to succeed to authenticate the user, while in case of "AnotherApp2" just one of the modules needs to succeed to authenticate the user. Actually, if the `SampleLoginModule` succeeds, the following module (`NTLoginModule`) does not even try to perform user authentication.

Summary

JAAS is suitable for authentication of the user who is running the application and to ensure some parts of the code will be run only by users with proper permissions (e.g. only by authenticated users). On the other hand, JAAS is not designed for authentication of the communicating parties or for secure exchange of messages. For that purpose *Generic Security Service Application Program Interface* has been designed. It will be presented in the following section.

2.2.2 Generic Security Service Application Program Interface

The other very important package that allows to work with security protocols in a uniform way in Java is `com.sun.security.jgss`, that brings *Generic Security Service Application Program Interface* (*GSS–API* in short) to Java. GSS–API was designed in *Common Authentication Technology working group* of the *IETF*[3] and firstly published in September 1993 in RFC 1508[4]. Its last revision is *GSS–API Version 2, Update 1*, as defined in RFC 2743[4].

The main benefit of using it, is that the applications can be written independently on the underlying security mechanism and therefore the security mechanism can be easily changed. The application code is also simpler, because all the protocol specific communications are handled by the GSS–API implementation. Security mechanism to be used is specified by the *Object Identifier* (*OID* in short) when creating a security context. Every security

3. Internet Engineering Task Force <http://www.ietf.org/>
4. RFC 1508 <http://www.ietf.org/rfc/rfc1508.txt>

mechanism has a unique OID assigned to it by *Internet Assigned Numbers Authority*[5]. The GSS–API allows to authenticate a client to a server and when mutual authentication is required, even the server to the client. It ensures authenticity of the messages sent between the peers and it also supports integrity and confidentiality of exchanged messages.

The GSS–API divides the initialization of a security context among the peers between two functions. The initiator has to call `GSS_Init_sec_context()`, which initializes a security context. And send the returned GSS–API token to the second peer. The second peer then has to call `GSS_Accept_sec_context()` providing it with the received GSS–API token as a parameter, to accept the initiated security context. If mutual authentication was required, the second peer then has to send the returned token back to the first peer, which will use it in the successor call of `GSS_Init_sec_context()` to authenticate the second peer. After this, both peers are authenticated and a security context is established.

After a security context has been established, messages between the peers can be transferred securely. To prepare a message for sending, sender passes it to the `GSS_Wrap()` function. The function `GSS_Wrap()` ensures message authentication, integrity and confidentiality (if it was requested during establishing the security context) and returns an encapsulated message, that can be sent to the second peer. After receiving the encapsulated message, the receiver has to pass it to the function `GSS_Unwrap()`. The function `GSS_Unwrap()` reverts the encapsulation done by the function `GSS_Wrap()`, deciphers the message (if confidentiality was applied), checks the message integrity and validates its authenticity (that it was sent by the other peer).

After message exchange is done, security context should be destroyed. To flush all the context–level information, server calls `GSS_Delete_sec_context()`. Server can also call it with optional token buffer to retrieve a GSS–API token that can be sent to the client. When client receives the token, it passes it to `GSS_Process_context_token()`, which deletes all the context–level information in the client system.

The Java bindings for GSS–API has been defined in RFC 2853[6]. The GSS–API in Java is described in the paper *Single Sign–on Using Kerberos in Java* [2], sample codes and configuration examples can be found in the lab series *Advanced JGSS Security Programming* [3].

The specification allows to use a system-wide GSS–API implementation as well as a custom implementation, which can support types of security mechanisms not supported in the system-wide implementation. The API also provides a flexible framework to manage GSS–API mechanisms. To support pluggability of mechanisms, GSS–API uses *Java Cryptography Architecture* (*JCA* in short). JCA is a framework for developing and accessing cryptographic functionality in the Java platform. It contains part of the *Java Security API* related to cryptography along with a set of conventions. The main benefit of it is that it has a provider–based architecture, that allows using of cryptography independently on the implementation used. Also more providers can support the same cryptographic operation,

5. IANA — Internet Assigned Numbers Authority <http://www.iana.org/>
6. RFC 2853 <http://www.ietf.org/rfc/rfc2853.txt>

but using different algorithms. Such an architecture allows to replace an algorithm with a new one absolutely transparently to the application. It can be handy if a new faster or more secure algorithm is developed. [5] A GSS–API mechanism can be added on the system level (available to all users of the framework), or per instance of the GSS–API. There are two standard security mechanisms defined for the GSS–API: *The Kerberos Version 5 GSS–API Mechanism*[7] and *The Simple Public-Key GSS–API Mechanism (SPKM)*[8]. Support of the Kerberos V5 mechanism is mandatory for all Java GSS–API implementations in J2SE.

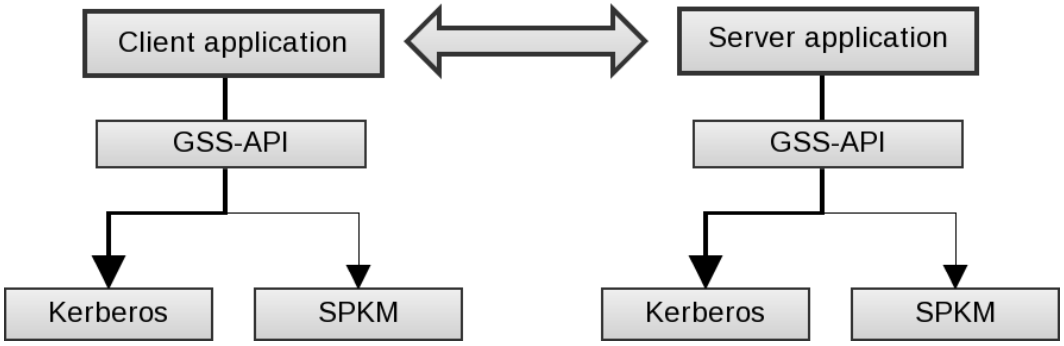

Figure 2.1: Use of multiple mechanisms in the GSS–API

The Java GSS–API framework consists of few components — `GSSManager`, `GSSName`, `GSSCredential`, `GSSContext` and `GSSException`. All security related functionality is delegated to the components from the security mechanisms.

GSSManager

`GSSManager` is the main class in the GSS–API. It can be used to list available mechanisms or to configure new providers. It also serves as a factory class for implementations of `GSSName`, `GSSCredential` and `GSSContext` interfaces. The default instance of the manager can be obtained by calling a factory method as follows:

```
GSSManager manager = GSSManager.getInstance();
```

GSSName

The `GSSName` interface represents an entity in Java GSS–API. It is a multi-mechanism container that lazily asks individual providers to perform the mapping when their mechanism is used. An implementation of the interface for a selected mechanism can be obtained by a factory method of the `GSSManager` as follows:

7. RFC 1964 <http://www.ietf.org/rfc/rfc1964.txt>
8. RFC 2025 <http://www.ietf.org/rfc/rfc2025.txt>

```
GSSName GSSManager.createName(String name, Oid nameType)
    throws GSSException;
```

The returned `GSSName` is mechanism independent, but the implementation should internally map it to a more mechanism specific form.

For example:

```
GSSName serverName = manager.createName("nfs@bar.foo.com",
                                        GSSName.NT_HOSTBASED_SERVICE);
```

The Kerberos V5 mechanism would internally map it to `nfs/bar.foo.com@FOO.COM`, where `FOO.COM` is the default realm, `bar.foo.com` is the name of the host machine and `nfs` is the name of the service.

GSSCredential

The `GSSCredential` interface is also multi-mechanism container as the `GSSName` interface. It represents the credentials of the entity. The implementation is obtained by a factory method of the `GSSManager` as follows:

```
GSSCredential GSSManager.createCredential(GSSName name,
                                          int lifetime, Oid mech, int usage)
    throws GSSException;
```

The type of the stored credentials depends on the specified mechanism and the requested usage. For example when the Kerberos V5 mechanism `Oid` is specified and requested usage is `GSSCredential.INITIATE_ONLY`, the mechanism would store an instance of a subclass of `javax.security.auth.kerberos.KerberosTicket` containing a TGT. On the other hand, if the requested usage is `GSSCredential.ACCEPT_ONLY`, the stored object would be an instance of a subclass of `javax.security.auth.kerberos.KerberosKey` containing the secret key.

GSSContext

The `GSSContext` interface is the one, which implementation provides services to the communicating parties. There are three factory methods of the `GSSManager` instantiating the context, one for the initiator, one for the acceptor and one for creating a previously exported context.

The initiator (the client) can obtain the implementation and then initialize the context as follows:

```
GSSContext GSSManager.createContext(GSSName peer, Oid mech,
                                    GSSCredential clientCreds,
                                    int lifetime)
    throws GSSException;
```

```
byte[] GSSContext.initSecContext(byte[] inToken, int offset,
                                 int len)
    throws GSSException;
```

The initiator has to specify the target peer. Desired mechanism and its credentials can be `null` to use the default ones.

The acceptor (the server) can obtain the implementation and then accept the context by the following API calls:

```
GSSContext GSSManager.createContext(GSSCredential serverCreds)
    throws GSSException;

byte[] GSSContext.acceptSecContext(byte[] inToken, int offset,
                                   int len)
    throws GSSException;
```

The acceptor may specify only its own credentials or pass `null` to use the default ones. All the other parameters are specified by the initiator of the context and the acceptor only accepts them.

GSSException

The `GSSException` is thrown by most methods in the framework. It encapsulates the original exception occurred within the GSS–API framework or mechanism providers.

Summary

The Java GSS–API is suitable for authentication of the communicating parties and for secured message exchange. It can be also configured to use the credentials of the logged-in user, if the authentication mechanism supports the credential cache of the system. The Kerberos V5 mechanism supports it, so it can be used with Kerberos without JAAS. On the other hand the Java GSS–API can not guarantee that an action will be executed only by an authorized user. So in some cases, combination of JAAS and GSS–API is needed.

2.3 Usable protocols

The thesis is focused on using the Kerberos protocol, so it will be described in short in the following section. Especially its concept and basic architecture. Also the concept of the names used in the protocol will be described. To use a previously saved Kerberos ticket, the Kerberos protocol may be used directly, but also some negotiation protocol may be useful to achieve that. Therefore *Simple and Protected Negotiation Mechanism* will be also introduced in this section as such a mechanism, that can be used along with Kerberos.

2.3.1 The Kerberos protocol

Networks in these days can be easily eavesdropped by insiders and sometimes even outsiders, so sending a plain password over such networks would be very dangerous. Therefore authentication mechanisms that do not transport password in a plain form has been developed. Also sometimes two peers need to authenticate each other, but should not know each other's passwords. To satisfy these requirements, the Kerberos protocol was developed in mid eighties. The Kerberos protocol is a distributed authentication service designed to provide reliable prove of client's identity to a verifier (typically a service) over an insecure network that can be eavesdropped. Kerberos was developed as part of *Project Athena*[9] on *Massachusetts Institute of Technology* (*MIT*).

The main idea is to use tickets with limited lifetime issued by trustworthy third party to prove client's identity to a service. Every user and service shares a secret key with the *Authentication Service* (*AS* in short). The shared key is usually a hash of the client's password (potentially concatenated with a salt). Clients can request from AS ticket-granting ticket (TGT) or ticket for any other service. Under the term TGT a ticket with a Service Principal Name "krbtgt/REALM@REALM" (where *REALM* is a Kerberos realm) is meant. TGT can be later used to obtain tickets for other services from the *Ticket Granting Service* (*TGS* in short). The AS and the TGS are usually implemented as one application, sometimes called *Key Distribution Center* (*KDC* in short) with two entry points. One for the authentication component and one for the ticket granting component.

There are two important terms in the Kerberos protocol that still need to be clarified. The first is the term *realm*. A realm is a designation of a virtual space, which is administrated by a certain Kerberos authentication server (or KDC). A realm is an upper-case string usually derived from the domain name. So a domain *domain.localhost* would probably use a Kerberos realm *DOMAIN.LOCALHOST*. The second one is the term *Service Principal Name*. A Service Principal Name (*SPN* in short) is a unique designation of a service. An SPN composes of three parts — a service type, a machine name and a Kerberos realm. The service type can be for example a protocol name (e.g. LDAP, HTTP). The machine name can be a name of the computer (e.g. server1), or a full domain name (e.g. server1.domain.localhost). The realm is a Kerberos realm the service is located under.

If clients want to communicate with a service, they need to prove their identity to the service. To do that using the Kerberos protocol, they have to send a Kerberos service ticket along with the request. If they do not have a service ticket for the requested service yet, they need to retrieve it first from a TGS. To retrieve a service ticket from the TGS, they need to prove their identity to the TGS using the TGT. If they do not have even a TGT, they need to retrieve it first from the AS and follow the steps in the reverse order. Or they can request a service ticket for the desired service directly from the AS, if they do not wish to receive a TGT, they could use for further service ticket requests.

9. Athena at MIT | IS&T <http://ist.mit.edu/services/athena>

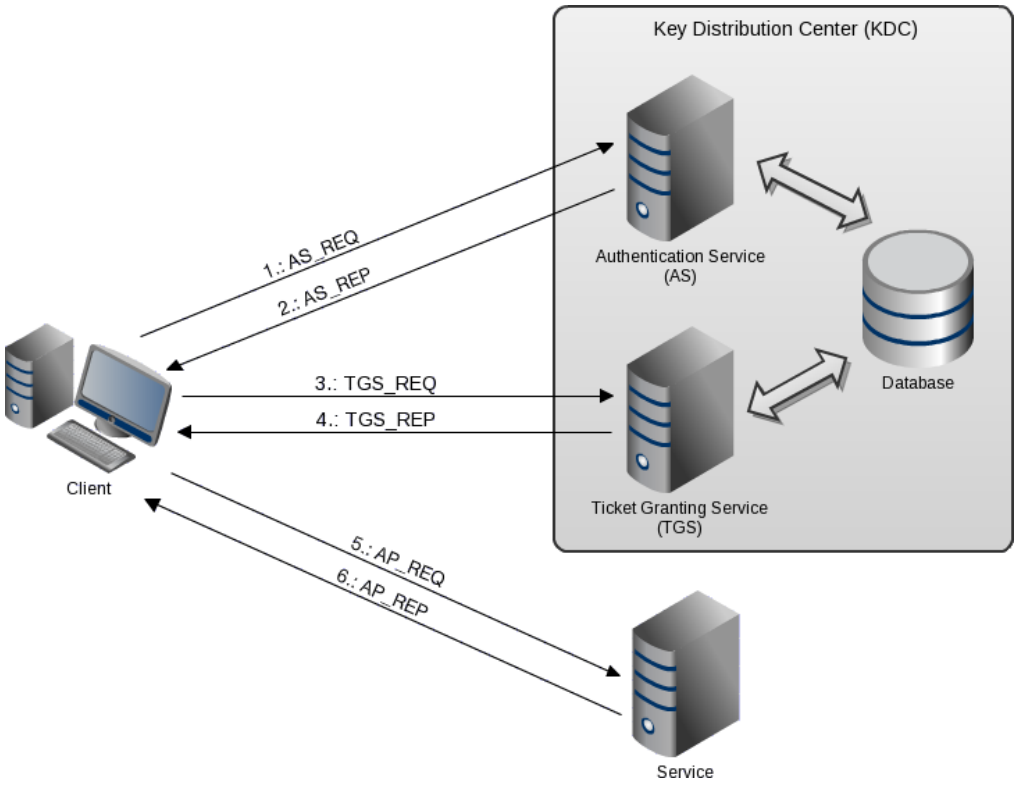

Figure 2.2: The protocol Kerberos steps

All the steps from authentication to application request are depicted on Figure 2.2 and are as follow:

1) Authentication Server Request (AS_REQ)

To authenticate himself/herself, the client has to send a plain text request to the AS. The request consists of client's principal name, principal name of the desired service, list of client's IP addresses and desired lifetime.

$$AS\_REQ = (Principal_{Client}, Principal_{Service}, IP\_list, Lifetime)$$

$Principal_{Service}$ is usually in a form "krbtgt/REALM@REALM" to receive a TGT that can be used in further requests for service tickets. But it can also be the principal name of the service, if client wants to use only that service.

2) Authentication Server Reply (AS_REP)

The AS sends the client a reply containing session key SK_{TGS} encrypted by client's secret key K_{Client} and a TGT encrypted by the secret key of the Service (usually TGS) K_{TGS}.

$TGT = (Principal_{Client}, krbtgt/REALM@REALM, IP\_list, Timestamp, Lifetime, SK_{TGS})$

$AS\_REP = (\{Principal_{Service}, Timestamp, Lifetime, SK_{TGS}\}_{K_{Client}}, \{TGT\}_{K_{TGS}})$

From the reply, the client can obtain SK_{TGS}, but can not decrypt the TGT encrypted by K_{TGS}. That means the client is not able to change the TGT, but can obtain SK_{TGS}, which is also stored in the TGT.

3) Ticket Granting Server Request (TGS_REQ)

To obtain a service ticket for the desired service, the client has to send a request to the TGS.

$Authenticator = \{Principal_{Client}, Timestamp\}_{SK_{TGS}}$

$TGS\_REQ = (Principal_{Service}, Lifetime, Authenticator, \{TGT\}_{K_{TGS}})$

The request composes of $Authenticator$, which proves client's knowledge of SK_{TGS} to the service, and a TGT which the client obtained from the AS.

4) Ticket Granting Server Reply (TGS_REP)

After validating the TGS_REQ, TGS generates a reply TGS_REP containing a session key $SK_{Service}$ encrypted by SK_{TGS} and a service ticket $T_{Service}$ encrypted by a service secret key $K_{Service}$.

$T_{Service} = (Principal_{Client}, Principal_{Service}, IP\_list, Timestamp, Lifetime, SK_{Service})$

$TGS\_REP = (\{Principal_{Service}, Timestamp, Lifetime, SK_{Service}\}_{SK_{TGS}}, \{T_{Service}\}_{K_{Service}})$

From the reply, the client can obtain $SK_{Service}$, but can not decrypt the service ticket encrypted by $K_{Service}$. That means the client is not able to change the service ticket, but can obtain $SK_{Service}$ also stored in the service ticket.

5) Application Request (AP_REQ)

Finally the client can use the received service ticket and send a request to the service.

$Authenticator = \{Principal_{Client}, Timestamp\}_{SK_{Service}}$

$AP\_REQ = (Authenticator, \{T_{Service}\}_{K_{Service}})$

The request composes of $Authenticator$, which proves client's knowledge of the $SK_{Service}$ to the service, and the service ticket encrypted by $K_{Service}$ obtained from the TGS.

6) Application Reply (AP_REP)
An optional step, in which the requested service proves its identity to the client, if mutual authentication has been requested.

If a service ticket is requested directly from the AS, steps 3 and 4 (*Ticket Granting Server Request* and *Ticket Granting Server Reply*) are skipped during the communication.
[6]

In the version 5 of the Kerberos protocol (Kerberos V5), support for cross–realm operations was added. Cross–realm operations allow a user from one organization (realm) to be authenticated to a service in another organization (realm). To support cross-realm authentication, the two TGS have to exchange inter–realm keys. Also a realm hierarchy can be created, so even if two TGS do not share an inter–realm key, they will still be able to authenticate each other's users, because there exists a path through some higher common realm.
[7]

2.3.2 Simple and Protected Negotiation Mechanism

Simple and Protected GSS–API Negotiation Mechanism (often referred as *SPNEGO*) is, as the name suggests, a negotiation mechanism designed for GSS–API. The GSS–API itself allows one of the communicating peers to authenticate itself to the other one, or both of the peers to be authenticated mutually. However it does not prescribe how should the peers agree on a common security mechanism supported by both of them to be able to do that. Therefore SPNEGO was later designed. It was defined in October 2005 in RFC 4178[10]. SPNEGO is a pseudo mechanism that can be chosen as a security mechanism for GSS–API by the initiator. It allows to negotiate a common security mechanism with the second peer, that will be used for authentication itself.

The following text is based on the specification of SPNEGO – RFC 4178 [8].
SPNEGO uses the following model:

- The initiator proposes a list of supported security mechanisms to the second peer (acceptor).
- The acceptor looks up the proposed list for mechanisms supported also by it and choose the first common mechanism in the list, or rejects the proposed values, if none is supported.
- The acceptor then informs the initiator about its choice.

10. RFC 2025 <http://www.ietf.org/rfc/rfc4178.txt>

The initiator starts a negotiation by sending an ordered list of available security mechanisms in the first negotiation token. Supported security mechanisms are listed in descending order of preference (the preferred one is the first). Thanks to this, peers can easily choose the most preferred security mechanism supported by both of them just by going through the list and selecting the first common mechanism. The initiator may also send an initial mechanism token for the preferred mechanism within the negotiation token, thus speeding up the authentication process, if the second peer accepts the preferred mechanism. If the preferred mechanism is accepted by the acceptor of the negotiation token, the contained initial mechanism token can be used to start the authentication process. Such a solution saves one message exchange because otherwise the acceptor would have to send a confirmation of the preferred mechanism and the initiator could only after that send the initial mechanism token for the selected mechanism.

Because SPNEGO is a pseudo mechanism for GSS–API, it uses methods defined for GSS–API to accomplish the negotiation. When the initiator wishes to use SPNEGO, the application invokes `GSS_Init_sec_context()` as normal and specifies that it wants to use the SPNEGO mechanism. Based on the credentials provided during this context establishment, the GSS–API implementation generates a negotiation token containing a list of one or more available security mechanisms. The initiator application then sends the generated token to the second peer. When the second peer receives the token, it passes it to `GSS_Accept_sec_context()` as any other GSS–API initialization token.

One of the following states may be returned by the GSS–API implementation:

I) GSS_S_BAD_MECH
If none of the proposed mechanisms is acceptable.

II) GSS_S_CONTINUE_NEEDED
If MIC token exchange is required or at least one additional negotiation token from the initiator is needed to establish the context. For example if the initiator did not send an initial mechanism token for the preferred mechanism, or the acceptor selected some other mechanism than the one preferred by the initiator.

III) GSS_S_COMPLETE
If the initiator's preferred mechanism was selected and authentication based on the received initial mechanism token was successful.

SPNEGO is also used for Single Sign–On solutions for Intranet applications. IIS supports it, there is module for Apache web server and there is also a Servlet, which can be used for Single Sign–On in any Java EE application. Users use web browsers to access Intranet applications, so they are more interested in whether they can use their favourite web browser than what technology is used to achieve it. A list of browsers with SPNEGO support can be found in Appendix A.

2.4 Java Card

Java Card is a technology that allows small applications (called applets) written in Java to be run on devices with very limited memory and computing power, such as smart cards or USB tokens. It is widely used in SIM (*Subscriber Identity Module*) cards in cell phones and ATM (*Automated Teller Machine*) cards. Java Card Platform specification was developed by *Sun Microsystems* (now part of *Oracle*) and development still continues. The current version of the specification is *Java Card Platform Specification 3.0.1*[11]. The API is compatible with smart cards related international standards such as ISO 7816[12] (which specifies cards from their physical characteristics, such as sizes of cards and positions of contacts on them, up to interindustry data elements that can be used for communication with the card) and EMV[13] (stands for *Europay, MasterCard and VISA*), which defines the standard for credit and debit payment cards. Applets run in a secure environment in Java Virtual Machine (*JVM* in short) on the device (on the card). [9] That gives an opportunity to work with sensitive data (such as private keys) in a secure environment. The JVM also contains an applet firewall which prevents applets to access data of other applets. [10] So even if a malicious applet is uploaded to the card, it is guaranteed that data of other applets are not in any danger.

There can be several applets on the card. To distinguish them, every applet has its unique application identifier (*AID* in short). An AID is a sequence of bytes up to sixteen bytes long, where first four bits determine a category of the AID according to Table 2.1.

Table 2.1: Categories of application identifiers [11]

Value	Category	Meaning
'0' to '9'	—	Reserved for backward compatibility with ISO/IEC 7812-1
'A'	International	International registration of application providers (ISO/IEC 7816-5)
'B', 'C'	—	Reserved for future use by ISO/IEC JTC 1/SC 17
'D'	National	National registration of application providers (ISO/IEC 7816-5)
'E'	Standard	Identification of a standard by an object identifier (ISO/IEC 8825-1)
'F'	Proprietary	No registration of application providers (free to use)

AIDs for International and National categories are the most common, because such AIDs allow providers to ensure that their applets will have a unique AIDs. In opposite there can be many applets with the same AID from category F, because there is no regulation. AIDs for categories A and D consists of five bytes long Registered Application Provider Identifier (*RID* in short) and up to eleven bytes long Proprietary Application Identifier

11. Java Card 3.0.1 Platform Specification <http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html>
12. ISO 7816 Smart Card Standard, links to ISO7816 parts 7816 1-5 <http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx>
13. EMVCo <http://www.emvco.com/>

Extension (*PIX* in short). Each application provider offering international applications needs an international RID, which is issued by the *ISO/IEC 7816-5 Registration Authority* (TDC Services A/S). [12]

For international RIDs, the first quartet is set to the hexadecimal number 'A'. Whilst each application provider offering national applications needs a national RID, issued by a national registration authority.

For national RIDs, the first quartet is set to the hexadecimal number 'D', followed by three quartets specifying the Country Code according to the ISO 3166 (*203* for Czech Republic). Registration authority issuing national RIDs in Czech Republic is *CZECH OFFICE FOR STANDARDS, METROLOGY AND TESTING*. [13, 14] A typical application identifier for national and international formats is described in Table 2.2.

Table 2.2: A typical application identifier (AID) format

Registered Application Provider Identifier (RID)	Proprietary Application Identifier Extension (PIX)
5 bytes	0 to 11 bytes

When a card is inserted into a card acceptance device, an applet located on the card has to be selected by specifying its AID, before any communication with the applet may occur. Also PIN verification or some other authentication may be required before an applet accepts any other commands. An application communicates with an applet by exchanging small blocks of data — application data units (*APDU*s in short). APDU is usually up to 261 bytes long (4 bytes in header, 2 body bytes Lc and Le and up to 255 bytes in body data field). However newer cards support a feature called *Extended APDU*, which allows to transfer up to 32,767 bytes in APDU body data field. [15] An application sends a *Command APDU* to an applet and as a response receives a *Response APDU*. The Command APDU composes of a mandatory header and an optional body. The header contains following bytes: instruction class, instruction code and two instruction parameters. The optional body contains an additional data bytes whose meaning is applet specific. The body usually contains up to 255 bytes of data, unless an extended APDU is used. The Command APDU is depicted in Table 2.3.

Table 2.3: Command APDU format

Mandatory header				Optional body		
CLA	INS	P1	P2	Lc	Data field	Le

Fields description:

CLA	Instruction class — indicates the structure and format of the command and the response APDU
INS	Instruction code — specifies the instruction to be executed
P1, P2	Parameters of the instruction
Lc	Number of bytes sent as the data field of the command
Data field	A sequence of bytes sent as data to the applet
Le	Maximal expected size of the response data field

The Response APDU composes of an optional body and a mandatory trailer. The optional body contains an additional data byte array. The trailer contains two status words. The Command APDU is depicted in Table 2.4.

Table 2.4: Response APDU format

Optional body	Mandatory trailer	
Data field	SW1	SW2

Fields description:

Data field	A sequence of bytes sent as data from the applet
SW1, SW2	Status words — denote the processing state

More details and an example applet can be found in the article *How to write a Java Card applet: A developer's guide*. [16]

An application communicates with the selected applet according to the following scenario:

1. The application sends an application data unit (APDU) to the applet.
2. The applet processes the command and replies by sending a response APDU which contains status words (SW1 and SW2) indicating the result of the operation.
3. The applet may also send some data back to the application in the response APDU.
4. The application should check the status words and eventually process the data sent back in the response APDU.

When writing a Java Card Applet, card hardware limitations and memory model needs to be taken into account. Especially all data stored in persistent memory should be allocated in the applet constructor and transient arrays which will be stored in RAM should be allocated by `JCSystem.makeTransientByteArray()` or corresponding alternatives for other data types. Also writes to persistent memory are very slow and memory can only stand a limited number of writes, so writing to persistent memory should be avoided if possible.

Chapter 3

Design

This chapter deals with the initial design of the applications and its development. Division to packages and structure of important classes will be presented. Because two demonstration applications should be created, the core functionality should be provided in common packages and used by the applications. To support storing of tickets on a smart card, an applet for a smart card needs to be created as well. Later a requirement of creating a library encapsulating the core functionality has appeared. Therefore common packages should be packed in the library and some reasonable interface should be created for it. At first the design of the library and the applet will be presented and then the design of the applications will be shortly presented.

3.1 Service Ticket Authorization Library

The library should provide methods for saving of Kerberos tickets to a storage, for loading of a previously saved tickets from the storage and for generating an SPNEGO token that can be used in a service request for authorization. Because several types of storage may be used, all storage related functionality should be encapsulated in a separate package. On top of that, a package providing the core functionality should be created.

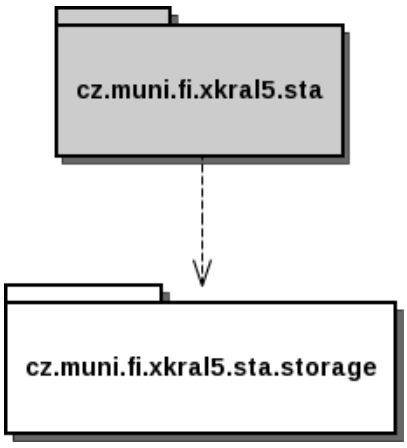

Figure 3.1: Packages of the Service Ticket Authorization Library

These two packages might be packed as a library, which would provide the required functionality to any application. The library packages are depicted on Figure 3.1, where the main package is `cz.muni.fi.xkral5.sta` and `cz.muni.fi.xkral5.sta.storage` provides the storage functionality for it.

Package cz.muni.fi.xkral5.sta

The package `cz.muni.fi.xkral5.sta` provides the core functionality — loading/saving of service tickets and generating of SPNEGO tokens from a previously saved service tickets. All the functionality is provided through methods of the only public class located within this package, the `ServiceTicketAuthorization` class. Methods of the class internally use other protected classes of the package and storage functionality from the sub–package `cz.muni.fi.xkral5.sta.storage`. The only exception thrown by the methods of the class is `AuthorizationException`, which wraps the cause of the exception. The `ServiceTicketAuthorization` class provides the following methods:

setLoginCallbackHandler(javax.security.auth.callback.CallbackHandler handler)
: Sets a login callback handler that should be used to retrieve credentials from the user, if valid credentials are not found in the system credential cache. If a login callback handler is not set and no valid credentials are found in the cache, the user is not authenticated and the invoked saving operation is terminated.

loadServiceTicket(java.security.Principal spn, Storage storage)
: Loads a previously saved Kerberos ticket from the given storage.

loadServiceTicket(java.security.Principal spn, ProtectedStorage storage, char[] pin)
: Loads a previously saved Kerberos ticket from the given storage locked via the given PIN.

saveServiceTicket(java.security.Principal spn, Storage storage)
: Retrieves and saves a Kerberos service ticket for the authenticated user for service specified by the spn to the given storage.

saveServiceTicket(java.security.Principal spn, ProtectedStorage storage, char[] pin)
: Retrieves and saves a Kerberos service ticket for the authenticated user for service specified by the spn to the given storage, which is/will be locked with the given PIN.

generateToken(java.security.Principal spn, Storage storage)
: Generates an SPNEGO token from a previously saved Kerberos ticket stored in the storage.

generateToken(java.security.Principal spn, ProtectedStorage storage, char[] pin)
: Generates an SPNEGO token from a previously saved Kerberos ticket stored in the storage locked via the PIN.

Package cz.muni.fi.xkral5.sta.storage

The package `cz.muni.fi.xkral5.sta.storage` provides the storage functionality for the library. Its initial class diagram is depicted on Figure 3.2.

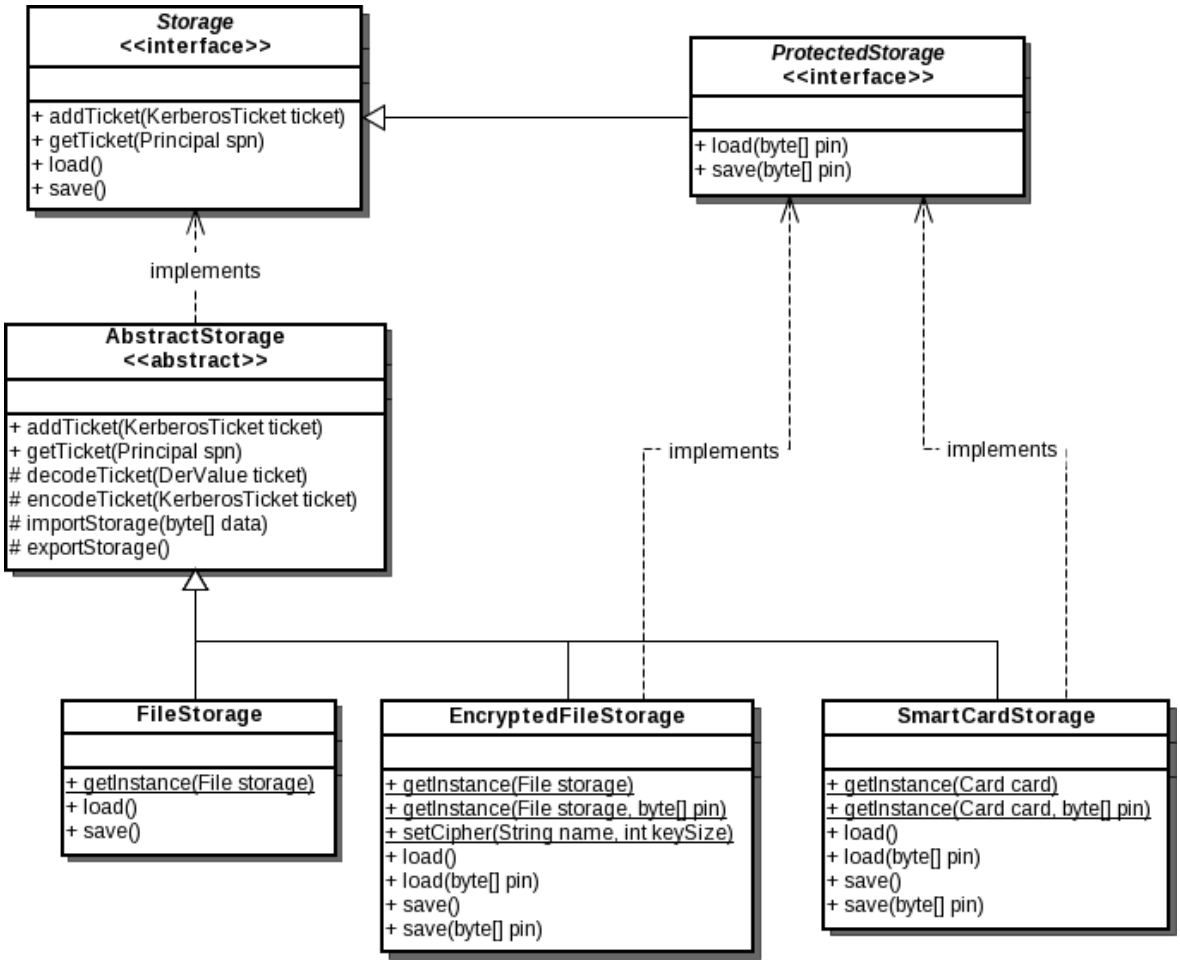

Figure 3.2: The initial class diagram of the `cz.muni.fi.xkral5.sta.storage` package

The package contains two interfaces: `Storage` and `ProtectedStorage`. The `Storage` interface prescribes basic functionality, whilst the `ProtectedStorage` interface extends the `Storage` interface and prescribes other methods for loading and saving of tickets using a PIN. The `Storage` interface is partially implemented by `AbstractStorage` abstract class, which provides the basic functionality. All implementations of the storage inherits the `AbstractStorage` and implements methods of the `Storage` or `ProtectedStorage` interface.

There are three implementations available in the package:

FileStorage
: Implements the `Storage` interface and allows to save tickets to a file and later load them from the file.

EncryptedFileStorage
: Implements the `ProtectedStorage` interface and allows to save tickets to an encrypted file and later load them from the file. A cipher and a key size can be set for encryption of the storage to provide required level of security. However it should be pointed out, that the key is generated from a PIN, so all possibilities can be tried to unlock the storage.

SmartCardStorage
: Implements the `ProtectedStorage` interface and allows to save tickets to a smart card containing a `StorageApplet` and later load them from the card. The applet can be protected via a PIN, with a limited number of tries, after which the applet becomes locked and can not be unlocked any more. This is the most secure way of storing tickets, because they can not be retrieved from the card even by trying out all possible combinations for the PIN.

The initial design was focused on the support for file storages, which was implemented first. Therefore there were methods `load()` and `save()`, that loaded/saved the whole storage. Such solution is however not so good for smart cards, which showed up during adding of a smart card support to the library. There is no reason why to load or save all tickets from/to smart card at once. Tickets stored on a smart card are stored securely, however once loaded to a computer, they can be compromised. Also loading of all stored tickets from a smart card to a computer would be quite time consuming. Loading/saving of a single ticked is therefore the only reasonable approach to using a smart card as a ticket storage. Methods `load()` and `save()` are then not necessary, because loading and saving of a ticket from/to a smart card can be invoked from methods working with a ticket — `getTicket()` and `addTicket()`. In fact, file storages can load the storage to memory with the first `getTicket()` call, thus method `load()` is not necessary for them either. The only usage which is left is using of a method `save()` with a file storage to save several added tickets at once. It is not a requested functionality, but if it was, it could be easily supported without the method `save()` just by introducing a method `addTickets()`, which would add a collection of tickets to the storage.

The re-factored class diagram of the package is depicted on Figure 3.3. The final design of the package is more transparent — its interface abstracts from the way how the data are actually stored and allows more straightforward using. It is not necessary any more to load the storage before a ticket can be requested, or save a storage after a ticket is added. Method `addTicket()` just needs to be called to add a ticket to the storage and `getTicket()` to load a ticket from the storage. For `ProtectedStorages` also a PIN verification method `verifyPIN()` was introduced, which showed to be very useful for more

natural using of storages in the applications. Without this method, verifying of a PIN was not very straightforward. An application needed to attempt to load a ticket from the storage to detect, whether the storage can be unlocked by the provided PIN. The package stayed internally intact, so tickets are encoded and saved in the same way as they were before re-factorization. Used encoding of tickets is described in the following section.

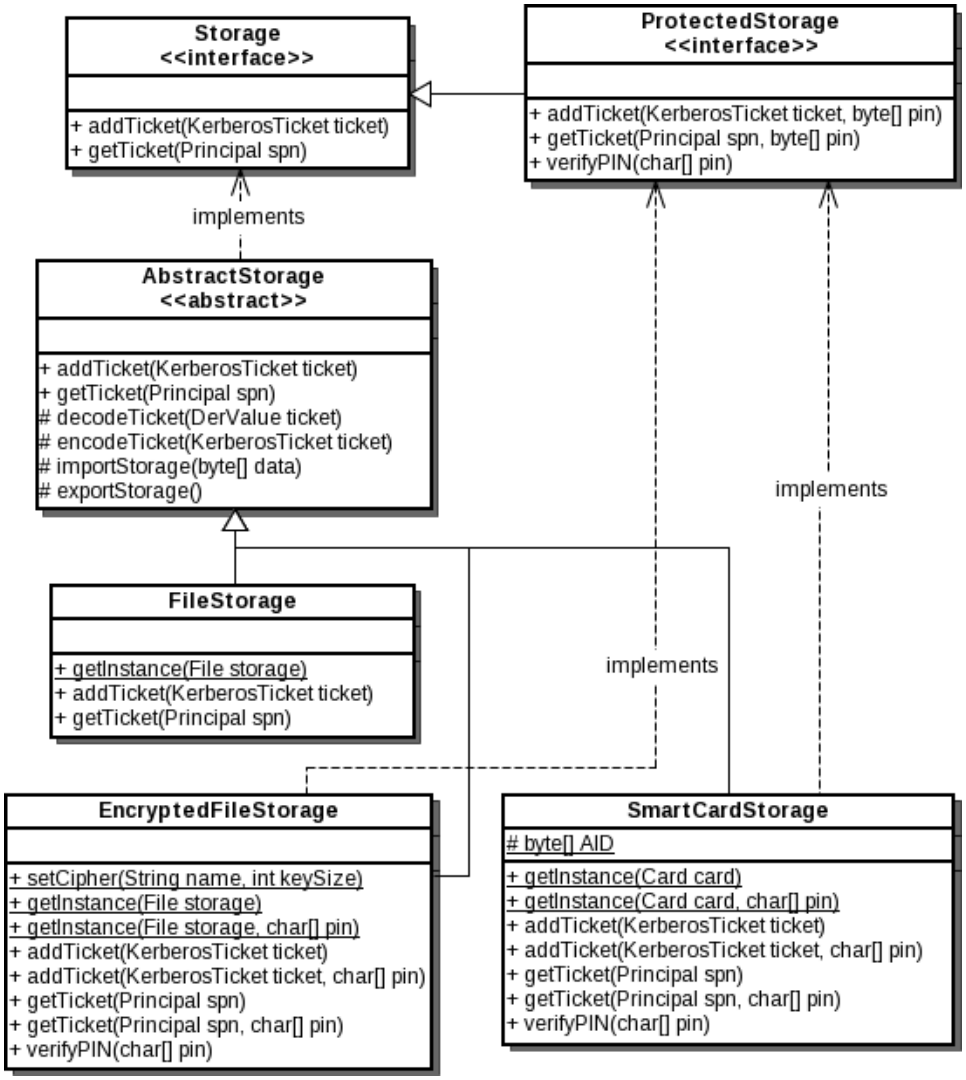

Figure 3.3: The re-factored class diagram of the `cz.muni.fi.xkral5.sta.storage` package

Encoding of tickets in a storage

Tickets are stored in the memory of a computer as Java objects. To be stored to a storage, they need to be somehow encoded. Otherwise later it would not be possible to load them back from the storage. While encoding an object for storing, the only things that need to be stored are object's attributes. If object's attributes are stored and it is known what class is the object, the object can be restored by instantiating a new object and filling its attributes with the stored values.

Several different common methods of encoding come to mind:

Serialization

Using of serialization is the simplest way. Java language provides an output stream `ObjectOutputStream` for serializing of objects with `writeObject()` method, which serializes the object passed as a parameter and write it to the output stream. And an input stream `ObjectInputStream` with `readObject()` method for retrieving of a previously serialized object. [17] Serialization is however language dependent and also the serialized data are quite large, so it is not convenient when an exchangeable format for different languages is needed, or when a size of data matters.

Extensible Markup Language

The *Extensible Markup Language* (*XML* in short) is a text format designed for large-scale electronic publishing. [18] The format is nowadays widely used as an exchange format between various applications and is also very common on the Internet. This format, can be used for message exchange between applications written in different languages. It is also human readable and easy to generate and process. However XML data are pretty large (even more that serialized) because of the human readability of the format. Therefore the XML format is not convenient when the data size matters.

Distinguished Encoding Rules

The last of the common formats, which comes to mind is *Abstract Syntax Notation One* (*ASN.1* in short). ASN.1 is a flexible abstract notation, which allows to specify the structure of the data for representing, encoding, transmitting and decoding. However it does not specify a concrete way the information should be encoded. To address this, several encoding rules has been developed for ASN.1.

List of encoding rules for ASN.1:

- X.690[1]: Basic Encoding Rules (BER)
- X.690: Canonical Encoding Rules (CER)
- X.690: Distinguished Encoding Rules (DER)
- X.691[2]: Packed Encoding Rules (PER)

1. ITU–T Rec. X.690 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>
2. ITU–T Rec. X.691 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.691-0207.pdf>

- X.692[3]: Encoding Control Notation (ECN)
- X.693[4]: XML Encoding Rules (XER)
- RFC-3641[5]: Generic String Encoding Rules (GSER)

[19]

Distinguished Encoding Rules is a constrained version of *Basic Encoding Rules* same as *Canonical Encoding Rules* but it uses definitive lengths form. It is widely used in cryptography, especially in *X.509* digital certificates.

Because it is a standard format, which is space saving, this format has been chosen for encoding of tickets in storages.

For proper working, not all attributes of `KerberosTicket` class need to be stored. If only the required attributes are stored, the size of the data may be reduced. Ticket authentication time, start of validity time and renew until time are not necessary and can be opt out. Correct values are also stored in the encrypted ticket for the service, so opting out does not pose any security risk. Also client addresses where the ticket can be used are not necessary for correct working of authorization. If the ticket can not be used from some addresses, the requested service rejects the request. Because the user has no other way how to authenticate himself/herself on such devices, rejecting of the service is the only correct result of such a request. Also tickets retrieved from KDC in AD do not limit the addresses by default. Last elements that do not need to be explicitly saved are a service realm and a service principal name, because those elements are already contained within the `ticket` element in a plain text format.

To hold the required data, the following format for a Kerberos ticket has been chosen:

```
Kerberos-Ticket ::= SEQUENCE {
      client          [0]  PrincipalName,
      endTime         [1]  KerberosTime,
      keyType         [2]  UInt32,
      sessionKey      [3]  OCTET STRING,
      flags           [4]  KerberosFlags,
      ticket          [5]  Ticket
}
```

ASN.1 representation of a truncated Kerberos ticket

3. ITU–T Rec. X.692 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.692-0203.pdf>
4. ITU–T Rec. X.693 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.693-0112.pdf>
5. Generic String Encoding Rules (GSER) for ASN.1 Types <http://www.ietf.org/rfc/rfc3641.txt>

Description of `Kerberos-Ticket` elements:

client
: The principal name of the client requesting the ticket. The `PrincipalName` class provides `asn1Encode()` method to get an ASN.1 representation of the principal name.

endTime
: The expiration time of the ticket. The `KerberosTime` class provides `asn1Encode()` method to get an ASN.1 representation of the time.

keyType
: The key type of the session key. It is an integer value, which can be put to the `DerOutputStream` directly by `putInteger()` method.

sessionKey
: The session key shared by the client and the service. The session key is represented by a `SecretKey` class, which provides `getEncoded()` method to get an encoded representation of the key in its primary format, which is returned as a byte array.

flags
: Flags of the Kerberos ticket. It is an array of boolean values, which can be converted to `BitArray` and put to the `DerOutputStream` by `putUnalignedBitString()` method.

ticket
: The ticket for the service containing an encrypted part, which only the service can decrypt. The `KerberosTicket` class provides `getEncoded()` method to get an ASN.1 representation of the ticket according to the Kerberos protocol specification:

```
Ticket ::= SEQUENCE {
      tkt-vno       [0] INTEGER (5),
      realm         [1] Realm,
      sname         [2] PrincipalName,
      enc-part      [3] EncryptedData -- EncTicketPart
}
```

: It contains a ticket version number, a service realm, a service principal name, and an encrypted ticket part for the service.
[20]

Because storing of tickets to a file is required and more than one ticket should be stored in a storage, an ASN.1 representation of a whole storage has been designed. All the required information about a single ticket can be retrieved from the `Kerberos-Ticket`, so the ASN.1 representation of the storage can be just a sequence of tickets.

```
Storage ::= SEQUENCE OF Kerberos-Ticket
```

ASN.1 representation of the storage

3.2 Applet for a smart card

The applet `StorageApplet` should support saving of service tickets to a smart card and loading tickets from the card. It should be possible to store more than one ticket on the card, so it should be also possible to look up the desired ticket on the card. Because the card would belong to a certain user, it is sufficient to look up tickets on the card just by the Service Principal Name (*SPN* in short) of the service. To save space and have an easy logic of looking up tickets, when saving a ticket with an SPN for which a ticket is already stored on the card, the stored ticket should be replaced with the new one. Also because a card can hold only a limited number of tickets, some kind of replacing of tickets with new ones needs to be implemented.

Saving of a ticket

To support saving of multiple tickets on a card, several slots (ticket storages) needs to be used in the applet. When using multiple ticket storages, one needs to be selected to hold the ticket which is being saved. To ensure replacing of a ticket with the new one, ticket with the same SPN needs to be looked up between currently stored tickets. If no ticket with matching SPN is found, the least recently used ticket storage should be used. That means a slot that has not been used yet, or the slot which has not been read from or written to for the longest time.

It is also important to ensure that a ticket, which is being saved is not marked as valid until it is completely saved. Otherwise a corrupted ticket would be later retrieved from the card and used for authorization. To ensure that without degradation of performance, the size of the uploaded ticket has to be compared with the real one before the ticket can be considered as successfully stored.

To support the required functionality, saving of the ticket is divided between the following three commands:

INS_SAVE_START
: This command initializes the saving process. If a ticket storage index is 0x00, it tries to find a storage containing a ticket with the given SPN. If there is no ticket with the given SPN already stored, the least recently used ticket storage is selected for storing the ticket. If some other ticket storage index is used, the specified storage is used.

INS_SAVE_UPLOAD
: This command appends next data block to the ticket which is being saved. The saving process had to be started by `INS_SAVE_START` command.

INS_SAVE_COMPLETE
This command finishes the saving process. It is there to guarantee, that the ticket was saved completely. Until this command is successfully finished, the ticket is considered to be corrupted (even if all its bytes were transferred).

Loading of a ticket

Loading of a ticket is much simpler than saving. The ticket just needs to be looked up between stored tickets and then transferred from the card in several data blocks. To support this functionality, only one command is sufficient:

INS_LOAD
This command allows to load a previously saved ticket. If the ticket storage index is 0x00, it tries to find a storage containing a ticket with the given SPN. The ticket is obtained in multiple blocks (chunks) and remaining ticket size is always indicated in the returned data. Also the ticket storage index is indicated in the response, so that the ticket needs to be looked up only once.

3.3 Service Ticket Saver

The application should allow a user to retrieve and save a Kerberos ticket for the requested service. It can be a simple GUI application, that will just allow to select a storage and then use the *Service Ticket Authorization Library* to save the ticket to the selected storage.

The user should be able to choose whether to store the ticket on a smart card, or in an encrypted file located on a flash drive selected from a list of connected flash drives. To get an accurate list of flash drives, a library written in other language than Java, which can access system API, would have to be used. But for this demonstration application, it is sufficient to list file system roots and filter out roots of network drives on Windows systems and list mount points from certain directory on *nix systems.

Graphical User Interface

The Graphical User Interface (GUI) of the application should be simple. There is no need for any configuration, so the user should be able just to select whether to store the ticket on a smart card, or on a flash drive. In case of a flash drive, select the flash drive from a list of connected flash drives and in both cases enter a PIN. Therefore the application can be composed from two tabs, one for storing of a ticket on a smart card and another for storing of a ticket on a flash drive.

The first tab for saving of a ticket on a smart card can be used only if a smart card reader is attached, therefore the tab should be disabled if a smart card reader is not attached. It could be also hidden, but then a user would not see, that saving to a smart card is also available. The tab needs to contain an input for entering a PIN and a button for saving of a ticket.

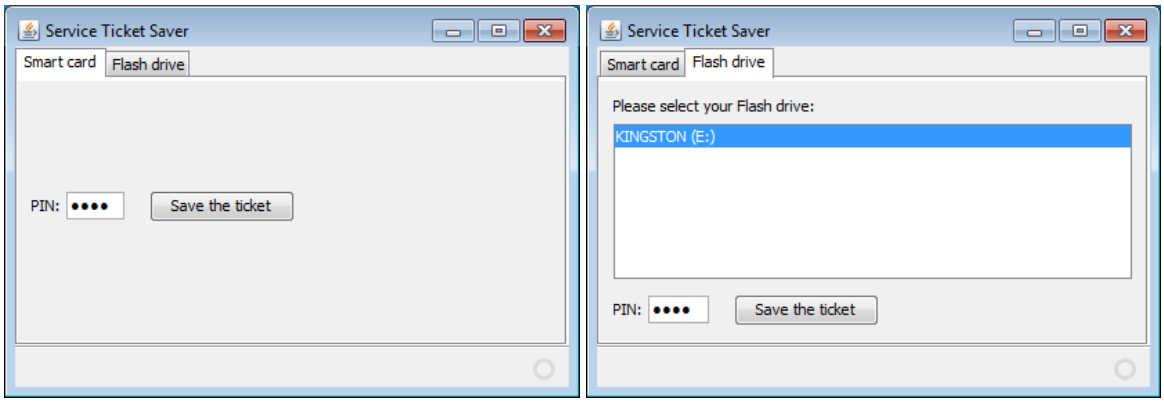

Figure 3.4: Smart card and Flash drive tabs of Service Ticket Saver application

The second tab needs to contain a list of connected flash drives for selecting the desired one, an input for entering a PIN and a button for saving of a ticket. Generated PINs are desired, so a PIN field should be enabled only if the selected flash drive already contains a storage file. Otherwise a new PIN should be generated and displayed to a user when a ticket is saved.

3.4 Card Terminal Demo

The application should demonstrate how the *Service Ticket Authorization Library* can be used in an application, which is awaiting a card and use the ticket stored on it to authorize a service request. The application might be composed from the following three classes:

ServiceAction
: This class represents the action that should be performed. When an action is run, a service ticket is retrieved from the card and a service request is performed.

TerminalPooler
: This class represents a card terminal pooler. It is a thread that periodically checks the card terminal for presence of a card. When a card is attached, the action is invoked.

Main
: This class is the main class containing static `main(String[] args)` method. When the application is started and the main method is invoked, a `TerminalPooler` thread for each card terminal is started.

Chapter 4

Solving the task

The first thing that needed to be done during the solving of the task, was the preparation of the environment. The environment should have been composed from a network with Windows server, Windows client and Linux machine. The Windows server with *Active Directory* installed should have served also as a *Key Distribution Center* for the Kerberos protocol. *Windows Server 2008* had been selected as a Windows server. The Windows client had been used for storing of a ticket for a service. It should have been connected to the *Active Directory* domain, so that the user who logs-in can automatically obtain a Kerberos TGT and therefore save a Kerberos ticket for a service without being prompt for credentials. *Windows 7* had been selected as a Windows client. The Linux machine should have served as a machine, where the user could have applied the ticket to make a service request. Because I was using system Fedora during that time, the system *Fedora 12* I have had already installed, was used.

To be able to run three systems at the same time on one computer, a virtual environment needed to be used. Because Linux was the hosting operating system, it did not need to be run in a virtual environment. The Windows server and client were installed into *VirtualBox*[1] virtual machines. A virtual network between the virtual machines and the hosting operating system was created, so that all the systems could communicate with each other.

4.1 Configuration of systems

The virtual network between systems used an address **192.168.56.0** and the name of the domain was **domain.localhost**, addresses and names were assigned as follows:

- Linux machine: **delisek**.domain.localhost (**192.168.56.1**)
- Windows server: **ws2008**.domain.localhost (**192.168.56.101**)
- Windows client: **virtual7**.domain.localhost (**192.168.56.102**)

4.1.1 Windows server

Support for the Kerberos protocol is integrated into *Active Directory*, so only thing that needs to be done is to select supported Kerberos encryptions on the Account tab of a user profile in the application *Active Directory Users and Computers*.

1. Oracle VirtualBox <http://www.virtualbox.org>

In a Kerberos environment clients uniquely identify services by their *Service Principal Names* (*SPN* in short). SPN composes of a service type, machine name and a realm. The service type can be for example a protocol name (e.g. LDAP, HTTP). The machine name can be a name of the computer (e.g. *virtual7*), or a full domain name (*virtual7.domain.localhost*). The realm is a Kerberos realm, which can be omitted, if a default one is configured on the computer. On computers connected to a Windows domain, the name of the domain is used as a default realm by default. SPN is mapped to a user account or a computer account on a Windows server via the `Setspn` application. For proper working in Windows domain, SPN for both computer name and full domain name should be registered.

Application should have been tested against Microsoft *Internet Information Services* (*IIS* in short), so proper SPNs for it had to be registered via the `Setspn` application. To do that, following commands were used:

```
setspn –A HTTP/ws2008.domain.localhost ws2008
setspn –A HTTP/ws2008 ws2008
```

The first command added mapping for SPN *HTTP/ws2008.domain.localhost* (containing full domain name of the machine) to the computer account *ws2008*. The second one did the same but for SPN with machine name *HTTP/ws2008*.

To allow Kerberos authentication for a web domain in IIS, *Windows Authentication* has to be enabled at the domain *Authentication* panel. The *Windows Authentication* contains Kerberos and NTLM authentication mechanisms.

4.1.2 Windows client

On a Windows client, there is almost nothing to configure. The client just needs to be connected to the Windows domain. Once connected to the domain, the client is configured to use the proper *Key Distrubution Center* (*KDC* in short) and the proper realm.

However in newer versions of Windows, Microsoft does not export a session key nor its encryption type for tickets obtained from the native credential cache. Such a TGT cannot be used for obtaining service tickets. Luckily there is a registry entry, that can be set to allow exporting of a session key and its encryption type when retrieving a ticket from Windows native credential cache. The registry entry is `allowtgtsessionkey` and has to be set to `0x01` (type `REG_DWORD`). The location of the entry vary between different versions of Microsoft Windows. On the Windows Server 2003 and Windows 2000 SP4, the location is:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters

On the Windows XP SP2 and later systems, the location is:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos

[21]

4.1.3 Linux machine

On a Linux machine, Kerberos has to be configured to use the KDC of the Windows Server. In Linux, configuration of Kerberos is done in a configuration file */etc/krb5.conf*. The configuration file uses INI-style format and is divided to several sections. The most important sections are:

libdefaults
: This section contains default values for Kerberos V5 library. For example the default realm, ticket lifetime and renew lifetime can be configured in this section.

realms
: This section contains configuration of realms keyed by realm names. For each realm, KDC and administration server addresses can be configured.

domain_realm
: This section provides mapping from host names or domain names to Kerberos realms. If the name starts with a period "." (e.g. "*.example.com*"), it is a domain name. If not (e.g. "*example.com*"), it is a host name.

For the Linux machine to use the Windows domain Kerberos realm as the default realm and the KDC and administration server running at the Windows server, Kerberos at the Linux machine had to be configured as follows:

1. KDC and administration server addresses had to be configured for the domain realm `DOMAIN.LOCALHOST`, which was the Kerberos realm of the Windows domain. Configuration was done by putting the following lines to the **realms** section of the configuration file:

```
DOMAIN.LOCALHOST = {
  kdc = ws2008.domain.localhost:88
  admin_server = ws2008.domain.localhost:749
}
```

2. Windows domain name `domain.localhost` had to be mapped to the Kerberos realm of the Windows domain `DOMAIN.LOCALHOST`. Configuration was done by putting the following line to the **domain_realm** section of the configuration file:

```
.domain.localhost = DOMAIN.LOCALHOST
```

3. The Kerberos default realm had to be set to the Kerberos realm of the Windows domain `DOMAIN.LOCALHOST` to support SPNs without specified realm. Also ticket lifetimes and forward-ability had to be set. Setting was done by putting the following lines to the **libdefaults** section of the configuration file:

```
default_realm = DOMAIN.LOCALHOST
ticket_lifetime = 24h
renew_lifetime = 7d
forwardable = yes
```

4.2 Work process

At first, information about Kerberos and its support in *Active Directory* and Java authentication and authorization support were collected. Then according to the found information, systems were configured. When the systems were configured, work itself could have started. At first the *Service Ticket Saver* application was prototyped and an application verifying tickets using the service keytab[2] file. The *Service Ticket Saver* application used JAAS for authentication of the user and authorized execution of actions retrieving and saving a ticket. The method used GSS–API to retrieve a service ticket. After this first prototype was presented to the Y Soft company, a demand of using a ticket for authorized request to a third party application was introduced. Also a request of storing multiple tickets to a file was added.

In the next stage, a storage package was developed encapsulating the logic of saving and loading of tickets. Also support for encryption of storage files was added. To test a saved ticket, a demonstration application *SPNEGO Client Demo* was created. The application used the previously stored ticket to make an authorized request to a website running on IIS. The application again used GSS–API to generate an SPNEGO token to make an authorized request. After a live demonstration of those two applications, the last requirement was introduced. The requirement was to use a contactless smart card to store tickets on.

In the next stage, an applet for smart cards was developed and the storage package was extended with a smart card support. Writing an applet was the most challenging part. At first only contact smart card was used for developing purposes. After the applet and the package were working fine together, a contactless smart card was tested. During that time the last requirement was introduced — requirement of developing a library that would encapsulate all the related functionality.

In the end, the packages that contained the core functionality were merged to a single package `cz.muni.fi.xkral5.sta` and a main package class providing the functionality to other applications was created. The class is `ServiceTicketAuthorization`. Also another demonstration application *Card Terminal Demo* for smart cards was implemented. This application awaited smart card and performed an authorized action, when a smart card was attached. As the last thing, following performance improvements of the applet were done:

- Putting all ticket slots to a one byte array.

2. What is a keytab, and how do I use one? <http://kb.iu.edu/data/aumh.html>

- Putting all SPNs to a one byte array.
- Pre-computing offsets to byte arrays mentioned above in the constructor.
- Using a transient session byte array to hold computed current offsets, instead of computing of offsets for every request.
- Using a transient byte array for uploading a ticket and copying of the ticket to the persistent memory in the end of the upload as a whole.

4.3 Arisen problems

During solving the task, several problems have appeared. Some were easy to solve, some were more challenging. Six of the most significant problems will be described in this section, along with the solution that has been used to overcome the problem.

Listing of connected flash drives

The first one was listing of connected flash drives in Java application. Java is a platform independent language and programs written in it run on Java Virtual Machine. Because of this, Java does not support many platform dependent interfaces. It is possible to list all file system roots, but it is not possible to list connected devices. Listing of file system roots could be used on Windows systems, where every connected flash drive is listed between file system roots. But this approach is unusable on Linux based systems, where there is only one file system root. All connected devices have to be mounted somewhere to the folder structure. On Linux systems, where flash drives are mounted automatically, flash drives are usually mounted to the **/media** or **/mnt** folder.

To get an accurate list of connected devices, a library written in another language, that allows to access a system API, would have to be used. However for this demonstration application, it is sufficient to retrieve an approximate list for presentation purposes only. Integrating of the library to an existing application is expected, not using of the *Service Ticket Saver* application as is.

On Windows systems, it is sufficient to list file system roots that are writeable and do not contain several strings that are usual for optical or network drives. The rules used are as follows:

- drive root **must be writable**
- path does not start with **C:**
- name does not contain **ROM** nor \\

On Linux systems, the approximate list of connected flash drives can be retrieved from a list of folders in **/media** or **/mnt** folder, that does not have **ROM** in their name and are writable.

Setting of a Service Principal Name

The next problem was configuration of SPNs using the `Setspn` program. During initial collecting of informations a tutorial about setting SPNs was found. The tutorial contained the following command for setting SPN in `Setspn` program, which supports the usage of an account name:

```
setspn –A servicename/machine ad–service–account–name
```

[22]

If the SPN was mapped to an account name, it was working in the first stage, when a demonstration application was using a keytab file to verify the ticket. However this setting was not working once, the ticket should have been used to authorized request to IIS. Even though the IIS was configured to use this account.

Solution was fortunately simple, just instead of an account name, the SPN had to be mapped to a computer account (specified by a host name of the computer) as described in the `Setspn` documentation. [23]

Null session key in retrieved tickets on Windows client

When the *Service Ticket Saver* application was first tested on a Windows client, an exception was thrown even though the application was working on the Linux machine.

Solution was fortunately again easy to find, it is a well documented security feature of new Microsoft Windows systems. To fix this problem the `allowtgtsessionkey` registry key had to be set as described in the subsection 4.1.2. After setting up, the application was working on the Windows client too.

Unlimited Strength Cryptography

Another problem came from unexpected field. When the *Service Ticket Saver* application was first tested on Windows machine with *Java SE 6 Update 21* from *Oracle* an exception `java.security.InvalidKeyException` was thrown while encrypting a storage file. Encryption was done using a cipher AES with key size 256 bits, which is supported with the AES cipher and was also working with the Java from the OpenJDK[3] package on Linux. After short research, a page[4] at IBM website was found that redirected attention to the right page at the *Sun* website. From the page, it was clear that there are some import–control restrictions that do not allow to ship Java with all cryptographic functionality. Because of this, only key size of 128 bits could be used with AES. To support unlimited strength cryptography, two JAR files *local_policy.jar* and *US_export_policy.jar* need to be downloaded from the *Oracle* website[5] and copied to $< JavaHome > /lib/security/$. However this should be

3. OpenJDK <http://openjdk.java.net/>
4. IBM - Error: java.security.InvalidKeyException: Illegal key size <http://www-01.ibm.com/support/docview.wss?uid=swg21307099>
5. Java SE Downloads <http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>

done, only if the law in the country allows to use such a strong cryptography. [24]

Addressing a resource packed in application's JAR file on Windows

Another Windows related problem appeared, when the system property for setting of the JAAS configuration file to be used (`java.security.auth.login.config`) needed to be set. Not to require any configuration of the library, a default JAAS configuration file was added to its JAR. To get a path to the file, method `getResource()` of the `Class` had been used. Every class in Java provides two methods to retrieve a resource:

getResource(String name)
: This method returns an URL of the resource, which can be used to retrieve it.

getResourceAsStream(String name)
: This method returns an InputStream that can be used to read the resource content.

Although both of these methods work fine on Windows as well as on Linux, there is a problem in using of the URL returned by the `getResource()` method on Windows systems. On Windows such an URL can not be used to address the file in the JAR file by its file path. Addressing of a file by its file path is the only way, how to set the system property `java.security.auth.login.config`. A file path retrieved from the URL does not exist on Windows, while on Linux, the file inside of the JAR file is correctly addressed.

However method `getResourceAsStream()` works perfectly even on Windows systems, so it can be used, to read the file from the JAR file. It needs to be mentioned, that the system property can be set only to a file path of the configuration file. Therefore the file read through the `getResourceAsStream()` method, has to be saved to a temporary file, to be able to use its file path to set the system property.

Time of saving/loading of a ticket to/from a smart card

When the applet was first tested with a contactless smart card, need of some performance improvements appeared. An average reading time was 520 ms and an average writing time was 498 ms for a contact smart card. For a contactless smart card, an average reading time was 1107 ms and an average writing time was 991 ms. From the average times, it can be seen that reading was more time consuming then writing for both contact and contactless cards. Such difference is little suspicious, because writing should be slower than reading. During whole reading it is communicated in both directions, but it is exchanged only few more bytes than during saving of a ticket. It is because, during every loading exchange, remaining size and ticket slot ID is always sent back along with the ticket data. On the other hand, when saving a ticket, there are two more commands sent to a card, than when loading a ticket. Tested ticket was 1065 bytes long, so using a 252 byte long ticket blocks, the ticket was transferred in five blocks. With such a small number of exchanged ticket blocks, the 3 bytes overhead in a loading command makes a total of only 15 bytes overhead. On the

other hand the overhead of two commands during saving makes $8 + 8 = 16$ bytes. So the difference is only one byte.

Table 4.1: Applet times of reading and writing of a ticket

Contact card		Contactless card	
read [ms]	write [ms]	read [ms]	write [ms]
519	503	1106	988
521	496	1109	989
516	497	1105	995
523	496	1108	991
519	499	1105	991

After several improvements, such as putting of all ticket slots to a single byte array, using of a transient session to hold already computed offsets and pre-computing of offsets bases in the constructor, the average reading time was 206 ms and the average writing time was 288 ms for the contact smart card. For the contactless smart card, an average reading time was 812 ms and the average writing time was 791 ms. Complete list of performance improvements made can be found in the end of the section 4.2. From the average times, it can be seen, that the average times have significantly improved for a contact smart card. About 300 ms (60 % improvement for a contact smart card and 27 % improvement for a contactless smart card) for reading and 200 ms (42 % improvement for a contact smart card and 20 % improvement for a contactless smart card) for writing of a ticket.

Table 4.2: Applet times of reading and writing of a ticket after few improvements

Contact card		Contactless card	
read [ms]	write [ms]	read [ms]	write [ms]
204	290	812	791
206	287	812	792
206	288	812	791
207	285	811	789
208	291	814	790

To find out whether it would be possible to improve the times even more, the times of data transfer between the computer and the smart card were measured. Times were measured using an applet, where most of the logic was removed, so that the data are only transferred between the computer and the card. Only offsets were computed on the card to provide the application at the computer correct remaining sizes. An average time of data transfer from the contact card was 173 ms and an average time of data transfer to the card was 198 ms. For the contactless card, an average time of transfer from the card was 782 ms and to the card 707 ms.

Table 4.3: Applet data transfer times from/to transient memory

Contact card		Contactless card	
read [ms]	write [ms]	read [ms]	write [ms]
173	195	780	704
174	195	784	707
175	205	784	705
173	197	783	709
170	197	779	708

From the transfer times, it can be clearly seen that the logic and persistence of data increase the time of reading only about 30–33 ms and the time of writing about 84–90 ms. It can be also seen that transfer times for the contactless smart card are about three quarters of a second, while for the contact smart card the times are under a quarter of a second.

Chapter 5

Created applications

Several applications have been created. One of the applications allows a user to save a Kerberos ticket for the requested service. The user can choose where to store the ticket, whether on a smart card, or in an encrypted file located on a flash drive selected from a list of connected flash drives. Other two applications demonstrate how can a previously stored ticket be used to make an authorized request. All three applications use the same library encapsulating the core functionality. All the applications, the library and the applet including their source codes are distributed under *original BSD license*. The text of the licence can be found in Appendix C.

5.1 Applications

5.1.1 Service Ticket Saver

The application allows a user to retrieve and save a Kerberos ticket for the requested service. The user can choose where to store the ticket, either on a smart card, or in an encrypted file located on a flash drive selected from a list of connected flash drives. The list of connected flash drives is not 100 % accurate, because Java does not have a way how to retrieve system specific information about connected devices. So in a list of flash drives, only drives complaint with the following rules are listed:

- Windows
  - drive root **must be writable**
  - path does not start with **C:**
  - name does not contain **ROM** nor \\
- Other systems (Linux, Mac OS)
  - drive is mounted in **/media** or **/mnt**, if /media directory does not exist
  - mounted directory **must be writable**
  - name does not contain **ROM** nor \\

To get an accurate list of flash drives a library written in other language would have to be used. The library would have to access system API to retrieve the list.

The application can be launched by the following command:

```
java -jar ServiceTicketSaver.jar <SPN>
```

SPN — the Service Principal Name of the service, for which a ticket should be retrieved

5.1.2 SPNEGO Client Demo Application

The application allows a user to use a previously stored ticket to make an authorized request. The request consists in retrieving of a web page located on IIS using an SPNEGO authorization token. It allows to use a ticket stored on a smart card or in an encrypted file located on a flash drive.

The application can be launched by the following commands:

```
java -jar SPNEGOClientDemo.jar <STORAGE DIRECTORY> <URL>
java -jar SPNEGOClientDemo.jar "SC" <URL>
```

STORAGE DIRECTORY	the directory, where the storage file is located, i.e. "*/media/KINGSTON*" or "*F:*" If the value is "SC" as in the second command, a ticket is retrieved from a smart card.
URL	the url of the page that should be retrieved

5.1.3 Card Terminal Demo Application

The application demonstrates such an application, that periodically checks a card terminal for presence of a card and performs an authorized service request, when the card is attached. The request again consists in retrieving of a web page located on IIS using an SPNEGO authorization token.

The application can be launched by the following command:

```
java -jar CardTerminalDemo.jar
```

5.2 Library

The library that supports saving and loading of Kerberos tickets to/from a file. The file can be encrypted using a key generated from 4-digits PIN. Library also supports saving and loading to/from a smart card using the `StorageApplet`. Also an SPNEGO token can be generated from a previously saved Kerberos ticket using the library. The SPNEGO token can be then used in a service request.

5.2.1 StorageApplet

The applet for Java CARDs allows to store several Kerberos tickets on a smart card. The limitation is given by the limited memory of the smart card. To allow easy changing of the

supported number of tickets, the maximal number of tickets is specified during installing the applet to a smart card. Due to this, the applet is installed only if a card has sufficient memory to hold the requested number of tickets. Applet has been successfully tested on a smart card NXP JCOP 41 V2.2.1.

Chapter 6

Conclusion

The main purpose of creating this thesis was to investigate and demonstrate possibilities of authentication and authorization in Java applications using the Kerberos protocol. This protocol is widely supported. A lot of organizations have a *Key Distribution Center* in their *Active Directory* even though they might not be aware of it. On the other hand, there are also open source solutions such as *ApacheDS*[1]. On clients, Kerberos is also supported on all major platforms and its support is mandatory for GSS–API implementations in Java. Such a wide support makes it a perfect choice.

At first, requirements on applications that should be created were analysed. According to the requirements, possibilities of authentication and authorization in Java were analysed. Also information about usable protocols were collected. After that, applications, the library and the applet were designed to support the required functionality.

In the end, three demonstration applications, a library and an applet for a smart card were created. One of the applications — *Service Ticket Saver* allows a user to save a ticket for a requested service on a flash drive or a smart card. The other two applications demonstrate how can be a previously saved ticket retrieved and used to make an authorized service request using *Simple and Protected GSS–API Negotiation Mechanism* (SPNEGO). The library with the applet provide the core functionality for those tree applications. Detailed description can be found in chapter 5, print screens can be then found in Appendix B.

During implementation, several problems had showed up. All the problems had been overcome and the most significant were along with their solution described in section 4.3.

The most interesting and challenging part was creating of an applet for a smart card. Even though there was possibility of implementing an applet supporting Extended APDU through `javacardx.apdu.ExtendedLength` interface. In the end, `ExtendedLength` interface was not implemented in the applet, because this interface was added in Java Card 2.2.2, which is not so widely supported by cards. Therefore a solution that can be used with lower versions of Java Card specification was selected.

The possible usage of the created library is in any set of applications that should allow a user to be authenticated on one computer and use the ticket on another machine or device to get access to it. For example to get access to a printer. Therefore the library is the main outcome of this work and the other applications just demonstrate its usage to do the

1. ApacheDS v1.5 — An extensible, embeddable LDAP and Kerberos server entirely in Java <http://directory.apache.org/apacheds/1.5/>

required operations. Such a solution could be used by companies which already have a *Key Distribution Center* in their infrastructure, to secure their printers or other devices from being misused. Because no special infrastructure would have to be created, such a solution would be cheap.

In the future, the library could be extended and used not only on devices, but also on client stations to authorize user requests to some intranet applications. Also support for other authentication protocols could be added in the future, because the library internally uses standard *Java Authentication and Authorization Service* (JAAS) and *Generic Security Service Application Program Interface* (GSS–API), which are protocol transparent. To support some other protocols, the storage package would have to be modified to support storing of some other types of credentials. The library internally uses JAAS and GSS–API to retrieve and use the previously stored credentials (Kerberos tickets in current state), so the internal logic could stay intact. Only its interface and credentials related actions would just needed to be generalized to accept other types of credentials, than just instances of `KerberosTicket`. Also proper OID for the used type of credentials would have to be passed to GSS–API. This could be easily achieved by introducing of a mapping between credentials types and OIDs.

Bibliography

[1] Oracle and/or its affiliates: *Java™ Authentication and Authorization Service (JAAS) Reference Guide*,
May 22, 2011
<http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html> 2.2.1

[2] Oracle and/or its affiliates: *Single Sign–on Using Kerberos in Java*,
May 22, 2011
<http://download.oracle.com/javase/1.5.0/docs/guide/security/jgss/single-signon.html> 2.2.2

[3] Oracle and/or its affiliates: *Advanced Security Programming in Java™ SE Authentication, Secure Communication and Single Sign-On*,
May 22, 2011
<http://download.oracle.com/javase/6/docs/technotes/guides/security/jgss/lab/> 2.2.2

[4] John Linn: *Generic Security Service Application Program Interface Version 2, Update 1*,
May 22, 2011
<http://www.ietf.org/rfc/rfc2743.txt> 2.2.2

[5] Oracle and/or its affiliates: *Java™ Cryptography Architecture*,
May 22, 2011
<http://download.oracle.com/javase/1.5.0/docs/guide/security/CryptoSpec.html> 2.2.2

[6] Fulvio Ricciardi: *KERBEROS PROTOCOL TUTORIAL*,
May 22, 2011
<http://www.kerberos.org/software/tutorial.html> 2.3.1

[7] Clifford Neuman, Tom Yu, Sam Hartman, Kenneth Raeburn: *The Kerberos Network Authentication Service (V5)*,
May 22, 2011
<http://www.ietf.org/rfc/rfc4120.txt> 2.3.1

[8] Larry Zhu, Paul Leach, Karthik Jaganathan, Wyllys Ingersoll: *The Simple and Protected Generic Security Service Application Program Interface (GSS–API) Negotiation Mechanism*,
May 22, 2011
<http://www.ietf.org/rfc/rfc4178.txt> 2.3.2

[9] Oracle and/or its affiliates: *Java Card Technology Overview*,
May 22, 2011
<http://www.oracle.com/technetwork/java/javacard/overview/overview-jsp-135353.html> 2.4

[10] Oracle and/or its affiliates: *Applet Firewall and Object Sharing*,
May 22, 2011
<http://java.sun.com/developer/Books/consumerproducts/javacard/ch09.pdf> 2.4

[11] IEC: *ISO/IEC 7816-4*,
May 22, 2011
<http://webstore.iec.ch/preview/info_isoiec7816-4%7Bed2.0%7Den.pdf> 2.1

[12] ANSI: *International Registered Application Provider Identifier (RID)*,
May 22, 2011
<http://www.ansi.org/other_services/registration_programs/rid.aspx?menuid=10> 2.4

[13] IEC: *ISO/IEC 7816-5*,
May 22, 2011
<http://webstore.iec.ch/preview/info_isoiec7816-5%7Bed2.0%7Den.pdf> 2.4

[14] UNMZ: *Česká národní registrační autorita pro RID k přidělení registrovaného identifikátoru poskytovatele aplikace (RID) podle ISO/IEC 7816-5*,
May 22, 2011
<http://www.unmz.cz/urad/ceska-narodni-registracni-autorita-pro-rid-k-prideleni-registrovaneho-identifikatoru-poskytovatele-aplikace-rid-podle-iso-iec-7816-5> 2.4

[15] Sun Microsystems, Inc.: *CHAPTER 5 — Using Extended APDU*,
May 22, 2011
<http://www.cs.ru.nl/~tews/jcdocs/app-notes-2.2.2/extapdu.html> 2.4

[16] Zhiqun Chen: *How to write a Java Card applet: A developer's guide*, May 22, 2011 <http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html> 2.4

[17] Oracle and/or its affiliates: *Discover the secrets of the Java Serialization API*, May 22, 2011 <http://java.sun.com/developer/technicalArticles/Programming/serialization/> 3.1

[18] W3C® (MIT, ERCIM, Keio): *Extensible Markup Language (XML)*, May 22, 2011 <http://www.w3.org/XML/> 3.1

[19] Wikipedia®: *Abstract Syntax Notation One*, May 22, 2011 <http://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One> 3.1

[20] The Apache Software Foundation: *Apache Directory Server v1.0 – Kerberos ASN.1 codec*, May 22, 2011 <https://cwiki.apache.org/DIRxSRVx10/kerberos-asn1-codec.html> 3.1

[21] Oracle and/or its affiliates: *Troubleshooting*, May 22, 2011 <http://download.oracle.com/javase/1.5.0/docs/guide/security/jgss/tutorials/Troubleshooting.html> 4.1.2

[22] Blogger The Java Monkey: *Active Directory and Kerberos Service Principal Names*, May 22, 2011 <http://thejavamonkey.blogspot.com/2008/03/active-directory-and-kerberos-service.html> 4.3

[23] Microsoft: *Setspn Overview: Active Directory*, May 22, 2011 <http://technet.microsoft.com/en-us/library/cc773257(WS.10).aspx> 4.3

[24] Oracle and/or its affiliates: *Using AES with Java Technology*, May 22, 2011 <http://java.sun.com/developer/technicalArticles/Security/AES/AES_v1.html> 4.3

Appendix A

Browsers with SPNEGO support

There are browsers supporting SPNEGO for all major platforms (Windows, Mac OS and Linux). Especially Mozilla Firefox, Google Chrome and Chromium are supported on all platforms. There is a list of platforms with browsers available for them which are supporting SPNEGO:

- Windows
  - Internet Explorer
  - Mozilla Firefox
  - Google Chrome/Chromium
- Linux
  - Konqueror
  - Mozilla Firefox
  - Google Chrome/Chromium
- Mac OS[1]
  - Safari
  - Mozilla Firefox
  - Google Chrome/Chromium

Internet Explorer

Support has been added in version **5.0** and it is supported only on Windows. To allow SPNEGO, following steps have to be performed:

- in the **Security settings** check **Enable Integrated Windows Authentication (requires restart)**
- add all requested domains to **Sites** list in **Local intranet**

Detail instructions can be found on Microsoft's website: `http://msdn.microsoft.com/en-us/library/ms995329.aspx`.

1. List is based on the available information and support has not been tested on Mac OS due to hardware restrictions of the system.

Mozilla Firefox

Support has been added in version **0.9** and it is supported on all platforms. To allow SPNEGO, all domains has to be added to **network.negotiate-auth.trusted-uris** configuration property, which can be found on Firefox configuration page about:config. Detail description of integrated authentication support can be found at: https://developer.mozilla.org/En/Integrated_Authentication.

Google Chrome/Chromium

These two browsers are based on same community source codes for Chromium. Support has been added in version **6.0.472** and it is supported on all platforms. To allow SPNEGO, browser has to be run with following arguments:

- --auth-server-whitelist="*comma separated list of domains*"
- --auth-negotiate-delegate-whitelist="*comma separated list of domains*"

Detail description of authentication support can be found at: http://dev.chromium.org/developers/design-documents/http-authentication.

Konqueror

Support has been added in version **3.3.1** and no special configuration is needed.

Appendix B

Print screens of created applications

Service Ticket Saver

Flash drive tab

This tab is always available and allows to store a ticket to a flash drive.

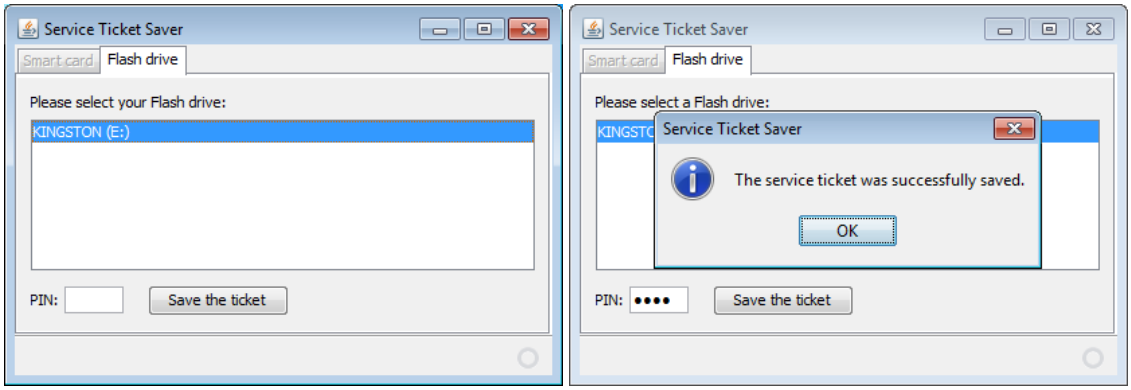

Figure B.1: Flash drive tab with selection of a flash drive and a confirmation dialog

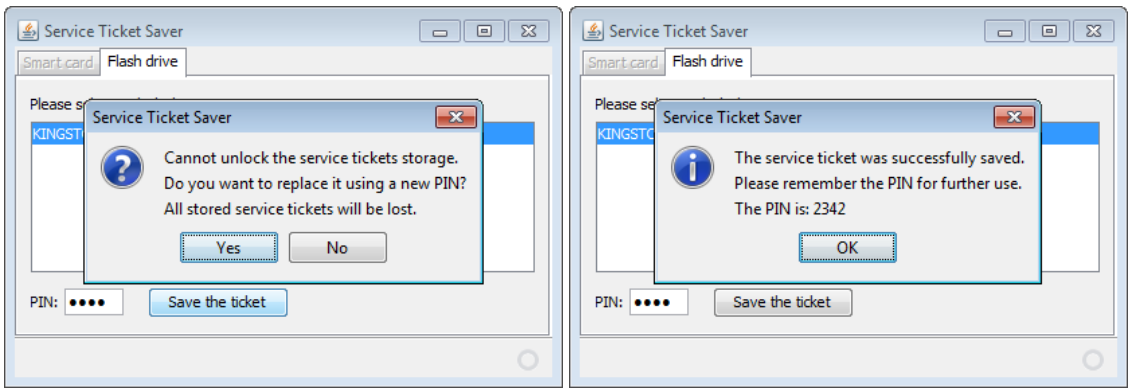

Figure B.2: Flash drive tab with a wrong PIN dialog and a new PIN dialog

Smart card tab

This tab is available only if a smart card reader is attached and allows to store a ticket to a smart card.

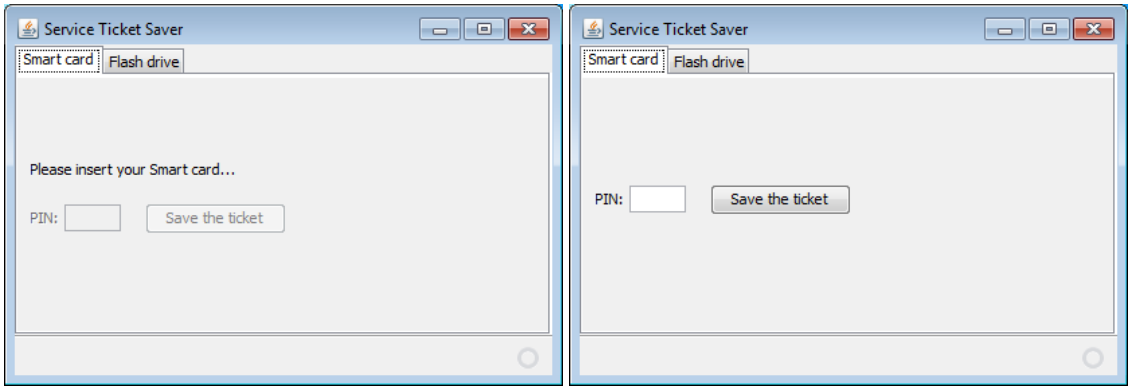

Figure B.3: Smart card tab with a prompt for a smart card and after a card is inserted

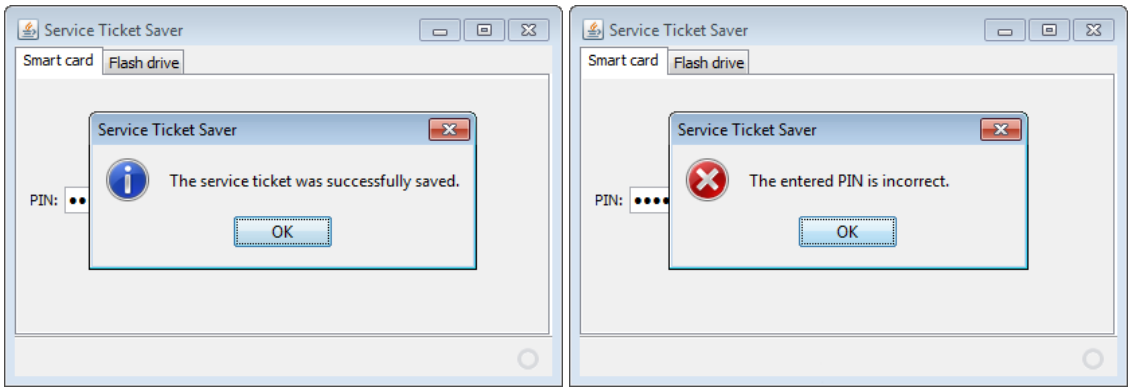

Figure B.4: Dialogs when a ticket is successfully saved and when an incorrect PIN was entered

SPNEGO Client Demo

```
[hopkins@delisek SPNEGO Client Demo]$ java -jar SPNEGOClientDemo.jar "SC" "http:
//ws2008.domain.localhost"
Please enter yor PIN:
Response code: 200 (OK)
Content:
<html>
<head>
<title>IIS Test</title>
</head>
<body><h1>Test example</h1></body>
</html>
```

Figure B.5: Successful loading of a page from IIS

```
[hopkins@delisek SPNEGO Client Demo]$ java -jar SPNEGOClientDemo.jar "SC" "http:
//ws2008.domain.localhost"
Please enter yor PIN:
```

Figure B.6: PIN prompt, if a card requires PIN verification

```
[hopkins@delisek SPNEGO Client Demo]$ java -jar SPNEGOClientDemo.jar "SC" "http:
//ws2008.domain.localhost"
Please enter yor PIN:
Cannot generate an SPNEGO token.
No valid ticket exists.
```

Figure B.7: Warning when no valid ticket was found on the card

```
[hopkins@delisek SPNEGO Client Demo]$ java -jar SPNEGOClientDemo.jar "SC" "http:
//ws2008.domain.localhost"
Cannot read the service tickets storage.
A smart card with the applet is not attached.
```

Figure B.8: Warning when a card is not attached

Card Terminal Demo

```
[hopkins@delisek Card Terminal Demo]$ java -jar CardTerminalDemo.jar
```

Figure B.9: Awaiting a card

```
[hopkins@delisek Card Terminal Demo]$ java -jar CardTerminalDemo.jar
Please enter yor PIN:
Cannot communicate with the smart card storage.
```

Figure B.10: Warning when a card communication error occurred

```
[hopkins@delisek Card Terminal Demo]$ java -jar CardTerminalDemo.jar
Please enter yor PIN:
Cannot generate an SPNEGO token.
No valid ticket exists.
```

Figure B.11: Warning when no valid ticket was found on the card

```
[hopkins@delisek Card Terminal Demo]$ java -jar CardTerminalDemo.jar
Please enter yor PIN:
Response code: 200 (OK)
Content:
<html>
<head>
<title>IIS Test</title>
</head>
<body><h1>Test example</h1></body>
</html>
```

Figure B.12: Successful loading of a page from IIS

Appendix C

Licence

Appendix D

Contents of the attached CD

A CD is attached to the thesis with the following contents:

- source codes of the applications and the library (Maven projects)
- source codes of the applet (NetBeans Ant project)
- documentation of all source codes in *JavaDoc* format
- built applications with attached default configuration files
- this thesis in PDF format