Credential Mapping in Grids

ESTEBAN TALAVERA

Master of Science Thesis
Stockholm, Sweden 2007

ICT/ECS-2007-33

ROYAL INSTITUTE
OF TECHNOLOGY

Credential Mapping in Grids

Master of Science Thesis

ESTEBAN TALAVERA GONZÁLEZ

Supervisor: Mehran Ahsant (PDC/CSC/KTH)
Examiner: Assoc. Prof. Vladimir Vlassov (ECS/ICT/KTH)
Stockholm, Sweden – March 19, 2007

Center for Parallel Computers (PDC)
Royal Institute of Technology (KTH)
SE-100 44 Stockholm
SWEDEN

Abstract

Grids provide a way of utilizing a vast array of linked resources by sharing computing powers, databases, and services on-line across Virtual Organizations (VOs) that are dynamic and geographically distributed. In Grids, organizations participating in VOs might use different security mechanisms, and they might deploy different security infrastructures. Thus, the Grid security mechanism needs to interoperate with these mechanisms rather than replacing them.

One challenging issue is how to prove the identity and authority of service requesters and service providers when operations between these entities cross realms and VOs. Security credentials proving the identity and the authority of the holder may not be syntactically or semantically recognized inside remote domains. Furthermore, the dynamic and distributed nature of VOs in most cases makes it difficult to pre-establish trust relationship among sites before executing Grid applications. Therefore the security credentials may not be trusted inside other domains. In fact the credential translation mostly deals with issuing the same assertion in equivalent form of identity and trust.

The purpose of this project is defining a way of translating security credentials and developing appropriate software components for addressing trust and identity issues when converting these credentials coming from and to be used on different and heterogeneous security realms in Grids.

Acknowledgments

First of all, I want to thank the person who guided me during the whole project, from its very beginning to the end: Mehran Ahsant, my supervisor at PDC. He made me interested in the topic, and helped me a lot with theoretical issues, as well as with problems that arose during the implementation. He gave me important suggestions when writing this report, too.

I am very grateful to my examiner at KTH, Dr. Vladimir Vlassov, who gave me good advices when starting the project and really useful feedback about my report.

Special thanks to my parents and my sister, who always took care of me during my stay in Sweden. Without their help these 20 months abroad would have been impossible. They have always supported me on all situations.

Thanks to all people who lived through the "Erasmus experience" with me, those at the beginning, those at the end, and specially those who were there the whole period. I will never forget the dinners, trips, parties... that we enjoyed together. I have made really good friends in Stockholm.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter provides a general introduction to the area the Master's Thesis is located in. It is a summary of the whole report. It starts with a short overview of the concept of Grids and Grid security problems related to our work. Then, we give a detailed problem statement that motivates our work. Following this we note the work already done on this project when this thesis got started, subsequently examining the main goals of the thesis and our approach to achieve them. A description of the limitations of our contribution is given, concluding with the summary of the content of the rest of the report.

1.1 Overview

In a **Grid** environment, clients and resources from different security realms interact with each other. When users submit jobs to the Grid, they might make use of many different remote resources, perhaps crossing many realms while executing. Each time these applications want to use a resource (e.g., they need to be executed for some time in a certain processor, or read from a certain database), they have to be first *authenticated* and then *authorized* by the access control mechanism of the target resource. As different security domains usually implement different authentication protocols, the credentials that clients use to be authenticated at certain local or remote domains could be invalid. Moreover, the resources that the jobs need to use are usually not known beforehand, since it depends on the execution path the applications take given the input data and intermediate results they generate. Then, it is very difficult for the users to "prepare" all credentials they will need before launching their jobs. If during the execution a job intends to use one resource but it does not have the needed credential (because the target domain uses a different protocol, or even both parties use the same protocol but the credential is not trusted by the resource), the job could be stopped before it finishes the desired tasks.

To prevent the above situation for occurring, we need a way of *translating* credentials that clients already have into another ones valid for the resources inside the Grid that they want to use.

1.2 Problem Statement

As exposed, in a Grid environment members participating in VOs, services, and end-users need to interact with each other. The Grid service requests may cross organizational boundaries where different security models are used. This includes the authentication and authorization procedures the services and their requesters of the administrative domains composing the Grid must perform in order to successfully carry out a job. The administrator of every domain chooses the security credential (or credentials) that will be valid to prove other parties' identity depending on the security mechanisms implemented in that realm and the administrator preferences for every resource. The main problems for the holder of a security token trying to be authenticated are:

Security credentials in **different formats**: An organization may receive a security credential from a remote domain that uses a different security mechanism. For example it might happen for a recipient to receive a Kerberos ticket when it doesn't support Kerberos functionality. Thus one challenging issue is how to convert the format of security credential from one standard or technology into another. For example converting a security token in form of Kerberos ticket into an X.509 certificate, or vice versa.

Trust issues: Each request asking to perform an operation has to provide a collection of claims to prove the accuracy and authenticity of the request. In order to verify such a request, any individual service provider must rely on pre-established trust between requester and hosting domain. This trust establishment helps participants to validate the security credentials presented by the requesters, and the mechanisms involved are reasonably well understood and developed. Therefore due to the lack of direct or brokered trust relationships between the requester and the service provider domains the security credentials may not be trusted. The implications that above mentioned credential conversions may have in the trust establishment process should be investigated.

Finding a way of managing these two points guarantees Grid organizations to receive a security credential which is understandable and may be trusted. Otherwise, if authentication cannot be performed at any point of the application execution, the access to a needed resource might be prevented by the resource's access control mechanism. This could lead to the termination or the stoppage of that application before performing the job it was intended to do.

1.3 Previous work

When this thesis started, PDC had already implemented a Web service able to translate Kerberos credentials into X.509 certificates. Our contribution from that

point has been *adding new functionalities* to that "basic" system. As we will see during this paper, we have added Kerberos signature for the communication translation service→client, and 2 more translations: Kerberos to SAML and X.509 to Kerberos. We have also investigated the interoperability of our system with a new technology that is growing in importance, Shibboleth.

The functionalities that were already developed were **tested in a real Grid environment** before I continued adding new features.

1.4 Goals

To address the exposed difficulties, an effort needs to be made for translating security credentials from a format comprehensible in the requester domain into an understandable format in the relying domain. This conversion should be applicable for diverse formats of security credential supported by different security realms. The approach needs also to cover the trustworthiness of translated security credentials as described in Section 1.2.

This translation has to be made without any lack of security, or at least minimizing the risks to the minimum. When converting a security credential into a new one valid for the target resource, the client should not be able to perform actions that she is not allowed to carry out according to her identity and the security policy. The translation will just map the old security token to an equivalent one understandable and trustable by the party that makes the authentication procedure, without adding capabilities the client does not have.

1.5 Approach

The available standards, specifications, and tools will be investigated, choosing the most suitable ones in terms of their *importance*, *scalability*, *interoperability*, and *level of security* offered. In order to make the solution usable, it will try to follow the most widely used standards in this field. The security credentials that will be taken into account for mapping are **Kerberos tickets**, **X.509 certificates**, and **SAML assertions**.

After choosing the technologies and designing the system's architecture, a prototype will be implemented. As a result, a **translation** service will be running, waiting for credential conversion requests coming from clients. The service will get the request, validate it, and map the received credential into another specified by the client which is valid on the target resource. This new credential is sent to the client, to be used for being authenticated by the resource.

This translation service will be tested to show that the designed model is valid. Different tests covering the implemented conversions between different credentials would be performed, showing how the results are valid in terms of format and trust. These tests would consist in several client applications requesting a service with different input parameters, showing then the obtained results. Having the

time constraints, the evaluation of the service will be focused on the validity of the results rather than on measuring its performance.

1.6 Limitations

The problem statement described in Section 1.2 has many issues (dealing with security or not) that need to be addressed in order to construct a system fully functional in a real Grid environment. Solve all of those issues when designing and implementing the system is too much to cover for this project, given the manpower and time budget constraints at hand. These unsolved problems could be addressed in the future through adding new functionalities to the prototype developed on this project:

- **Attributes/Name-space mapping:** Security credentials might be semantically different and incomprehensible for the recipients. Security credentials' attributes might not be able to be mapped directly from an issuing domain to a relying domain. In that case, the attributes associated with the security credential must also be transformed into an understandable format and semantics. Authorization servers need to understand the formats in order to use authorization information such as groups and roles associated to an identity. For instance how to translate a role attribute ("administrator") to a set of implicit or explicit privileges in another domain ("userid=root", "write access to database", and so on).
- **Attribute issuance:** Presenting a token with the requester's attributes to the resource often makes more sense that presenting a credential stating her identity. This is even more important for large and dynamic environments such as Grids: As new subjects are continuously being added to and removed from the system, it is usually easier to provide them with the corresponding attributes understandable by resources they can access to ("student", "registered in the course 2G1004", etc.), rather than giving them credentials only including their identity (e.g. "Esteban Talavera", that may be understandable by the target resource or not). A desired quality of the translator could be the issuance of attributes based on the client's identity (contained in the credential to be translated) and the translator's policy. For example, providing a credential including "Esteban Talavera" as identity information the client could get another token stating that he is "registered in the course 2G1004".
- **Dynamic trust relationships:** As we will see during this report, certain trust properties between *client* and *token issuer* and between *token issuer* and *target resource* must exist beforehand. The establishment of those trust relationships are out of the scope of this project. We will always suppose that trust relationships are established before the credential translation takes place.

1.7 Thesis Outline

The rest of the report is organized as follows:

`Chapter 2, Background:` It gives a briefly explanation of the Grid technology. Then, this chapter explains the important protocols and specifications needed to understand our project.

`Chapter 3, Credential Mapping:` It provides the extension of our Approach, introduced in Section 1.5. The knowledge acquired from the literature studied is applied to meet our goals. The architecture and behaviour of the resulted system is explained for each situation.

`Chapter 4, Related Work:` It describes some other important projects related to credential translation, pointing out similarities and differences and possible interoperability with our system.

`Chapter 5, Prototype Implementation:` First, it specifies the functionalities implemented. Then, it introduces the environment used while implementing the designed system, summarizes the code structure, and explains the main problems found.

`Chapter 6, Analysis of Results:` It describes which tests were performed to check the validity of our prototype, underlining the conclusions we can extract from the obtained results.

`Chapter 7, Conclusions:` It contains the summary of the contributions that we have made, and suggestions for the next steps of the project.

Some `Appendices` include abbreviations and definitions of concepts used on this report, and the XML messages exchanged by the parties during the tests.

Chapter 2

Background

This chapter covers the "theory" the reader needs to be familiar with to understand our work: what is Grids, Grid security issues, and which standards and protocols have already been proposed in the area of our work.

2.1 Grids

A *Grid* (aka. *Grid Computing*) has today many different definitions. It could be defined as *"the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across multiple administrative domains"* [2]. A Grid is composed of a set of *Virtual Organizations (VOs)*, which may be dynamically and geographically distributed and are interconnected by a network (such as the Internet). Those VOs are groups of resources (computing power, data storage, specific applications, etc.) that may or may not be located in the same administrative domain. VOs may even be overlapped.

Grids try to take advantage of this distributed scenario and large amount of resources, providing services to different clients. Grids can abstract the client from the complexity of its infrastructure, providing the same service as supercomputers or computer clusters. This is performed balancing the amount of work between different VOs. Scalability, performance and heterogeneity are desired characteristics of a Grid. Three different types of Grids can be distinguished [2]: *Computational Grids* (mainly used for applications demanding processing power), *Data Grids* (focused on the management of large amounts of information), and *Equipment Grids*, where the Grid is used to control some kind of equipment (e.g. a telescope).

In order to give Grid environments the external appearance of a single powerful platform, where client's applications that are executing there does not need to care about where the resources are or their peculiarities, some open standards and middlewares supporting large scale data and computation have been developed.

These middlewares hide the heterogeneity of Grids, giving standardized interfaces to applications regardless the specific resource they are using. The *Open Grid Forum*[1] is the largest community behind the standardization of Grids, being the *Globus Toolkit*[2] a widely used open source framework for building Grids.

2.1.1 Grid Security

As explained in Section 2.1, heterogeneity is a key characteristic of Grids: Each administrative domain and resource may be based in different platforms and architectures and can use different methods (e.g., for authentication and authorization). This includes security mechanisms, resulting in many security problems specific of Grids environments. Security maintenance when every realm could have implemented a varied set of security mechanisms, policies, individuals, group memberships, etc., is not a trivial issue. Another key quality of a Grid, its dynamic nature, makes it difficult to establish previous trust relationships, and policies and security mechanisms used in organizations and resources that compose the Grid.

In order to allocate a job in a specific resource, the security processes (authentication, authorization, etc.) have not only to be carried out at a global level (the Grid), but also satisfy the local requirements stated in the local policies. The access control is performed locally, at a resource level. Since it is usually not possible to know all the resources the client's job will need during execution time, it is also not possible to generate the security information that will be needed in advance. In the worst case, the security mechanisms in two VOs that have to communicate at some step of the execution could be incompatible in some way. Thus, these cases have to be managed to perform the job.

The Grid framework has to ensure that every client uses only the resources she is entitled to access to, and with the needed restrictions. For example, the job from a client could be authorized to execute up to a limit of time in the processor `P1`, and read information from the database `DB1`, but writing on `DB1`, accessing in any manner to another database `DB2` and executing in the processor `P2` could be forbidden. This access will be given by the authentication and authorization procedures of the user in the target resource's domain. The problem is that, given the heterogeneity and dynamic nature of the Grid, the identity of the client might not be known in the resource's local domain, or the credential presented by the client for authentication might not be valid or trusted there.

2.2 Kerberos

Kerberos is a network authentication protocol, which uses symmetric cryptography (the encryption is performed with a shared key that only the parties involved in

[1] http://www.ogf.org/
[2] http://www.globus.org/toolkit/

the conversation know) for authenticating clients and servers. It was developed at MIT, becoming the latest version *Kerberos V5* an Internet Standard Track in 1993, revised later in 2005 [3]. It is a widely used standard in open (unprotected) networks, designed to provide single sign-on for these networks. This means that users do not need to authenticate themselves against every service they want to use, but once per session (e.g. through a login at the beginning composed by her username and password).

The actors of a Kerberos realm (an authentication administrative domain) are **clients** that want to access **services** provided by one or more **application servers** (hosts). These operations are supervised by the **Authentication Server (AS)** and one **Ticket Granting Server (TGS)**. Secret keys are shared between the AS and the TGS, the AS and the clients, and between the TGS and every server in the realm (in the latter case they are called *master keys*). The pair AS–TGS is known as **Key Distribution Center (KDC)**, and they may be executing in the same machine or not.

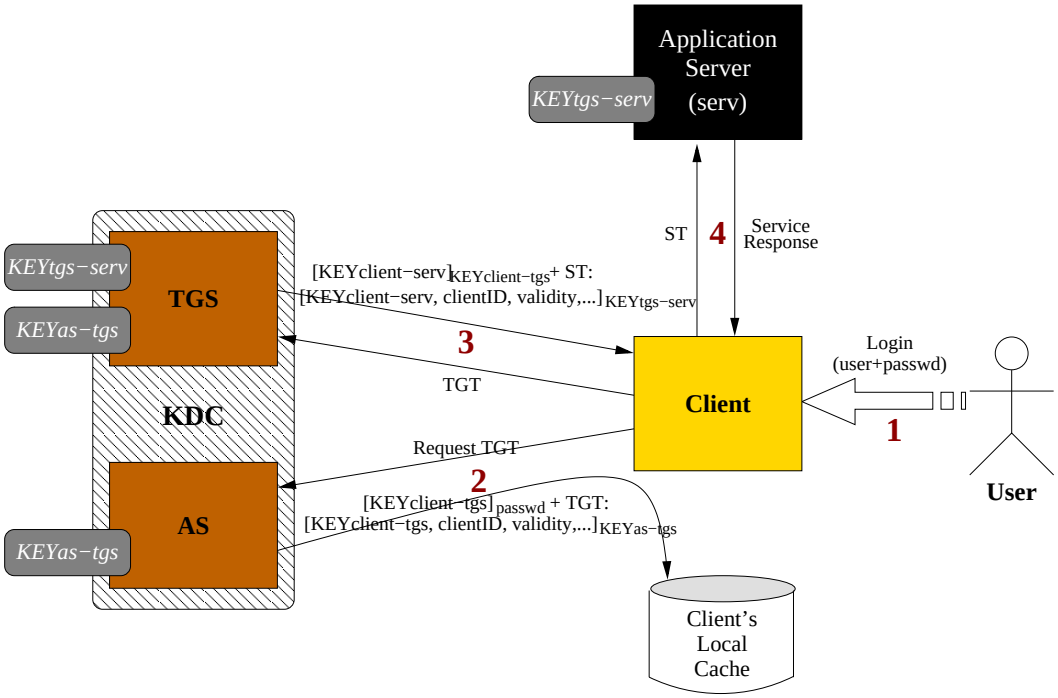

Figure 2.1. Basic Kerberos operation

As illustrated in Figure 2.1, a basic Kerberos operation has 4 steps:

1) The user inserts the Kerberos user name and password in the local computer. This is not done again until the credentials expire.

2) The client requests a credential to the AS. After authenticating the client, the AS sends to the client the secret key for communicating with the TGS and the *Ticket Granting Ticket (TGT)* for the current session. The TGT is encrypted with the key shared by the AS and the TGS, and contains the secret key and the client's identity.

3) The client requests credentials for a specific service she wants to use to the TGS, presenting the TGT for authentication. The communication is protected with the secret key that the client already has and the TGS can obtain from the ticket. If the client is authorized to use that resource, the TGS answers with a valid credential valid for the desired service, the *Service Ticket (ST)*. It is encrypted with the key shared by the TGS and the application server, and includes a key (the *session key*) to be used for communicating with the service, and the client's identity. That key is also given to the client. Simplifying, we could say that the Ticket Granting Ticket is a special kind of Service Ticket, being the issuer the Authentication Server instead of the Ticket Granting Server, and the application server the client is intended to access to the TGS.

4) The client is now able to request the desired service to the application server. It presents the ST recently obtained, and authentication information encrypted with the session key received from the TGS. The application server will obtain that session key from the ticket, and after validating the information will decide whether it should serve the request or not, depending on the application server policy.

The KDC's database will store the name of all clients, services and hosts (the so called **principals**). The principal for a user is usually following *name@REALM* (e.g. "esteban@NADA.KTH.SE"), while for services is of the form *service/hostname@REALM* (e.g. "imap/mailbox.kth.se@NADA.KTH.SE") [4]. The TGS of a specific realm has the principal *krbtgt/REAML@REALM* [3].

According to the standard [3, Section 5.3] a Kerberos ticket is composed of the `Realm` that issued the ticket and the `PrincipalName` of the target service (either an application server or the TGS), and a part called `EncTicketPart` that is encrypted with the key shared by the issuer and the entity that will verify the ticket. The main information of this part is the *session key* that will be used in the communication between the parties, the *identity* of the holder (`Realm` and `PrincipalName`), the *authentication time* (when the issuer authenticated the holder), and the *validity* of the ticket (the start and end times that tell the verifier when the ticket is valid). It is usually not a good idea to issue tickets with very long expiration time, since it decreases the level of security (e.g., if an attacker is able to get the session key of the communication, she could listen to it for a longer time). Typical durations of TGTs are 12 or 24 hours from the start time.

2.2.1 Cross-Realm Authentication

The scenario shown in Figure 2.1 is only valid when the client, the KDC, and the application server belong to the same Kerberos realm. A user can be authenticated and obtain certain services in her local Kerberos domain, where her user name and password are registered (the user is a *principal* in that realm), but not outside. The *Cross-Realm Operation* [4] is a mechanism that allows a Kerberos principal to be authenticated in remote Kerberos realms. Then, a user could be able to request a kerberized service from a remote domain.

If two Kerberos realms `A` and `B` want to "export" local services to users of the remote realm, they will need to **trust** each other. This trust is given by shared secrets between the two KDCs. The KDCs will share two secret keys: One will be used by principals in `A` trying to access to resources in `B`, and the other for the opposite direction. The principals in one realm will be able to get a *remote TGT* valid in the remote KDC: The principals "krbtgt/A@B" and "krbtgt/B@A" will be registered in both KDCs, representing the *remote Ticket Granting Servers*. The shared keys will be associated with these two new principals.

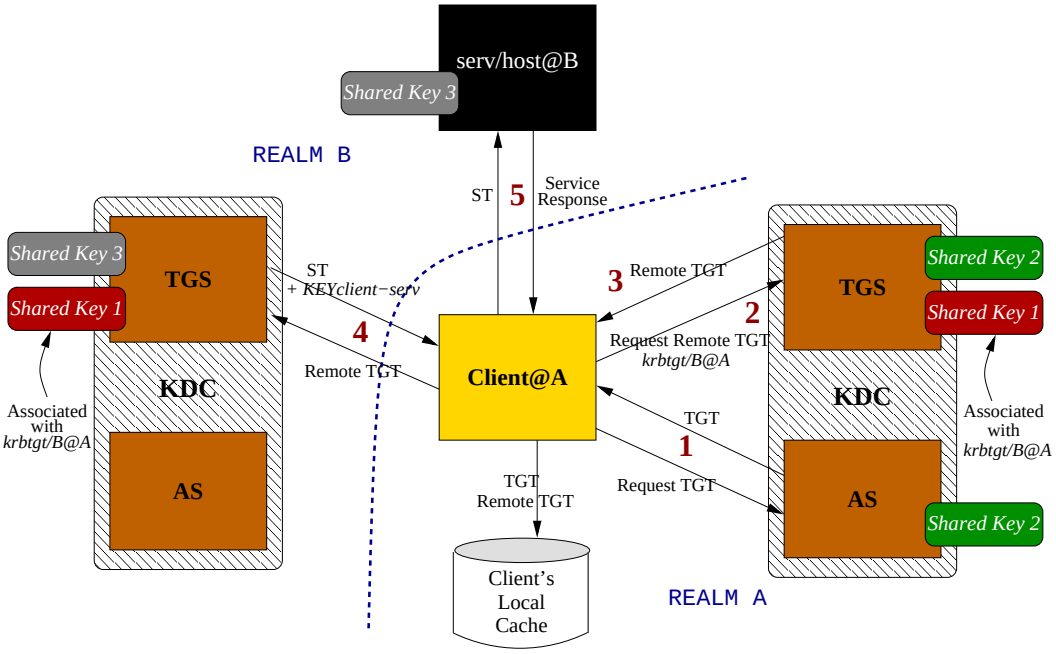

Figure 2.2. Basic Kerberos Cross-Realm operation

When a user from `A` wants to get a service of `B`, the operation illustrated in Figure 2.2 will take place[3]:

[3]Note that this diagram shows only one remote TGS principal and its corresponding key. For users in `B` to be able to access services in `A`, another secret key associated with the principal

1) The client gets a TGT from the KDC `A`.

2) As the client realizes that the service she is intended to use is located in a remote realm, the TGT is used to request a *remote TGT* valid for that realm. Thus, the principal specified by the client will be `krbtgt/B@A` (the name of the remote TGT).

3) The local KDC now creates the remote TGT, encrypted with its associated key (the one that is shared by `A` and `B`).

4) With the received TGT, the client asks the KDC in realm `B` for a service ticket valid for the remote resource. The KDC in `B` is able to decrypt the TGT with the *Shared Key 1* and validate the information. If valid it issues the requested ST.

5) The client requests the service presenting the ST.

To reduce the number of shared keys when a Kerberos realm has a trust relationship with many others, Kerberos v5 introduced the *transitivity* of trust: If realm `A` trusts `B` (they share two keys), and `B` trusts realm `C`, then automatically `A` trusts `C`.

2.3 Public Key Infrastructure

The *public key encryption* is based on *asymmetric key cryptography* (see Appendix A). This means that there are a pair of keys, one private (that only the owner possesses) and one public (accessible for everybody): the data encrypted by one key can only be decrypted by the other one and vice versa. This technology is maintained by the principle that it is computationally infeasible (with the computer power and algorithms currently available) to derive the pair key of a given one [5]. The public key mechanism provides confidentiality (the data encrypted with the public key can only be decrypted by the holder of the private key) and integrity and authentication through *digital signatures* (only the owner of the key pair is able to encrypt information with the private key, and it can be verified by the recipient with the public key).

The Public Key Infrastructure (PKI) is *"a system of CAs [Certification Authorities] [...] that perform some set of certificate management, archive management, key management, and token management functions for a community of users in an application of asymmetric cryptography"* [5]. A *public key certificate* is a data object that binds the user's identity with her public key [5]. In PKI, this binding is done through a *Trusted Third Party* (the CA), which signs the certificate.

krbtgt/A@B must be shared between the KDCs.

2.3.1 X.509 certificates

The most used variant of the Public Key Infrastructure is the X.509-based PKI. It uses **X.509 certificates** as credentials for publishing the public keys. This model works as follows:

1) When a user wants to create a certificate, it first creates a pair of keys. The private key will not leave the user's machine, but the public key is sent along with the user's identity in a *Certification Request (CR)* to a *Certification Authority (CA)*. A CA is a trusted party, which also holds its own pair of keys and certificate (possibly generated by a higher level CA).

2) Given the certification request, the CA adds to its data a validity period, a serial number, and the CA's identity. This information is signed with the CA's private key. This includes computing the hash value of the data included in the certificate so far and then to encrypt it with that key. The digital signature is appended at the end of the certificate, and therefore it links the user's identity with the public key she generated. The result is the certificate, which is sent to the user.

3) Now the user can use the certificate as a credential together with its private key to prove its identity. For example, she could send the certificate and a value encrypted with the private key to another entity. If the receiver trusts the CA that signed the certificate (or the signer of the CA's certificate and so on), then that entity will match the certificate's identity and that public key, being able to verify any value signed with the private key.

The latest revision of the X.509 certificate format is *version 3* [6]. The most important fields of an `X.509 v3` certificate [6, Section 4] for us are the `serialNumber`, the `publicKey`, the `validity` (stating that the certificate is not valid before and after two given times), and the `issuer` and `subject` fields indicating the CA's identity and certificate holder's identity respectively. These two fields must be *Distinguished Names (DN)*. This standardized name consists of several fields that give information about the subject they are referred to, like country (C), organization (O), organization unit (OU), state or province name (ST), locality (L), common name (CN), etc. An example of my DN, if I am working at KTH/PDC, could be: "O=KTH, OU=PDC, L=Stockholm, ST=Stockholm, C=SE, CN=Esteban"

2.4 Securing XML

One way of exchanging XML messages in a secure manner is the application of XML Signature and XML Encryption to provide both integrity and confidentiality in the communication.

2.4.1 XML Signature

Digital signatures within XML messages [7] provide message integrity (has the message been changed since it was computed in the origin?), origin authentication (who sent the message?) and support for non-repudiation (i.e., the sender cannot deny that she was the origin of the message) for the objects the signature it is applied to. The standard [8] defines the `Signature` element, with the signature's information. It will have a child element `SignedInfo` which will contain the details of the signature process performed to obtain the signature value:

`CanonicalizationMethod:` As XML files with the same "meaning" could have different digests values (because of blank spaces, end of lines, and comments for instance), the file must be canonicalized before hashing it. The canonicalization process ensures that any XML message part with the same information (same elements containing the same data) will have the same binary representation, and therefore the digest result will be the same for all of them.

`SignatureMethod`. It tells us about the method used to compute the signature, such as "RSAwithSHA1" which means that the hash function used was SHA-1 and the signature was computed with the RSA algorithm.

`Reference` (there should be at least one in a `SignatureInfo` element). It includes the reference to the data that is being signed (i.e., which elements of the XML message or which data objects), the transformations (e.g. canonicalization) made to the data before hashing it (`Transforms` element), the algorithm used to calculate the digest value (`DigestMethod`), and that value (`DigestValue`).

After the `SignedInfo` appears the `SignatureValue`, that is the calculated value of the canonicalized `SignatureInfo` element (which includes hash value of the data specified as we have seen).

Then the `KeyInfo` is shown, which tell us about the key used during the signature. For instance, it could be a public key or the key contained in an attached credential. Public keys are transported inside a `KeyValue` element. For example, if the public key algorithm is RSA, then a `RSAKeyValue` with the information needed to construct the public key will be a child of the element before. Nevertheless, if the signature is performed with the public key associated with the sender's certificate, this certificate (or even only its serial number when the recipient already has the certificate stored locally) can be included in a `X509Data` element within the `KeyValue` element. It allows the receiver to retrieve the public key from the certificate and validate the signature. The `KeyInfo` element is not mandatory.

Listing 2.1 illustrates the structure of a `Signature` element[4] according to the standard [8].

[4]Where "?" means zero or one occurrence, "+" denotes one or more occurrences, and "*" denotes zero or more occurrences

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
       (<Transforms>)?
       <DigestMethod/>
       <DigestValue>
     </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Listing 2.1. Structure of a `Signature` element

2.4.2 XML Encryption

Through *XML Encryption* [9], confidentiality is added to the XML content so that only the holders of the key used for encryption are able to inspect the plain text of the protected data. The element `EncryptedData` will encapsulate both all information concerning the encryption procedure and the resulting cipher value itself.

For the recipient to be able to decrypt the data, the `EncryptedData` includes the `EncryptionMethod` used (for example AES or RSA) and the `KeyInfo` element with the same meaning as in XML Signature (it has actually been taken from its name space). The result of the original data encryption using the specified algorithm and key is encapsulated inside the `CipherValue` child of the `CipherData` element.

If the recipient does not have the key used for the sender when encrypting, it can be included inside the `KeyInfo` element as `EncryptedKey`. For example, it can be encrypted with the recipient's public key. In that case, when the receiver gets the message, she uses her private key for decrypting the key inside the `EncryptedKey` element, and then this key is used to decrypt the rest of the cipher data. The `EncryptedKey` element will include the `EncryptionMethod` and `CipherData` explained above to give the recipient enough information about how to decrypt the key.

The encryption can be applied to:

- An entire XML file: The XML file will be composed just by an `EncryptedData` element.

- An element of the XML file: The `EncryptedData` element will replace the plain text element, and will have a `Type` attribute specifying that the content encrypted was an element.

- Contents of an element: The `EncryptedData` element will replace the plain text of the protected content, and will have a `Type` attribute specifying that the content encrypted was an element's content.
- Non-XML data (e.g. any attached file).

The (simplified) structure of a `EncryptedData` element[4] according to the standard [9] is described in Listing 2.2.

```
<EncryptedData Id? Type? MimeType? Encoding?>
  <EncryptionMethod/>?
  <ds:KeyInfo>
    <EncryptedKey>?
    <AgreementMethod>?
    <ds:KeyName>?
    <ds:RetrievalMethod>?
    <ds:*>?
  </ds:KeyInfo>?
  <CipherData>
    <CipherValue>?
    <CipherReference URI?>?
  </CipherData>
  <EncryptionProperties>?
</EncryptedData>
```

Listing 2.2. Structure of a `EncryptedData` element

2.5 Security Assertion Markup Language (SAML)

The *Security Assertion Markup Language (SAML) v1.1* standard [10] defines an XML based syntax to represent security credentials. Although there is a more recent SAML specification v2.0 [11], when we talk about elements of the SAML assertions in this section we will follow the version 1.1. The reason is that there is not open implementation of the version 2.0 to be used in the project yet. Those two versions follow the same principles, but there are some incompatibilities (mainly syntactical, in names and attributes of the elements). The differences between SAML v1.1 and SAML v2.0 are described in [12, Section 5].

SAML is used for exchanging authentication and authorization data between different security domains (the one that provides the user's identity and the domain that provides the service). Its aim is to provide an standard technology to provide SAML Web single sign-on.

According to the SAML standard [10], an entity (the *requester*) may request a *SAML assertion* to a *SAML authority*. The `Assertion` XML element will wrap all the information related to the assertion. The `AssertionID`, the `Issuer` name and the `IssueInstant` (time when the assertion was issued) are important attributes of that root element. One child element describes some `Conditions` that can be specified for each assertion, for instance its validity period through the attributes

`NotBefore` and `NotOnOrAfter`. There are three kinds of assertions the SAML authorities may issue, which would be child elements of the root element `Assertion`. All of them will include a `Subject` sub-element with information about the holder of the assertion (the user that wants to be authenticated). This sub-element contains (either or both) a `NameIdentifier` and a `SubjectConfirmation` that gives a confirmation method for the holder to prove her identity. The confirmation method could be for example the user's public key, which can be included inside a `KeyInfo` as described in Section 2.4.1. Then, if a user can prove that she has the corresponding private key, she will prove that she has the identity specified in the assertion. Each different kind of assertion will state different facts about this subject:

The **Authentication Assertion**, given by an `AuthenticationStatement`. It specifies the `AuthenticationMethod` used and the `AuthenticationInstant` when the subject was authenticated.

The **Attribute Assertion**. It associates a set of attributes to a subject through an `AttributeStatement` element. For this purpose, child elements called `Attribute` are added. Each one associated pairs `AttributeName`–`AttributeValue`. These attributes are not standard but defined by the system designer of the system. One example of attribute could be "Role–PhD. Student".

The **Authorization Decision Assertion**. It is described in an element called `AuthorizationDecisionStatement`. This kind of assertion states whether the subject is entitled to use the service offered by the relying party (`Resource`) or not, the `Decision`. The subject will be authorized to perform a set of `Action` (e.g. read some data).

The standard [10] defines a number of request/response protocols. One of them is used to request security credentials to a SAML authority. When an entity makes a `Request` to ask for assertions it has to specify the assertions it is waiting for (i.e., if it needs authentication, attribute, or authorization decision statements).

In the `Response` the SAML authority will include the `Status` (e.g. success) of the corresponding requested assertions, according to the policies. Any number of `Assertion` elements (containing authentication, attribute, or authorization *statements*) could be included. A diagram of a SAML `Response` message is illustrated in Figure 2.3 (the message on the left). In this case, the `Response` element is carried in the SOAP Body (see Section 2.6).

To add integrity and origin authentication to the assertion requests, responses, and assertions themselves, they may be signed by the issuer adding a `ds:Signature` element as described in Section 2.4.1. When a resource receives an assertion signed by a SAML authority, it can be sure the assertion was issued by that authority and it has not been modified by another party. If the receiver trusts that authority, then it would accept the assertion as valid.

An example of a signed SAML assertion containing an authentication statement is shown in Listing 2.3. With it, the issuer "www.pdc.kth.se" assures that the subject identified by "esteban@kth.se" was authenticated at a fixed time. This assertion is valid 24 hours from the authentication time.

```
<Assertion
  AssertionID="..."
  IssueInstant="2006-04-27T00:46:02Z"
  Issuer="www.pdc.kth.se"
  MajorVersion="1"
  MinorVersion="1"
  xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Conditions
    NotBefore="2006-04-27T00:46:02Z"
    NotOnOrAfter="2006-04-28T00:46:02Z">
  <AuthenticationStatement
    AuthenticationInstant="2006-04-27T00:46:00Z"
    AuthenticationMethod="...">
    <Subject>
      <NameIdentifier Format="...">
        esteban@kth.se
      </NameIdentifier>
      <SubjectConfirmation>
        <ConfirmationMethod>...</ConfirmationMethod>
      </SubjectConfirmation>
    </Subject>
  </AuthenticationStatement>
  <ds:Signature
    ...
  </ds:Signature>
</Assertion>
```

Listing 2.3. Example of a signed SAML `Assertion`

The *SAML Bindings* specification, [13] for v1.1 and [14] for v2.0, defines several manners of carrying assertion request and responses. For example, one of them defines how to wrap assertions into SOAP messages. The requests and responses will be included into the SOAP body.

The assertions can be included as security tokens (i.e. a set of claims) in Web Services Security (WSS) messages (see Section 2.7). This case is illustrated on the right message in Figure 2.3. Through attaching SAML in the WSS header, the requester can prove something to the target of the message, e.g. its identity. It may also be used to contain information about the key used for signing some other parts of the SOAP message in these WSS messages. It is important to note the difference between the bindings specification of the SAML standard and the introduction of a SAML assertion into a WSS message: The assertion is placed in different parts of the SOAP message, and it has different purpose (in one case the issuer is just

delivering the assertion to its holder, and in the other one the assertion has an authentication/authorization purpose).

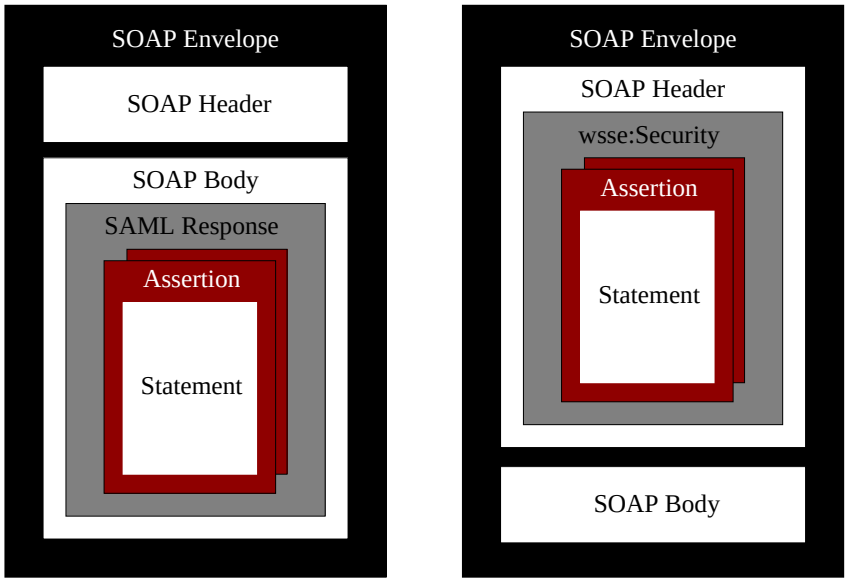

Figure 2.3. Assertion within a SAML `Response` message and a WSS Message

Both approaches may be used in the same session by a subject: For example, the subject could first request a SAML assertion to a SAML authority using the protocol explained above and the SOAP over HTTP binding specification, and after wrap that assertion into a WSS SOAP message. She could also sign some data in the message with the key referred in the assertion, if any, before sending it to the Web service. The Web service would then verify the signature and use the information included in the assertion to perform authentication and access control procedures.

2.6 Web Services

A *Web Service (WS)* is a general concept that has many definitions. The `W3C`, in a attempt to provide a common description and architecture of the general concept of Web service, defines a Web service in [15] as *"a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards"*.

SOAP is an XML-based standard for exchanging messages over a network, usually using HTTP. It defines a standard message format and guidelines for processing these messages. The structure of a SOAP message is:

- **SOAP message** (an XML document)
  - **SOAP part** (only XML content)
    - SOAP `Envelope`
      - SOAP `Header` (optional, zero or more of them)
      - SOAP `Body`, containing XML data or a SOAP `Fault` (response showing that something went wrong with the request message)
  - Zero or more **Attachment Parts**: content in XML or non-XML format (e.g. a binary file)

The *Web Services Description Language (WSDL)* is a standardized XML-based format for describing services. It includes the service interface, how to access it, etc. The client requests and service responses will be made through SOAP messages, following the information specified in the WSDL description (that could be published on the Web).

A set of specifications, known as the *WS-* family*, are being developed to cover different aspects of Web services, including security.

2.7 WS-Security

Emerging Web services security specifications are targeted to integrate currently available security technologies in addition to considering security requirements of future applications. These specifications provide a comprehensive model of security functions and components for Web services. *WS-Trust* and *WS-Federation* specifications are the extensions to *WS-Security* to provide a way of establishing, assessing and brokering trust relationships. While WS-Security is being standardized by OASIS[5], WS-Trust and WS-Federation are specifications defined by a group of several companies, like Microsoft, IBM, and VeriSign among others.

The *Web Services Security (WSS)* standard provides a way of adding XML Signature and XML Encryption for securing SOAP messages, and defines how to add identity information to those messages, such as security tokens. Therefore, it achieves message confidentiality (through XML Encryption), integrity (through XML Signature), and authentication (through including a security token). The key used for encryption/signature could be the one included in the attached credential, e.g. a X.509 certificate or a Kerberos ticket.

To achieve confidentiality and integrity, we could use *communication security*: applications send the information in clear, and other protocols located bellow these applications in the protocol stack protect it at the *packet* level. Examples of this behavior could be IPsec (network layer) and SSL/TLS (transport layer). However, protocols like WSS give us protection at the *document* level: the sensitive information is protected at the *application* level, and after it is passed through

[5] http://www.oasis-open.org/

the communication layers bellow, which will send the packets without the need of adding more security features. The latter model provides **end-to-end** security (the SOAP messages will be protected during the whole way between sender and receiver), while in protocols like SSL/TLS the messages are encrypted hop-by-hop, but they are vulnerable in these intermediate hosts. Since intermediate hops might be untrusted, the use of WSS instead of other approaches is desirable in many cases.

The current version of the so called **WS-Security 2004** standard is 1.1. This version is a review of the first 1.0 specification, with more security token profiles, errata revised, and a few new XML elements added to improve the standard. Given the recent publication of the version 1.1, no implemented open APIs to be used during the development of this project are available yet. For this reason, the 1.0 version will be studied more carefully.

The specification for each version is divided into several documents: one "core" specification describing the *SOAP Message Security* that covers the XML signature and encryption procedures to protect SOAP messages ([1] and [16]), and several token profiles about how to include security tokens in combination with the core specification (how to attach and use them in the SOAP security header defined in that standard). Thus, each security token has its own *profile* specification separated from the WS-Security core specification: Username token ([17] and [18]), SAML assertions ([19] and [20]), X.509 certificates ([21] and [22]), Kerberos Tickets (only supported by v1.1 [23]), and Rights Expression Language (REL). Figure 2.3 illustrates how a security token, a SAML assertion in this case, is included into a WSS message.

According to the WSS syntax, the SOAP `Header` will have only one child element `wsse:Security` containing all the security information:

- A security token can be included. As has been pointed, several kinds of elements could be included in this part: a `UsernameToken`, probably with a `Password`, SAML assertions in `Assertion` elements, or a `BinarySecurityToken` (an X.509 certificate or a Kerberos ticket). The type of binary token is specified in the `ValueType` attribute. As a text-based file like XML files is not suitable for including binary data, this data should be encoded before adding it to the file. The encoding algorithm used appears in the `EncodingType` attribute.

- A `ds:Signature`, the one explained in Section 2.4.1. The `Reference` member could refer to a SOAP `Body` element, while the `KeyInfo` could point to a secret key or a key contained in a security token. In the latter case, a `SecurityTokenReference` element will reference the token. It can be a local reference (referring the key of the attached token) or point to a token that is not attached but the other party knows.

Listing 2.4 shows an example of a WSS message, with a signature providing integrity to the SOAP Body part and a security token attached. The SOAP `S11:Header` contains the `wsse:Security` element. This security element is

composed of a `wsse:BinarySecurityToken` (lines 6–11), in this example an X.509 v3 certificate as specified in line 7, and a `ds:Signature` element (lines 12–36). This signature is applied to the SOAP body (line 18) and the key that the recipient will need for verifying it is the one contained in the security token, as written in line 33 (it is a reference to "X509Token", which is the ID of the `BinarySecurityToken` element that wraps the certificate). The `S11:Body`, from line 39, contains application-specific information to be processed by the recipient's application. For instance, it could be a client requesting the price of a product that the service is selling, appearing in the body the name of the product.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <S11:Envelope xmlns:S11="..." xmlns:wsse="..."
3    xmlns:wsu="..." xmlns:ds="...">
4    <S11:Header>
5      <wsse:Security>
6        <wsse:BinarySecurityToken
7          ValueType="...#X509v3"
8          EncodingType="...#Base64Binary"
9          wsu:Id="X509Token">
10          MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
11       </wsse:BinarySecurityToken>
12       <ds:Signature>
13         <ds:SignedInfo>
14           <ds:CanonicalizationMethod Algorithm=
15             "http://www.w3.org/2001/10/xml-exc-c14n#"/>
16           <ds:SignatureMethod Algorithm=
17             "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
18           <ds:Reference URI="#myBody">
19             <ds:Transforms>
20               <ds:Transform Algorithm=
21                 "http://www.w3.org/2001/10/xml-exc-c14n#"/>
22             </ds:Transforms>
23             <ds:DigestMethod Algorithm=
24               "http://www.w3.org/2000/09/xmldsig#sha1"/>
25             <ds:DigestValue>EULddytSo1...</ds:DigestValue>
26           </ds:Reference>
27         </ds:SignedInfo>
28         <ds:SignatureValue>
29           BL8jdfToEb11/vXcMZNNjPOV...
30         </ds:SignatureValue>
31         <ds:KeyInfo>
32           <wsse:SecurityTokenReference>
33             <wsse:Reference URI="#X509Token"/>
34           </wsse:SecurityTokenReference>
35         </ds:KeyInfo>
36       </ds:Signature>
37     </wsse:Security>
38   </S11:Header>
39   <S11:Body wsu:Id="myBody">
```

```
40      ...
41    </S11:Body>
42 </S11:Envelope>
```

Listing 2.4. Example of signed WSS message with binary token [1]

2.7.1 Credential and Trust Issues not Addressed by WS-Security

As we have seen, WS-Security covers confidentiality, integrity, and authentication at the SOAP message level, and we can also include and refer to security tokens. Now we deal with one question that is not addressed by the WS-Security specification: What if the sender and the receiver of the message do not implement the same security mechanism? For example, if the user cannot get a credential syntactically understandable by the service, all security procedures we have seen will be useless. As different security mechanisms as well as different name spaces are used in each domain, the same security credential might not be valid in different ones. This credential might be invalid in terms of [24]:

- *Format:* The syntax of the sent token should be understandable by the receiver. For instance, the requestor of a service could have a Kerberos ticket for authentication and message signature, but the service might only understand X.509 certificates.

- *Namespace:* The user's identity name and her attributes must be valid in the service, in order to make a successful authentication. The same entity could have different identity names in different domains (e.g., "Esteban" in domain1, "Esteban Talavera" in domain2, "etg" in domain3...).

- *Trust:* In order for a credential to be trusted, the recipient must be able to build a chain of trust from that credential to another entity the recipient trust on (e.g. a certification authority when using X.509 certificates).

Two other Web services specifications, *WS-Trust* and *WS-Federation*, have been developed to address those issues.

2.7.2 WS-Trust

WS-Trust [25] introduces appropriate security services and methods for exchanging security tokens to form the basis of trust relationships between different and heterogeneous trust domains. These services enable federation of entities to translate or exchange security credentials into other formats of security tokens. Those may also be used to exchange credentials to be trustable by the receiver when there is a lack of trust between the ends of the communication. WS-Trust is placed just over WS-Security in the protocol stack. Therefore, it makes use of the security mechanisms provided by WS-Security during the messages exchange.

WS-Trust just extends WS-Security for security token exchange and issuance, and trust establishment.

The WS-Trust standard specifies a request/response protocol for exchanging, issuing, renewing, and validating security credentials between the client requester of some service and a trusted authority called *Security Token Service (STS)*. These functions may be requested not only by the holder of the assertion it is being exchanged, but also by other entity (e.g. the service that does not understand a received credential) on behalf of that user. Some extensions in form of message exchange before the above functions, e.g. for negotiating parameters or exchange challenge/response messages to prove the possession of a secret associated with the token, are also covered by the standard.

Simplifying, a WS-Trust message exchange works as follows: If the credential is not valid for the recipient of the request, the client (or the service that receives the SOAP message with the invalid credential) sends a SOAP message to the STS with a `wst:RequestSecurityToken` (RST) element containing that credential. This message will have the same structure as the one in Listing 2.4, including in the SOAP Body the RST element. It is mandatory to specify in the `wst:RequestType` element the functionality that is being requested (e.g. security token issuance, renewal or validation). The `wst:TokenType` (optional) will specify the kind of token requested (e.g. an X.509 v3 certificate). If the client wants to specify the scope for which the requested security token is desired to be valid, she can add this information (e.g. the target service(s) or the service's realm) to a `wsp:AppliesTo` element. A RST should have either or both a `wst:TokenType` or/and a `wsp:AppliesTo` element, to tell the STS the format of the requested token. A set of `wst:Claims` may be included in the token issuance scenario. These claims could contain a certification request to be examined and signed by the STS, or the credential the client wants to exchange.

When the STS receives the SOAP message, it extracts the credential and maps it into a token valid in the requested domain (or just the one specified in the `wst:TokenType` of the request, if included). That new token is sent back in a `wst:RequestSecurityTokenResponse` (RSTR) element in the body of a WSS SOAP message. It might have an optional `wst:TokenType` element with the type of the returned token, which is included in the `wst:RequestedSecurityToken` part. Sometimes, this new token has an associated key that is generated by the STS at the same time: e.g., in Kerberos issuance the session key included inside the ticket has to be given to the client for the latter to be able to communicate with the target service, as explained in Section 2.2. This key can be delivered to the client in a `wst:RequestedProofToken`. As the key must be encrypted during the conversation to avoid another party intercepting it, it is sent in the `EncryptedKey` structure introduced in Section 2.4.2.

As this token will be signed in some way by the STS, the STS and the service should have a trust relationship (direct or indirect) for this token to be valid. The new token will be the one that the client sends to the service if the client was the origin of the token request.

A typical security token exchange scenario is described in Figure 2.4. First, the

requestor obtains its credential (Kerberos ticket, SAML Assertion, X.509 certificate, or any other) from an authority (1). This step is usually not performed every time, since the credential is usually stored in the client's local cache while it is valid. Then, the requestor sends a RST message to the STS, receiving after the RSTR with the new credential valid in the resource's domain (2), which will be used to request the service (3). It is important to note that the STS is located in the resource's domain in this example, but it could be placed in the requestor's local domain as long as the STS is trusted by the remote resource.

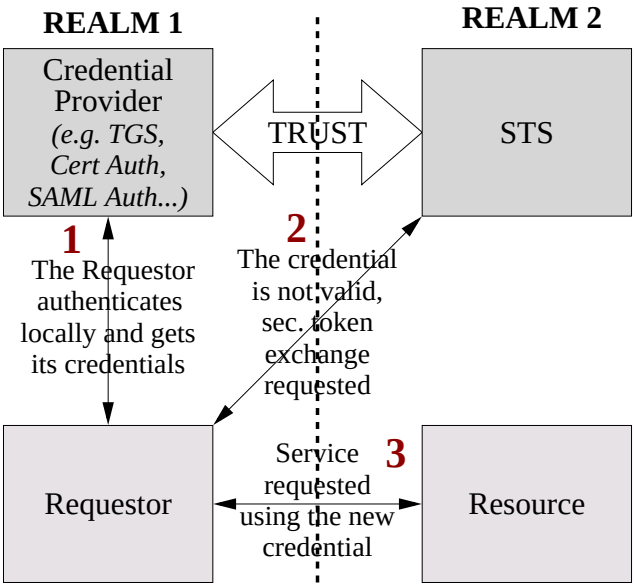

Figure 2.4. Typical WS-Trust security token exchange scenario

2.7.3 WS-Federation

> **Federation:** *"collection of realms that have established trust. The level of trust may vary, but typically includes authentication and may include authorization"* [26].

WS-Trust has solved the first of the three credential issues commented in Section 2.7.1 that are not covered by WS-Security: The issuance, renewal, and validation of a credential in a different format understandable by the recipient of the message, which can be trusted. However, WS-Trust does not specify any method for identity mapping and authorization information federation between the requestor and recipient domains.

WS-Trust only solves the third exposed point (trust) partially: Although requestor and recipient of the message do not need to have a trust relationship beforehand, there must be trust relationships requestor–STS and STS–recipient.

The requestor and the STS must have an established trust relationship for the STS to be able to trust the security token the client wants to exchange, for instance through direct trust between the STS and the issuer of that credential. This issuer could be an *Identity Provider (IP)*: an special case of STS that is able to provide identities (can make identity statements when issuing security credentials). If the STS trusts the token to be exchanged, the security token issued by the STS will be then trusted by the final recipient of the message if and only if the recipient trusts the STS. Therefore, WS-Trust does not address the problem when the requestor or the recipient and the STS do not trust each other, directly or indirectly. As shown in Figure 2.4 the requestor's realm and the recipient's realm must have an established trust relationship (direct in the illustrated example, although it could be indirect trust through a third party trusted by both).

The *WS-Federation* specification [26][6] *"defines mechanisms that are used to enable identity, account, attribute, authentication, and authorization federation across different trust realms"*. Therefore, it is focused on the second of the security token issues not covered by WS-Security (and neither by WS-Trust): Namespace translation. For example, the identity federation could be used to associate several pseudonyms with only one identity (e.g. when the same person has many email addresses). Thus, it describes how to perform identity mapping when different realms/resources accept different identities for the same subject.

The standard [26] defines two different kinds of requestors: *Passive requestors*, a Web browser only able to use HTTP and that cannot issue security tokens, and *Active requestors*, an application that is able to create Web services messages like the described in WS-Security and WS-Trust. They both have their own WS-Federation profile, separated from the "core" specification, to address how each requestor implements the general framework.

In the same way that WS-Trust extends WS-Security with new functionalities, WS-Federation is an extension of WS-Trust in order to address issues not covered by the latter: add federated identity mapping mechanisms to the WS-Trust exchange/issuance model. This is done through new services and standardized messages to manage those services. One important service is the *Pseudonym Service*, that stores information about alternative identities of the federated realms. This service allows to the same entity to have different *aliases* in different realms that host the resources. There are specific elements to place in WSS messages for getting, setting and deleting pseudonyms from the service, and some extensions to WS-Trust RST and RSTR messages for asking about how to process the pseudonyms of the requested credentials. An example scenario of the use of a pseudonym service is shown in Figure 2.5. When the IP/STS in the resource's realm receives the RST message (2), if the received identity is not valid locally, the IP/STS asks for the local alias of the subject (3). The IP/STS could also set a new temporally alias for

[6] A more recent specification (WS-Federation v1.1) has been released in December 2006. However, it was not available when studying the standards at the beginning of the thesis, so during the report we will refer to the previous version, the one listed in the Bibliography.

that identity. Then, the requestor gets the new credential with the mapped alias (4) and requests the service (5). Now, the identity of the received credential will be valid in the resource's realm.

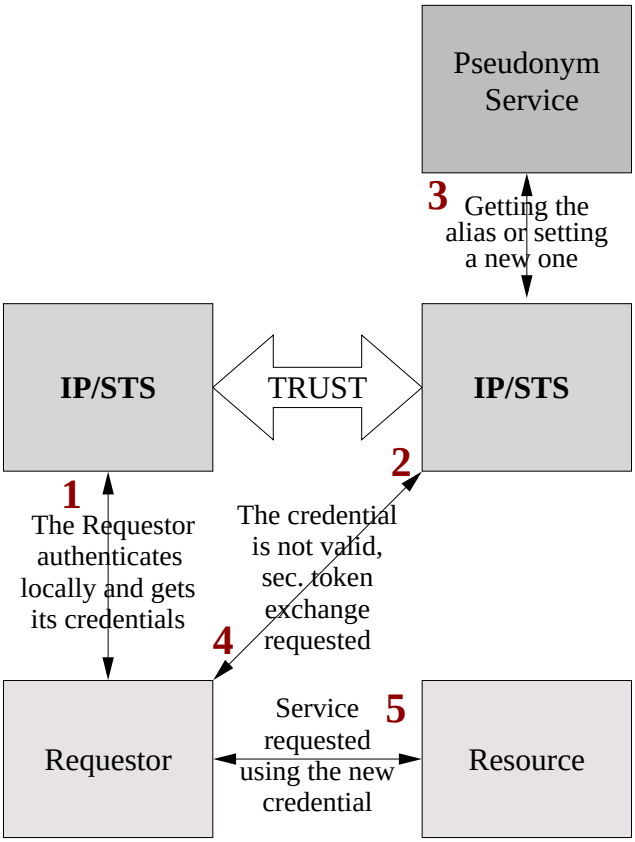

Figure 2.5. Example of credential mapping with Pseudonym Service

WS-Federation is placed just over WS-Trust in the protocol stack, and therefore it is also located above the WS-Security layer, being able to use its security services.

The definition of a protocol for dynamic establishment of trust relationships is out of the scope of WS-Federation, as stated in the specification. Therefore, it does not solve the trust issue described in Section 2.7.1.

Chapter 3

Credential Mapping

In this chapter, we provide an analysis of the Problem Statement described in Section 1.2. We study the requirements of the system to be constructed in order to solve the problem. We also include the identification of the actors and their basic interaction with the needed requirements to achieve the goals listed in Section 1.4.

Following the WS-Trust model illustrated in Figure 2.4, the conversion scenario will have 4 actors: The **Identity Provider** (the entity that issues credentials to users after some sort of authentication), the **Requestor** of the conversion, the **Resource** the Requestor wants to gain access to, and the **STS** (the entity that receives conversion requests and issues new tokens valid on the resource). First, the client will get a credential from its IP if she does not have a valid one yet. Then, the requestor will try to access to the resource and she will realize that the credential she owns is not valid to request that service. Therefore, the requestor will send a WS-Trust RST message attaching her security token (and maybe more needed information), and asking for a credential issuance of the format understandable by the resource. If the request is valid, the STS will translate the received credential into an equivalent one in the other format, returning it in the RSTR message. Additional information, like a *proof of possession* (i.e., a secret that allows the sender to prove that she is the holder of the credential) for the client, could be added to the response message. This proof of possession must be encrypted to ensure that only the credential requestor gets it. Both request and response messages must be signed to authenticate the source and be sure it was not modified during the way. Finally, the client requests the service with the new security token.

For this scenario to work, the credential generated by the IP must be trusted by the STS, and the new credential must be trusted by the target resource. Therefore, there must be previously established (direct or indirect) trust relationships between the IP and the STS, and between the STS and the target resource.

Table 3.1 shows specific information (simplified) corresponding to this general overview for Kerberos, PKI–X.509, and SAML. For each case, it includes the IP, the security credential and the main information that it contains in terms of identity, how the trust credential–authenticator is achieved, and which token the sender must

have to proof to the receiver that she is the holder of the credential.

	Kerberos	**PKI–X.509 Cert.**	**SAML**
Identity provider	KDC	CA	SAML Authority
Security credential(s)	TGT and ST	X.509 Certificate	Assertion
Typical validity period	1 day	1 year	1 year
Holder's Identity information	Principal and Realm names	DN	Name Identifier
Trust	Ticket encrypted with secret key the holder does not have (only KDC & Service do)	Signed with CA's private key	Signed by the SAML Authority
Proof of possession	Key inside the encrypted ticket is given to the holder by other means	The certificate includes the holder's public key	The assertion may contain the holder's certificate or public key (`SubjectConf.`)

Table 3.1. Peculiarities of each authentication mechanism and their credentials

The following sections give details of each conversion.

3.1 Kerberos $\Longrightarrow$ X.509/SAML Translation

In this scenario, a user who has access to a Kerberos domain (she is able to get a ticket from her local KDC) is requesting the exchange of her ticket for a X.509 certificate or a SAML assertion. Then, the client needs an STS acting as a Kerberos service.

Figure 3.1 illustrates Kerberos token translation to either X.509 certificate or SAML assertion. The actions that need to be performed for these conversions are the following (steps 1, 2, and 3 are standard Kerberos message exchanges, and they have to be performed only once until the expiration of the generated tickets):

1) The client performs login to her local machine, inserting her Kerberos username and password.

2) With that information, a TGT is requested to the AS. The returned TGT is encrypted with the key shared between the AS and the TGS. The corresponding key included inside the ticket is sent to the client, and it will be used on the

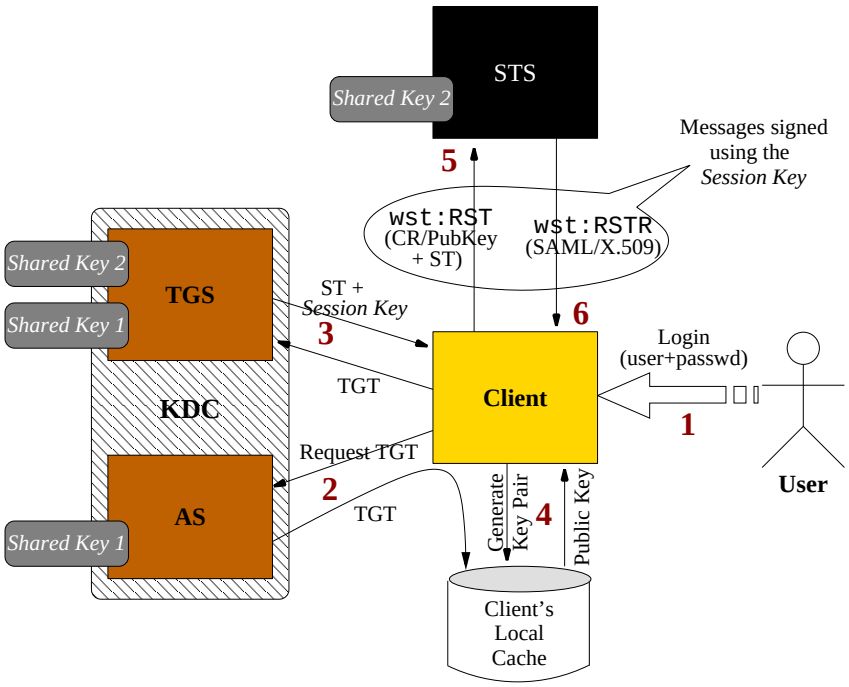

Figure 3.1. Kerberos Ticket → SAML/X.509 conversion

communication between the client and the TGS. The TGT is stored in the client's local cache for future use.

3) The TGT is used to request to the Ticket Granting Server a service ticket valid for the STS. If the client is authorized, the TGS sends her the ST (which is encrypted with the key shared between the TGS and the STS) and the corresponding session key (which is also included inside the ST). Now, the user has a security credential stating her identity, and a key to prove to the STS that she is the real holder of that token.

4) A public and private key pair is generated and stored locally in the user's machine. The private key will never leave the user, while the public key will be included in the request message. For the Kerberos → X.509 conversion, it is obvious that the certificate needs to have the client's public key. Let's see what happens with the Kerberos → SAML conversion: As mentioned in Section 2.5, the `Subject` may include a `SubjectConfirmation` that gives a `ConfirmationMethod` for the receiver to prove that the assertion came from the entity with the specified `NameIdentifier`. The standard confirmation methods are defined in [13, Section 5], of which three could be used when generating the assertion:

- **Sender Vouches:** *"Indicates that no other information is available about*

the context of use of the assertion. The relying party SHOULD utilize other means to determine if it should process the assertion further" [13]. As the final resource could have no other way to prove the client's identity, this method is not suitable for our system.

- **Bearer:** *"The subject of the assertion is the bearer of the assertion"* [13]. This method is commonly used, since many times the assertion is encrypted by the issuer before sending it to the holder, and then encrypted again by the latter when communicating with the resource. Then, no other party could have access to the "plain text" assertion, and the sender is therefore the subject that appears on it. In our case, this method is not useful since the assertion will be sent unencrypted by the STS. If this method is used, a third party that is listening to the translation message exchange could get the assertion and present it to the resource claiming to be the holder.
- **Holder of Key:** The `SubjectConfirmation` must include a `ds:KeyInfo` element, with information needed to obtain a key. Here, *"The subject of the statement(s) in the assertion is the party that can demonstrate that it is the holder of the key"* [13]. This method fits perfectly in our model: The client will send a public key to the STS, which will be included in a `ds:KeyInfo` element. Then, the client is able to prove to the target resource that she has the corresponding private key, and therefore she is the entity specified in the `Subject`.

5) The client now constructs the request message. The ST will be attached as `BinarySecurityToken` in the WS-Security header, and the message will be signed using the session key (which is inside the ticket) to add integrity and authentication to the SOAP message. The SOAP body will contain a WS-Trust `RequestSecurityToken` element. The `RequestType` will be set to token issuance, to inform the STS which service is being requested. The RST element will be different depending on the conversion:

   - In the *Kerberos → X.509 conversion*, the client generates a certification request to be included in the RST element as a `Claim`. The CR contains the distinguished name of the client, and the previously generated public key. The `TokenType` element is set to "X.509 v3 certificate".
   - In the *Kerberos → SAML conversion*, the client will just send the public key inside a `KeyInfo` element as a `Claim`, so that the STS can include it in the assertion allowing the client to prove that she is the holder of the assertion. The `TokenType` element is set to "SAML assertion".

6) The STS receives the message, decrypts the service ticket, gets the session key, and verifies the signature. If the verification is successful and the client is allowed for that credential translation according to the STS' policy, the STS will inspect the SOAP body and issue the token specified in the `TokenType` element. For this translation, the STS basically has the client's `PrincipalName` and `RealmName`

from the ticket to be included as her identity in the resulting token. The STS must also check the `start` and `end` time when the ticket is valid, and it must not issue a credential valid outside that time frame. For example, if the ticket expires in four hours, the generated security token must not be valid after that time, although the validity could be even less depending on the STS' policy. Depending on the token to be issued, the following actions will take place:

- In the *Kerberos → X.509 conversion*, the STS checks that the DN in the certification request matches the identity of the client specified inside the ticket. Then, it will add the remaining fields: `serialNumber`, the `validity` of the certificate, and the `issuer` DN. This information is finally signed with the STS' private key.
- In the *Kerberos → SAML conversion*, the STS creates a SAML assertion filling the `AssertionID`, the `Issuer` field with the STS' name, the `IssueInstant`, and the `Conditions NotBefore` and `NotOnOrAfter`. The assertion will contain an `AuthenticationStatement`, stating which was the `AuthenticationMethod` (Kerberos), the `AuthenticationInstant`, and the subject's `NameIdentifier` constructed from the client's principal and realm names which appear in the ticket (it will be *principalName@realmName*). For the final recipient of the assertion to be able to check that the sender is actually the entity identified in the assertion, the client's public key is included as `SubjectConfirmation`. Then, the holder of the assertion will be the one that can prove to be in possession of the corresponding private key (in the same way a X.509 certificate works). Finally, the assertion is singed with the STS' private key, so STS needs to own a certificate.

The generated credential will be packet in the `RequestSecurityTokenResponse` message that is sent to the client. This SOAP message will be signed using the Kerberos session key again. Therefore, the client will be sure about the origin of the message, since the STS is the only one that can decrypt the ticket to get the session key. The attachment of the ST is not needed for this message, since the client already has the session key.

After this request/response exchange, the client has the needed credential that can be used for authenticating to the target service until its expiration. When that resource receives the assertion, it will verify the signature. Therefore, the security token will only be accepted **if the target resource trusts the STS**. The resource must either trust the STS certificate directly or be able to construct a chain of trust from the STS' certificate to a trusted certificate.

In most cases, the STS will be in the same local domain of the client: The user will request a service ticket valid for the STS, the entity that will provide the security token issuance service. This is the case described above. However, the STS could be placed in a remote realm, and the client would be able to access that service through the Kerberos cross-realm operation explained in Section 2.2.

Of course, for the latter scenario client and STS domains should trust each other beforehand for the user to be able to perform the cross-realm authentication in the remote KDC of the STS domain. If the client uses cross-realm authentication, she must first get a ticket from her local KDC to be used in the remote KDC, and then use that ticket in step 2 instead of sending the username/password. Besides this additional actions, the operation has the same steps.

3.2 X.509 $\Longrightarrow$ Kerberos Translation

This is the case when a user has a X.509 certificate, but she is not able to use it for accessing to a target resource in a remote realm. Instead, that realm is using Kerberos for authentication, and therefore the client needs to get a valid service ticket for requesting the service.

3.2.1 First Approach: Issuing STs

In this solution, illustrated in Figure 3.2, the STS will issue temporal Kerberos credentials (service tickets) to be used on a certain service given the client's certificate.

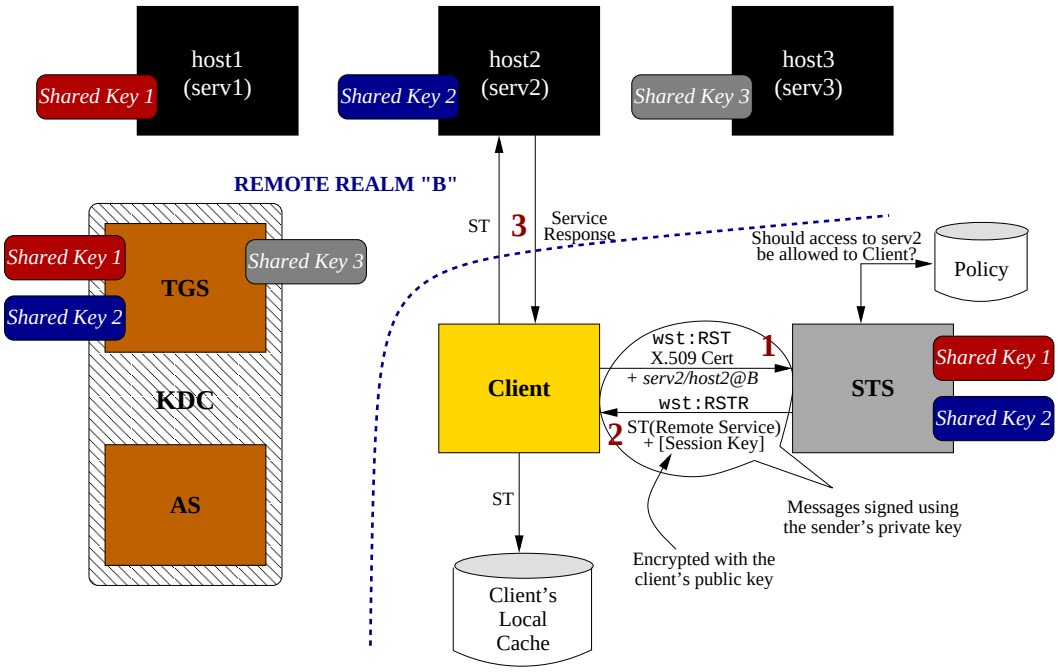

Figure 3.2. X.509 → Kerberos conversion, first approach (issuing STs)

The steps needed for a client holding a certificate to access to `serv2` are:

1) The client generates the request message to be sent to the STS. The X.509 certificate's information will be attached as `BinarySecurityToken` in the WS-Security header, and the message will be signed with the client's private key (whose corresponding public key is inside the certificate). The SOAP body will contain a WS-Trust `RequestSecurityToken` element. The `RequestType` will be set to token issuance, to inform the STS which service is being requested. The RST element will also specify which service of which realm the client wants to access to (in our example, *serv2/host2@B*). This will allow the STS to know which master key must be used when constructing the ticket. The `TokenType` element is set to "Kerberos ticket".

2) The STS receives the message and verifies the certificate itself. **The client's certificate must be trusted by the STS**. Then, the STS gets the client's public key from the certificate and verifies the signature. If the verification is successful and the client is allowed for that credential translation according to the STS policy, the STS will see that a ST for an specific kerberized service is requested. Now, the STS can check in its local policy whether it should issue a ticket given the client–service pair or not. The policy could allow a client accessing to some services of a remote realm, and deny her accessing to other services of the same realm. If access is granted, the STS starts generating the ST:

   - A random session key for the communication between the remote service and the client is generated. The encryption algorithm of the key must be supported by the remote service specified.
   - The `client`'s principal name will be the CN of the DN in the certificate, and the `realm` the STS' local realm.
   - The `authentication` and `start` times are set to the current time, and the `end` will be short to limit the risk of malicious use of the ticket by the client. At the same time, it should be long enough to allow the client making multiple requests for the same service without a new interaction with the STS. For example, the lifetime could be one hour, and never longer than the certificate validity.
   - All this information is encrypted using the master key shared between the service of the remote realm and the STS. Then, the target application server will be able to decrypt it and get the session key to communicate with the client.
   - The principal `name` of the service is filled with the service name sent by the client (*serv2/host2@B*).

   The generated credential will be packet in the `RequestSecurityTokenResponse` as `BinarySecurityToken`. As the client needs to know the generated session key, it is first encrypted with the client's public key (to avoid any other party accessing it) and then delivered as `EncryptedKey` in a `RequestedProofToken`

element. The client receives the ST, which could be stored in her local cache for a later use, and decrypts the session key with her private key.

3) Now, as in a usual Kerberos operation, the client presents to the remote service the new ST, proving that she also possesses the session key included in it, and if authorized the application server will execute the requested actions.

In this example, the Kerberos realm has 3 services (`serv1`, `serv2`, and `serv3`), but the local administrator decided to give to the STS the keys of only two of them (`serv1` and `serv2`). Thus, the STS can issue tickets for those two services.

Note that no interaction between client and service's local KDC is needed in this model.

The required **trust** between STS and target service is given by the master key: The STS will have the secret key shared between the service and its KDC. When a KDC of a Kerberos realm wants to allow a STS (local or remote) issuing tickets for certain services within that Kerberos realm (it could be all services or only a subset), the KDC has to give the STS the corresponding master keys of those services. Now, two problems arise:

- As the STS does not have direct access to the KDC's database, but the KDC sends it the keys, a method to keep the keys updated needs to be implemented. This will be more complex if the STS has keys from many different realms: If one realm changes the key of one service the STS can access to, the realm has to send the new key to the STS. Otherwise, the tickets that the STS issues from that time for that service would be invalid.
- The master keys are now vulnerable at the STS. Before, an administrator of a Kerberos realm only needed to "care" about keeping the keys safe in the local KDC's database. Now, if the realm gives some keys to many different STS, one attack to one of them compromises the keys. The security administrator could not rely on the security level of the STS, especially if it is located in a remote environment, and deny the key delivering.

3.2.2 Second Approach: Issuing TGTs

This model is similar to the Kerberos cross-realm operation described in Section 2.2. In this approach, illustrated in Figure 3.3, the STS acts as local KDC for the client, issuing a remote TGT to be used for authenticating to the remote TGS. To achieve this, as explained in the cross-realm model, STS and remote TGS must share a secret key and have a remote TGS principal (*krbtgt/remote.realm@sts.realm*).

A holder of a X.509 certificate requesting a service at a remote Kerberos domain needs to perform the steps shown in Figure 3.3:

1) The client generates the request message to be sent to the STS. The X.509 certificate's information will be attached as `BinarySecurityToken` in the WS-Security header, and the message will be signed with the client's private key

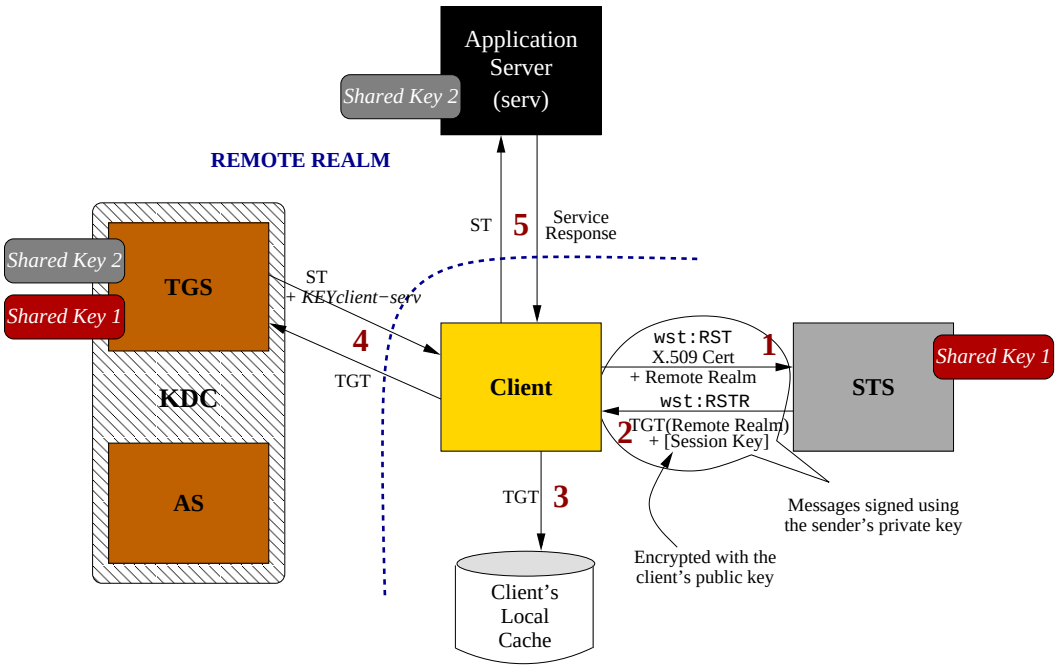

Figure 3.3. X.509 → Kerberos conversion, second approach (issuing TGTs)

(whose corresponding public key is inside the certificate). The SOAP body will contain a WS-Trust `RequestSecurityToken` element. The `RequestType` will be set to token issuance, to inform the STS which service is being requested. The RST element will also specify the realm of the target service in an `AppliesTo` element. This will allow the STS to know which shared key and remote TGS principal must be used when constructing the ticket. The `TokenType` element is set to "Kerberos ticket".

2) The STS receives the message and verifies the certificate itself. **The client's certificate must be trusted by the STS**, so the STS should have the issuer's certificate (or the certificate of its issuer and so on) as trusted certificate, to verify the certificate's signature. Then, the STS gets client's public key from the certificate and verifies the signature. If the verification is successful and the client is allowed for that credential translation according to the STS policy, the STS will see that a ticket for a specific remote realm is requested. Now the STS starts generating the TGT:

   - A random session key for the communication between the remote TGS and the client is generated. The encryption algorithm of the key must be supported by the remote realm specified.
   - The `client`'s principal name will be the CN of the DN in the certificate,

and the `realm` the STS local realm.

- The `authentication` and `start` times are set to the current time, and the `end` will be short to limit the risk of malicious use of the ticket by the client. At the same time, it should be long enough to allow the client making multiple requests to the remote realm without a new interaction with the STS. For example, the lifetime could be one hour, and never longer than the certificate validity.
- All this information is encrypted using the key shared between the TGS of the specified remote realm and the STS. Then, the TGS will be able to decrypt it and get the session key to communicate with the client.
- The principal name of the service is filled with the remote TGS principal name (*krbtgt/remote.realm@sts.realm*)

The generated credential will be packet in the `RequestSecurityTokenResponse` element as `BinarySecurityToken`. As the client needs to know the generated session key, it is first encrypted with the client's public key (to avoid any other party accessing it) and then delivered as `EncryptedKey` in a `RequestedProofToken` element.

3) The client receives the TGT and stores it in her local cache. The session key is decrypted with her private key.

4) Now, as in a usual Kerberos cross-realm operation, the client presents to the remote TGS the new TGT, proving that she also possesses the session key included in it. The TGS will then issue a ST for the requested service with a new key for the client-service communication (after checking the ticket validity and so on, and that the client is allowed to use that certain service).

5) The client may now request the service with the ST, and if authorized the application server will execute the requested actions.

Note that one shared key between the STS and the remote TGS is enough for us, since the cross-realm authentication is performed only the direction STS $\rightarrow$ TGS. If we would want bi-directional cross-realm authentication, so that a Kerberos client from the remote realm could get a ticket to be authenticated on the STS, we would need another shared key and principal for that direction.

The **trust** between STS and remote KDC is given by the shared key. The remote KDC administrator will create a secret key and the remote TGS principal for this cross-realm authentication, and give a certain amount of rights to tickets issued by the STS. This means that the STS will be able to access to some resources at the remote realm, which might not be all services that it provides. Then, when issuing the Kerberos ticket to a client, the STS will **delegate** the right of using all services that it is entitled to use to the client. However, depending on the TGS and target service's policy, the access could be denied for certain clients by the service's access control mechanism.

3.2.3 Comparison and Conclusion

Table 3.2 summarizes the advantages of each of the two approaches explained for the X.509 Certificate → Kerberos ticket conversion.

First Approach (issuing STs)	**Second Approach (issuing TGTs)**
• *Policy decisions:* The STS can allow or deny access to certain services in the same Kerberos realm, according to its policy. In the other approach, the STS can only allow or deny access to "entire" realms: When generating remote TGT the client may request all services at the remote realm the STS has access to • The client does not need to interact with the remote KDC, she goes directly to the target service after getting the ticket	• It follows the standard Cross-Realm operation, no changes at the remote KDC are needed. Although interaction between client and KDC is needed, the KDC will treat the client as a Kerberos principal from another trusted Kerberos domain • *Easier to administrate*, only one key is being delivered to the STS and updated when needed • *No master keys are compromised.* If the key shared by STS and KDC is stolen, it is enough with changing that one • The KDC can change master keys as needed, without any changes on the STS (the shared key will still work)

Table 3.2. X.509 → Kerberos approaches comparison

As we can see, the two main disadvantages of the first approach are that the master keys are compromised since they leave the KDC, and the key administration when many KDCs and STS are working together becomes arduous (a change of a master key could involve many key "refreshments", one for each STS in possession of that key). Moreover, if the administrator of a Kerberos domain wants to revoke the right of accessing a service by the STS, the key for that service must be changed on the rest of STS using it. This becomes much more complex when the KDC decides to finish the trust relationship with a STS: All master keys that were given to the STS must be changed. In the second approach only one change on the KDC's database, stating that tickets issued by that STS are not allowed to use one service (or any service) any more, is enough.

We want our solution to be as **standard**, **scalable**, and **usable** as possible. These characteristics are only fulfilled by the *second approach*. The possibility of being able to choose which clients are authorized to use which services of which realms when a STS receives a request is an important advantage, but it is not enough given the disadvantages. If Kerberos administrators does not want to rely on the STS to keep services' master keys secure, they will not let these keys be stored on the STS. As clients speak with services without going first through the KDC, administrators will lose control on those operations.

For those reasons, we have decided to implement the second model (in which the STS issues TGTs instead of STs) of Section 3.2.2 in our system.

3.3 Security Considerations

The main security issues (some of them were introduced in Section 1.4) which have been taken into account and which the proposed solution should manage are the following:

- The sender of every message exchanged between the parties while the translation is being performed must be **authenticated**, and the **integrity** of these messages must be preserved from sender to receiver (e.g. through digital signatures). Then, the receiver of a message must check from where it is coming and if the information has been changed during the communication.
- **Confidentiality** must be assured for sensitive information that other parties not involved in the conversation must not know, such as secret keys.
- The credential translation should not involve loss of security, or at least the loss must be limited to "acceptable" levels. The final receiver of the translated security token should be as certain of its authenticity as she is when receiving credentials generated by "usual" means instead of by conversion.
- Given the trust requirements that have to be fulfilled beforehand to be able to make the conversion, the requested credential must be **trusted** by the final resource.

Chapter 4

Related Work

This chapter introduces some projects that try to provide SSO for users requesting remote services on a heterogeneous federation.

4.1 Shibboleth

Shibboleth[1] [27] is a project that has created an architecture and an open source implementation on it to provide federated identity and single sign-on (SSO) features across different institutions. This means that members within the same federation can share identity information, agree in a common set of policies, and create a trust relationship. Shibboleth is designed to be used in *Web browsers*, and check if a member who navigates with the browser is entitled to access a resource located in another organization, based on the information the home institution has about that user. Shibboleth is standard-based, built over SAML (currently in the version 1.1). It is a project of *Internet2/MACE*[2], being developed mainly in some universities in the USA, such as The Ohio State University. Shibboleth is being used in several high education institutions through identity federation initiatives, like *Haka*[3] in Finland and *InCommon*[4] in USA.

When a member wants to access to a resource inside the same federation in a remote realm, she first authenticates to its home *Identity Provider (IP)*. Then her home IP sends to the *Service Provider (SP)* of the resource domain a SAML assertion containing the attributes of the specific user. Based on these attributes, the SP will allow or deny the user to execute the action with the remote resource. For example, the user could have two attributes: one could say that she is a student of the institution `I`, and another one that she is registered in the course `C`. If the security policy of the SP states that the students belonging to the institution `I` or

[1] http://shibboleth.internet2.edu/
[2] http://middleware.internet2.edu/MACE/
[3] http://www.csc.fi/suomi/funet/middleware/english/index.phtml
[4] http://www.incommonfederation.org/

people following the course `C` (or people that have both conditions at the same time) can use the resource, it will grant access.

As shown in Figure 4.1, the Shibboleth system's most relevant components are [28] (Shibboleth releases provide two implementations, one to be included in the identity provider and the other in the service provider side):

- **Identity Provider:** It authenticates principals and gives them attributes, which are represented as SAML assertions. Each principal of a federation must be registered in a IP of that federation. Three main entities can be distinguished at the IP side:

  - *Authentication Authority:* It issues authentication assertions to principals. When a principal is authenticated, the IP gives her a temporal "random" identity, which will be the content of the field `NameIdentifier` in the assertion. This temporal ID is called a Shibboleth *handle*. The reason of filling this temporal ID name instead of the user's "real" ID (e.g. "esteban@nada.kth.se") is to keep privacy: The IP will map internally the random ID with the real one, but the SP will not be able to know the user's ID unless it is authorized by the user. Then, the SP will be able to ask the IP for attributes of the temporal ID, and the IP will answer with attributes of the corresponding user. Alternatively, the Authentication Authority may include attributes when sending the user the response message after authentication, in addition to the authentication statement (this method is called *"attribute push"* [28]).

    Shibboleth does not specify which method should be used to authenticate users. It will be the protocol implemented in the IP's local realm (e.g., it could be Kerberos, PKI, etc.).

  - *Attribute Authority:* Issues attribute assertions for a specific user under SP requests (giving the handle). The attributes of each user will be stored internally in the IP, for example in a data base.

  - *Single Sign-On Service:* Processes authentication requests received from SPs through the Web browser. It is the entity that begins the authentication process.

- **Service Provider:** The side where resources are located. Provides a security context for service requesters, which gives the resource's access control mechanism the information to allow or deny access to the resource. The three main parts at the SP side are:

  - *Assertion Consumer Service:* It receives the authentication assertions generated by the IP's Authentication Authority, and after requesting the user's attributes if needed, creates a security context for the user at the SP side.

– *Attribute Requester:* When the user contacts the Assertion Consumer Service sending the authentication assertion, the latter could decide that it needs more information in order to create a valid security context (when the SAML assertion that the user received from her IP when authenticating does not contain "pushed" attributes, or additional attributes to those ones are needed). The Attribute Requester will then exchange messages directly with the IP's Attribute Authority, requesting attributes that the SP needs to decide whether the user should have access to the resource or not.

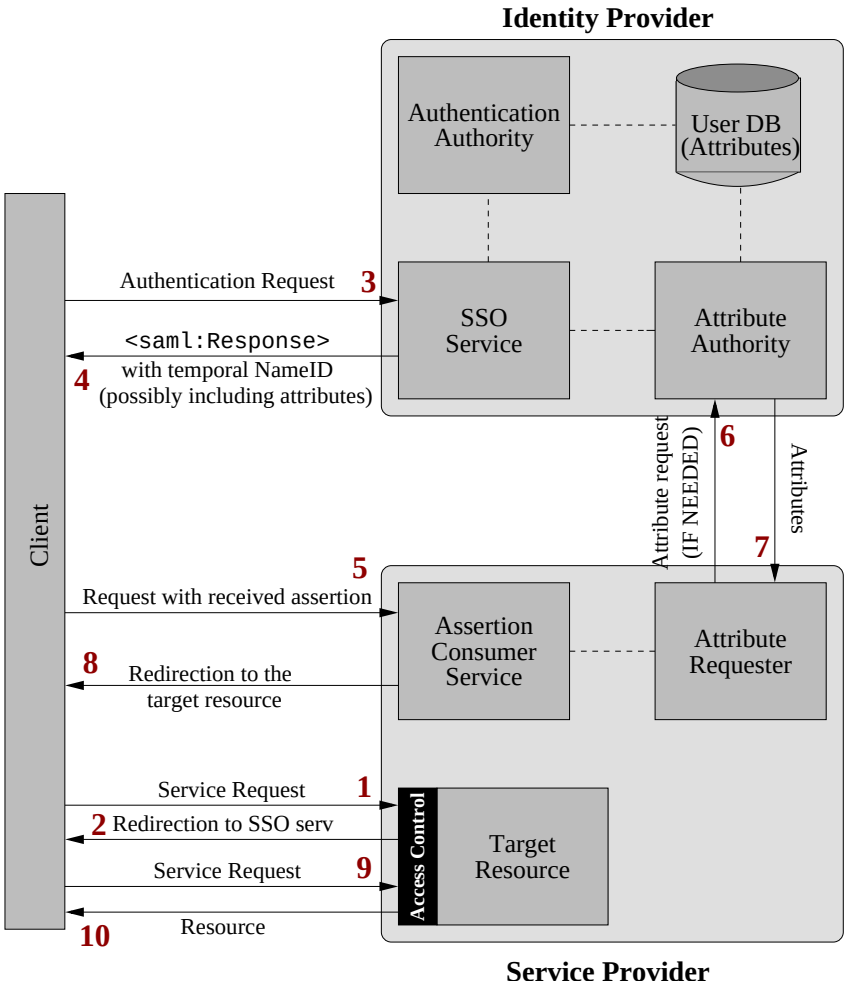

Figure 4.1. Access to resources within a Shibboleth federation

Figure 4.1 shows the actions executed when a user requests a resource running inside the same federation [27]. The following steps will be carried out:

1) A user requests a resource located at a Service Provider inside a federation that user is member of. The SP checks whether the user has a valid security context (which has been established before in a previous access of the same client to the SP). If there is one, the client just gets the resource if she has the needed attributes.

2) The client is redirected to her local IP by the SP.

3) The client makes a request to the SSO Service, specifying the target resource. If the principal does no have security context at the IP side yet, authentication is performed and the security context is created[5]. Then, the Authentication Authority generates the `AuthenticationStatement` for that client (with a temporal handle as Name ID).

4) The SSO gives a signed Assertion to the client, which contains her authentication statement and might include pushed attributes.

5) The client requests to the Assertion Consumer the establishment of a security context at the SP[6], attaching the assertion received from the IP. The Assertion Consumer checks the statements in the assertion. If all needed information are included, it jumps to step 8. Otherwise, the Attribute Requester is used to get the missing attributes.

6) The Attribute Requester asks the IP's Attribute Authority for certain attributes of the user (the ones required to access the resource). The handle is sent, to show the IP who the user is.

7) The Attribute Authority gets the requested attributes for the specific user, and returns them to the SP.

8) The Assertion Consumer creates a security context for the user at the SP and redirects her to the resource.

9) The client requests the service again (in the same way she did in step 1).

10) As the client has now a valid security context, the service is executed.

All these steps are done keeping user's privacy: The home IP and the user are able to control the information sent to the remote SP.

[5] As Shibboleth is designed to provide Single Sign-On, the user will need to insert her credentials (e.g. Kerberos username/password) only once per session. After that, the user will be able to access resources without the need of login again to the system, since a security context is created at the IP on the first access, and it will be used on the following accesses from the same client.

[6] Note that a security context is created separately at both IP and SP. If the user has a valid security context at SP at the beginning, she does not need to go to her local IP to get credentials. On the other hand, if the user has a valid security context at the IP, authentication is not needed to get an assertion for that recipient.

Without a SSO mechanism like the proposed by Shibboleth, the user would need to maintain an account in every realm where she has access to a resource, and do login every time she wants to access a resource in a different domain by other means, like username and password. This makes the administration job of realms easier (the administrators only need to care about attributes, like "PhD. student", rather than individual identities). The access to resources by users becomes more "comfortable", as it is not needed to remember many different usernames and passwords and do login in every access.

The Shibboleth protocol defines the explained mechanisms to exchange SAML attributes between Identity and Service Providers, allowing SSO within a federation. However, it does not specify standard attribute names, but lets each federation define the attributes to be used. This is, Shibboleth does not say which attributes an entity inside a federation may have (e.g., `IDNumber`, `FirstName`, `BirthDate`...). Instead, each federation chooses the set of attributes that can be given to their subjects.

The main advantage of our solution in comparison with the Shibboleth model is that in Shibboleth federations are *static*. New realms can join an existing federation, and some realms inside that federation could leave it, but these processes need some time to be performed. With STS, federations can change in a more *dynamic* way: just one key exchange between client's or resource's domain and the STS allows it to issue credentials for existing clients or to existing resources in the new domain.

4.1.1 Interoperability between Shibboleth and STS

Suppose a STS being part of a Shibboleth federation with certain attributes giving access to certain services within that federation. Suppose a client that is registered **outside** the federation, but which has a credential (e.g. a Kerberos ticket, a X.509 certificate, or a SAML assertion) trusted by the STS. Then, a new conversion would be desirable to allow that client accessing to the resources inside federation which the STS can use.

Today, Shibboleth only supports protocols for requesting and delivering short-term SAML Assertions (using *bearer* as `ConfirmationMethod`) for SSO. The introduction of more actors and roles requires new profiles, and even WS-Trust is not supported yet. Anyhow, new specifications they are working on for the next version (*Shibboleth 2.0*, which will be based on SAML v2.0) would be absolutely required to do some of the actions we are proposing since right now is not possible to talk to the IP with anything but a browser.

Figure 4.2 shows a possible scenario where a STS works within Shibboleth. The STS must already be a **member** of the *same* Shibboleth federation where the resource that the client wants to access to is located.

The needed steps are:

1) The client wants to access to a certain resource located in a certain Shibboleth federation. Then, she asks the STS inside that federation for valid credentials for that resource. The STS authenticates the client through a credential that she

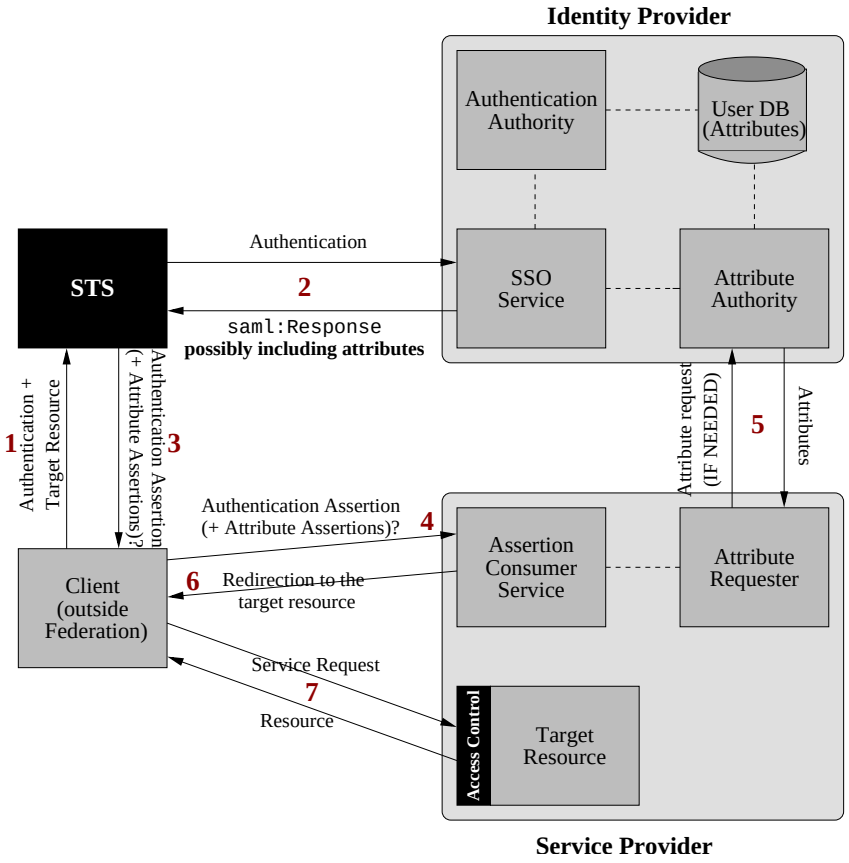

Figure 4.2. STS – Shibboleth interoperability approach

already has and that is trusted by the STS (e.g., a Kerberos ticket or a X.509 certificate).

2) The STS is authenticated by its local Identity Provider (if it does not have a security context yet), and asks for assertions to use on the specific service that the client is requesting. Here, the STS could just get an Authentication Assertion with the temporal handle, or even *pushed attributes*.

3) The received assertions are delivered to the client.

4) The client uses the assertions to start a security context at the Service Provider.

5) If the attributes were not included in the response message of step 2 (and therefore neither in step 3), or more attributes are needed, the Service Provider could ask for attributes of the identity contained in the authentication assertion received from the client.

6) After creating a security context for the client, she is redirected to the resource.

7) The client requests the service.

Given the information we have been able to gather, the following issues need to be solved for this proposal to work:

- *Can the STS ask for an authentication assertion and/or pushed attribute assertions in step 2?* As we have said, the only way to speak with the IP on the current version is through a browser, which will give you an assertion and redirect you to the SP. An alternative interface to make possible attribute requests by their owner would be desirable. Then, the STS could get directly its attributes from the IP.
- *Could the Client use the STS' Authentication Statement with the temporal random NameID for starting a security context in step 4? Could the Client directly use the STS' Attribute Statements to do that?* As they are **bearer** assertions, it should not be a problem for another party to use those assertions on the SP. The STS would get the assertion in a secure way from its IP, and then it could give them (also in a secure way) to third parties. In this case, the STS would **delegate** all its attributes, or certain number of them, to another client outside the federation by delivering the corresponding attribute assertion received form the STS' local IP to that client.

In short, we need the STS having **access** to its attributes (Shibboleth has to provide an interface for allowing members to ask their IP directly for attribute assertions), and being able to **delegate** them to external clients for a short time period (just the needed for the client to access to the resource). Shibboleth is a "young" project that is continuously being extended with new functionalities. When these actions are possible, the STS will be able to work with Shibboleth following the proposed approach.

4.2 KX.509 Protocol

The *KX.509 Protocol*[7] [29] is a protocol that allows a user to obtain a short-term X.509 certificate based on her Kerberos ticket. It has been designed and implemented in the University of Michigan. This protocol works as follows:

1) The user performs the usual Kerberos login to get the Ticket Granting Ticket from the Kerberos Distribution Center. KX.509 is not involved in this action. The TGT is received and stored on the user's workstation.

2) The KX.509 protocol introduces a new kerberized service in the network, the *Kerberized Certificate Authority (KCA)*, which will be the entity that issues certificates based on the user's Kerberos identity. The KX.509 mechanism first creates a public/private key pair on the user's machine and uses the issued TGT to request a ST for the KCA.

[7] http://www.kx509.org/

3) Once the local workstation has the ST and the key pair, it sends a certificate issuance request message to the KCA. This message contains the ST and the generated public key (the private key never leaves the user's machine). As every communication between a client and a kerberized service, it will be protected by the session key included in the ST.

4) The KCA determines the identity of the user, creates a temporally (usually with the same lifetime than the Kerberos credentials) certificate for that user containing her public key, and signs it with the KCA's private key (the KCA owns a certificate). This certificate is finally sent to the user and stored on the local cache.

This process is transparent to the user. The user just type his Kerberos password (the "login" to get the TGT), and the KX.509 module will do the rest on behalf of her. As a result, after the login the user will have stored both a TGT and the corresponding X.509 certificate ready to be used (e.g. in a Web browser SSL session). It is important to note that the KCA's certificate must be trusted by the party the user presents her certificate to in order to be valid.

In comparison with our STS, we could say that this solution is only suitable for Kerberos to X.509 translation, while the STS can make many other conversions between different formats. Moreover, the STS is following specifications like WS-Trust, and no changes at the resource's KDC are needed. On the other hand, the KX.509 protocol needs to add modules to the KDC's host which implement the functionality of creating short-term certificates when the user requests a TGT.

4.3 Kerberized Credential Translation

The *Kerberized Credential Translation* [30] is a system designed and implemented in the University of Michigan. The purpose of this mechanism is to provide access to remote Kerberized services through a Web browser using the user's certificate. This is a common scenario, since client authentication with browsers is usually performed through SSL/TLS using X.509 credentials, and Kerberos is used on many realms to control the access to local resources. If the user does not have a X.509 certificate but uses Kerberos as local system, the protocol described in Section 4.2 can be used to get that credential, rather than sending the Kerberos identity and password through a SSL browser session. If the password is sent, then the remote Web server could impersonate the client using that password to perform actions the client is not requesting. The typical actions carried out to enable a user accessing a remote resource in this model are:

1) The remote Web server that is being accessed with the browser authenticates the user by means of her certificate.

2) Here the so called *Kerberized Credential Translator (KCT)* comes on stage. It performs the opposite action carried out by the KCA explained in Section 4.2:

It is a **service extending the TGS** that translates X.509 credentials into Kerberos tickets. First, the Web server needs to get a service ticket for this service from its local KDC. This ticket is then presented to the KCT, to request access to the service the user is asking for. The message will contain the client's certificate.

3) The KCT validates the request and creates a temporal ST *for the user* valid on the target service. This ticket and the session key protected with the key shared between the KCT and the Web server is sent to the Web server.

4) Now the Web server acts on behalf of the user, accessing to the requested service using the received ST. This ticket should be cached by the Web server, so that it can be used in another request from the same user without the need of another ticket generation (until it expires). However, the service ticket's lifetime should not be long to avoid misuse on the user's behalf.

This model provides single sign-on. The user does "login" once locally (e.g. to the local Kerberos system, getting a certificate through the KX.509 protocol), and after she is able to access remote resources without additional user interaction.

The comparison between the STS and this protocol results in the same conclusions as when comparing STS and KX.509: The Kerberized Credential Translation protocol only makes one conversion, and the remote KDC's implementation has to be changed to add the translation functionality.

Our first approach for X.509 → Kerberos translation explained in Section 3.2.1 is similar to this protocol. There are two main differences:

- In our approach, the client requests the conversion and after that the service with the new ST by herself. There is not another entity (the Web server in the Kerberized Credential Translation protocol) acting on her behalf in principle.
- In the Kerberized Credential Translation protocol, the KCT has access to the KDC database to retrieve the master key of the service which is used to encrypt the generated ST. Therefore the KCT needs the physical security as the KDC, since if one key is stolen, the security of the corresponding resource is compromised (the thief could generate a valid ST for that service). For this reason, in this model the KCT runs in the same machine as the KDC. In our approach, it is not feasible for the STS to run in the KDC, because the STS service should give access to different KDCs, and provide different conversions. Moreover, the STS could even be remote to the Kerberos realm it is generating tickets to.

Chapter 5

Prototype Implementation

This chapter covers important details of the performed implementation, including the available tools used and the description of the most important classes, interfaces, modules, and so on.

As we described in Section 1.3, almost the whole Kerberos → X.509 conversion was already implemented when this thesis began. The rest of the functionality has been added to that available framework, both writing new code on existing classes and creating new ones.

The **implemented functionalities** were the Kerberos → SAML conversion described in Section 3.1, and the X.509 → Kerberos translation following the approach explained in Section 3.2.2. This includes messages signature for all conversions (the prototype already implemented allowed Kerberos signature for the client → STS direction, but not for the opposite one).

5.1 Implementation Environment and Tools

The entire source code for this project has been written in **Java** (*Java 2 Platform Standard Edition Development Kit 5.0* provided by SUN Microsystems[1]) on **GNU/Linux**, using **Eclipse**[2] as development environment.

To enhance and extend the functionalities of a "simple" Web server (like Apache Web Server) in a Java framework, Sun is developing two technologies: the *Java Servlet*[3] and *JavaServer Pages (JSP)*[4]. The Java Servlet provides a server- and platform-independent way of building Web based applications, giving a standard to be implemented in a Web server framework, which extends its functionality. Servlets have access to the Java API and HTTP calls. JSP is an extension of this servlet technology to easily and quickly develop and maintain Web based applications. Together, Java Servlet and JSP provide a platform-independent

[1] http://java.sun.com/javase/downloads/index_jdk5.jsp
[2] http://www.eclipse.org/
[3] http://java.sun.com/products/servlets
[4] http://java.sun.com/products/jsp

and easy alternative for building interactive Web applications. There are several alternative implementations of this technology available in the market. We chose the open implementation **Apache Tomcat**[5] (version 5.5.20) for our project since, as we will see, many other tools coming from the Apache foundation will be used.

Apache Axis[6] on its latest final version 1.4 is another Apache open source implementation, available in Java and C++, of the SOAP mechanism introduced in Section 2.6. It is a important tool for us, since the request/response messages exchanged with Web services, and therefore in the communication between client and STS, are SOAP messages. It will be installed inside Apache Tomcat. Both Axis and Tomcat can be installed on Windows and different UNIX "flavors".

Apache Tomcat and Apache Axis give us a platform and an API to build Web services, allowing the exchange of SOAP messages with clients. Still, we need to include the OASIS WS-Security specification in our project. This is given by Apache Web Services Security of Java (**WSS4J**)[7]. It is a library that basically enables signature, signature verification, and security token attachment to be used on SOAP messages. The latest version (1.5.0) implements the WSS SOAP message security 1.0 [1], Username Token profile 1.0 [17], and X.509 Token Profile 1.0 [21] specifications described in Section 2.7. It also includes support for sending and receiving WS-Trust RST and RSTR messages.

WSS4J will be used by Axis to process messages. With WSS4J and Axis, the signature/signature verification processes, and credential attachment into the SOAP header, will be made automatically for incoming and outgoing messages in a "transparent" way for the application. The Axis deployment descriptor files (.wsdd) in client and server side will provide all the information that Axis and WSS4J need to know about how to process messages. For example, it specifies which actions should be made with incoming and outgoing messages, e.g. signature, and which credentials and keys will be used to do so. Then, the Web service developer just needs to include the desired information in the deployment descriptor, and those actions will take place without the need of adding more code to the application.

WSS4J in turn uses **Apache XML Security**[8] for XML Signature and Encryption.

When dealing with SAML assertions, in the Kerberos → SAML implementation part, **OpenSAML**[9] v1.1 was the chosen library for creating and parsing SAML assertions. It is an open source implementation of the SAML v1.1 specification [10].

In short, we could say that our solution is a Java implementation which works over Tomcat+Axis+WSS4J running on GNU/Linux, with the help of other libraries like XML Security and OpenSAML among others.

As many libraries are used in the project, which in turn have other dependencies, it is difficult to build the project using only the Java compiler. Instead, we have

[5] http://tomcat.apache.org/
[6] http://ws.apache.org/axis/
[7] http://ws.apache.org/wss4j/
[8] http://xml.apache.org/security/
[9] http://www.opensaml.org/

used **Apache Maven**[10] v1.0.2 as project management tool. It provides different commands for automatically download the desired dependencies of our code, storing all libraries in a local repository. These libraries will be used when building the system. Maven will also be run to "install" the Web service inside Axis (copy the compiled code and configuration files, change the Axis descriptor `.wsdd`, etc).

Given the multi-platform nature of Java (there should be no problem in executing Java compiled code in two different platforms which have the same virtual machine version installed), and given that the servlet container is available for different platforms, our solution should be multi-platform, or at least not that many changes should be needed for that purpose. The principal problems of compatibility are given by the way we use to get the local Kerberos credentials. In GNU/Linux, the Kerberos *credentials cache* is a file in a temporal directory (typically something like `/tmp/krb5cc_1000`). How and where this information is stored could change in different systems. Besides this, our implementation should work on any platform if the same versions of the Java libraries and tools are used without any problem.

5.2 Main Implementation Tasks Performed

The implementation itself consists in two different projects: **kth-sts** containing the client and service parts of our framework, and **wss4j-sts** with the WSS4J 1.5 implementation with some changes.

As WSS4J is implementing the 1.0 version of WS-Security specification, it does not have support for Kerberos tokens (The Kerberos Token profile was introduced in version 1.1 [23]). The only binary security tokens that can be attached are X.509 certificates. Therefore, the main changes were added to support Kerberos binary token attachment in the SOAP `Security` header, and to perform Kerberos session key signature and signature verification for request and response messages. This is needed for the Kerberos → X.509/SAML conversions described in Section 3.1, as the RST and RSTR messages are signed with the key contained in the service ticket attached by the client (see Figure 3.1).

For the Client → STS direction, a Kerberos ticket must be attached, and the message signed with the corresponding session key. Then, the STS should be able to extract the ticket, decrypt it, get the session key, and verify the signature. In the opposite direction, the STS will just sign the message with the received session key, and the client will use that key for signature verification.

Other minor changes were made to WSS4J, to add some needed functionalities. The **kth-sts** project contains STS and client functionalities themselves. In short:

- The client will have the code for getting his credential from the IP, generating the needed data for the request, sending the signed RST with the attached information, waiting for the response message, getting the RSTR containing the new converted credential and additional information, verifying the signature, and storing the received data somewhere for future use.

[10] http://maven.apache.org/

- The STS will be waiting for client requests. When one message arrives, the signature is first verified by the WSS4J code. If valid, the STS will then look the kind of request (e.g. RST issuance). Then, for a issue security token message, it will distinct the kind of token requested (SAML, X.509, or Kerberos ticket). Each conversion will make the same steps: Get the mandatory information the client must have sent (credential to be translated, CR, additional keys, etc.), make the verifications in each case, extract the identity and extra information needed for generating the new credential, and build the new security token. Then, that token and perhaps more information (like the session key when generating a Kerberos ticket) is packet inside a RSTR message, which is signed with the key of each case.

- Both parts share some modules, for example utility modules to handle certificates, Kerberos tickets, extract and add info to SOAP messages, message handlers (which inspect the received message to see whether it is a RST or RSTR message, the kind of token, and so on), etc.

5.3 Main Problems Found

A problem which we had to face since the beginning of the thesis was the recent publication of many of the specifications used: Most of them are currently being developed, and new versions are continuously being released. Moreover, WS-Trust and WS-Federation are still not standards but drafts released for review and evaluation only. Sometimes, this made difficult to find information and tools for these specifications.

Focusing on implementation issues, we came up against a "simple" problem, which took a long time to be solved. It was caused by how Axis serializes and deserializes SOAP messages. When we create an assertion, it has the form: "`<saml:Assertion [...] <saml:Subject [...] </saml:Subject> [...] </saml:Assertion>`". Then, we sign that assertion with the STS' private key. However, before Axis signed and sent the RSTR message containing that assertion, it performed a namespace optimization which resulted in changing the assertion into: "`<Assertion [...] <Subject [...] </Subject> [...] </Assertion>`". That is, the `saml:` prefixes were removed. Then, when the client received the assertion and tried to verify its signature with the STS' public key, it failed since the digest of the assertion's text is different when removing the `saml:` parts (the text of the assertion has actually changed). It was a bit difficult to find out the origin the problem, which was solved by changing a parameter on the client and service descriptors to forbid this namespace optimization when serializing.

Another difficulty we found were in the generation of a Kerberos ticket on the X.509 → Kerberos conversion. The standard Java API does not provide the needed functionality for creating ticket fields and encrypt them. It only gives classes to treat tickets on the client's point of view: The client receives a TGT or ST encrypted with a key she does not have, and a session key which is also inside. Then, we had to use

internal Java libraries (`sun.security.krb5.*`), which had **no API specification**. The only information available were the classes' variables and methods' headers, as the source code was also not available. With this information and the Kerberos standard [3], the process of creating a valid ticket was quite difficult and slow at the beginning. It was a kind of "trial and error" implementation, trying different values when calling the methods until we got the desired result.

Chapter 6

Analysis of Results

This chapter enumerates the main tests we have executed on the prototype, with a short description of each one containing the functionalities that were intended to be tested in each case. The tests are based on the description of the implemented translations given in Chapter 3.

6.1 Local Tests Performed

The execution of these tests follows step-by-step the illustration and description given for the specific case in Chapter 3, depending on the conversion. Client and service are running on the same machine, so the messages are sent to and received from Apache Tomcat through the standard IP address used for a loopback network connection (typically `127.0.0.1` or `localhost`). However, the behaviour would be exactly the same as if we had client and STS running on separate machines. The only difference is that on the latter case the messages would travel through the network.

6.1.1 Kerberos $\Longrightarrow$ X.509 Test

1) We suppose that the client has already obtained a TGT valid for the STS' realm directly or through a cross-realm operation

2) The client application extracts the ticket from the local cache, and uses it for getting a ST valid in the STS.

3) With the Principal and Realm names specified when getting the ticket (in our case `etg` and `NADA.KTH.SE`), and some other information (organization=`Grid` and country=`SE`) available on client's configuration file, the application will create a CR. A key pair is generated, inserting the public part in the CR.

4) The client generates an issue RST message (Listing C.1), including the CR and the ST in the body. Lines 9–21 contain the ST in the header for signature verification. The signature element (Lines 22–45) indicates that the key used for

signing was the one inside the security token ("KrbID=6885751" in Line 41 is actually pointing to the ticket, which has that ID). The requested token is X.509 as appears in Line 54, and finally comes the CR (Lines 57–72) and the ST to be converted (Lines 73–84).

5) The STS receives the message and makes the corresponding verifications. If everything goes right, it creates the certificate from the certification request, inserting its Issuer DN (`C=SE ,ST= Stockholm ,O=KTH ,OU=PDC ,CN=STS`) and 12h. validity period from the current time, signing it with its private key.

6) The RSTR message is created (Listing C.2). Note that the message is signed with the session key again, but the ticket is not attached. Instead, the recipient will know that this key was used since the message is the response to the request message sent, as the `SignatureConfirmation` value on Line 33 is the signature value of the RST message (Line 35 of Listing C.1). The generated certificate is encapsulated inside a `RequestedSecurityToken` message (Lines 47–66).

7) The client receives the certificate (Listing C.3) and stores it in a local Key Store.

6.1.2 Kerberos $\Longrightarrow$ SAML Test

1) We suppose that the client has already obtained a TGT valid for the STS' realm directly or through a cross-realm operation.

2) The client application extracts the ticket from the local cache, and uses it for getting a ST valid in the STS.

3) The client generates an issue RST message (Listing C.4), including the ST in the body. A key pair is generated. Instead of the CR, a `KeyInfo` element with the needed information to construct the public key is inserted in the RST element (Lines 70–84). The rest of the message (signature, ticket attached, and so on) is the same as in the test explained in Section 6.1.1, changing the requested token from X.509 to SAML (Line 54).

4) The STS receives the message and makes the corresponding verifications. If everything goes right, it creates the assertion (Listing C.6) from the identity information inside the ticket (`etg` and `NADA.KTH.SE`, Line 18). The validity is set to 12h. from the current time (Lines 10 and 11), and the received public key is inserted as `SubjectConfirmation` (Lines 20–37), stating that the method is *holder of key* on Line 22. The assertion is signed with the STS' private key, including the certificate for signature verification (Lines 40–79). The STS inserts its NameID in the `Issuer` field (Line 4).

5) The RSTR message is created (Listing C.5), including the `Assertion` in the `RequestedSecurityToken` element (Lines 47–128). The message is signed as in the Kerberos $\rightarrow$ X.509 translation.

6) The client receives the assertion and stores it in a local plain text file, allowing any other application accessing it.

6.1.3 X.509 $\Longrightarrow$ Kerberos Test

1) We suppose that the client has a valid X.509 v3 certificate from a CA trusted by the STS, with the corresponding private key.

2) The client application extracts the certificate and private key from the local Key Store.

3) The client generates an issue RST message (Listing C.7). The signature element (Lines 9–38) indicates that the key used for the signature is one corresponding to the certificate with Issuer DN `''CN=Esteban, OU=NADA, O=KTH, C=SE''` and serial number `1171582785` (Lines 30–35). It supposes that the STS already has the client's certificate stored locally, or it is able to get it by other means. Then, the STS just needs to access its Key Store, and look for that certificate to verify the signature. If the STS would not have the certificate, the client had to attach it as security token. The requested token is Kerberos ticket as appear in Line 48. The certificate to be converted is included as a `Claim` (Lines 58–73). In order for the STS to know which realm the client wants to access to, the RST message includes the target domain `mehrana.nada.kth.se` in the `AppliesTo` element (Lines 50–55).

4) The STS receives the message and makes the corresponding verifications. If everything goes right, it creates the ticket (Listing C.9) for the specified realm with the client's information from the certificate and a randomly generated session key. The Principal `name` will be the CN from the ticket (`Esteban`), and the `realm` the STS' one (`PDC.SE`). The remote TGS principal is extracted from the STS' local database (in our example `sts/mehrana.nada.kth.se` instead of the usual `krbtgt/mehrana.nada.kth.se`), and the ticket is encrypted with the shared key between STS and remote TGS. The validity is 1 hour from the current time.

5) The RSTR message is created (Listing C.8). The message is signed with the STS' private key. Again, it only contains the Issuer name and Serial Number of the certificate (Lines 37–42), as the client has it stored locally as trusted. The ticket is returned as binary token inside the RSTR element (Lines 62–73), and the session key is included in a `RequestedProofToken`, encrypted with the client's public key (Lines 75–86).

6) The client receives the remote TGT, which can be used to be authenticated in the remote TGS in combination with the session key (decrypted with its private key).

6.2 Real Scenario Test

As stated in Section 1.3, most of the Kerberos ticket → X.509 certificate conversion was implemented before this thesis started. After that implementation part was finished, it was tested in a real Grid environment. The details of the test performed can be found in [31].

In short, the test scenario is as follows: An *animator* of a *producer* of high-quality video content is working on a high-definition video rendering job, which needs a large amount of calculations, for a customer. Given that the deadline to finish the work is too close, and the producer does not have the needed amount of computer power to meet it, the animator realizes that they need to get access to an existing Grid infrastructure that provides services for rendering high-definition video (the *service provider*). The problem is that this service provider requires users being authenticated via X.509 certificates. However, the operator uses Kerberos for local authentication, and does not have a relationship with a third party CA. The steps needed to get a certificate from a CA trusted by the service (e.g. Verisign) would take too long, and the deadline could not be met.

To solve this problem, the animator makes use of a STS which can be accessed by end users of the producer's Kerberos domain. The certificates issued by this STS are trusted by the Grid service. During the test, the system worked as expected, and the animator was able to get a certificate from the STS, and use it for requesting to the Grid the execution of the rendering job.

6.3 Analysis

As a conclusion of the performed tests, it is shown that the prototype met the goals listed in Section 1.4, given the limitations described in Section 1.6:

- Different conversions between the three formats we have used have been performed successfully.
- The generated credential is signed/encrypted by the STS to provide trust to the token (if there is a previous trust relationship between final resource and STS), assuring it was created by the STS. The validity of the new credential is short to limit security risks.
- The solution follows WS-Security, WS-Trust, and Kerberos standards as much as possible.
- The request and response messages are authenticated and their integrity is checked in each one.
- The sensitive information, like the session key, is encrypted during the communication.

Chapter 7

Conclusions

In this part we summarize our contribution, pointing the most important results obtained. It also proposes the next steps that should be performed in the project.

7.1 Summary of Contributions

As we have seen in the introductory part of this report, the heterogeneous nature of the Grid makes it very difficult for a client to know which security credentials would be needed beforehand. Even if that client knows which realms she will need to be authenticated in, she might not have a valid token to be used in some remote domains (e.g., a Kerberos ticket for a certain realm).

We have studied the emerging standards to perform credential translation in a *platform-independent* way. The client and the translator applications could be written in different programming languages and could run on different architectures in a Grid environment. The study was focused on three commonly used security credentials: *Kerberos tickets*, *SAML Assertions*, and *X.509 Certificates*.

The study concluded that the most suitable technologies for our case were Web services. Following *WS-Security* and *WS-Trust* we designed and implemented a STS able to make credential translation when some **trust** properties are kept beforehand. Some alternatives were compared during the design, trying to reach a solution with the highest level of *security* and with the least changes needed on other components involved in the system (IP, SP, and resource itself). As the request and response messages are XML formatted, client and service just need to agree on the included information needed in each case, but the internal architecture or implementation of each side is not fixed.

The implemented prototype was tested against the goals and design specification made, obtaining the expected results: A client was able to get a credential valid in another realm she could not access to before.

7.2 Future Work

There are a few aspects of the system which we have not studied in depth and which can be interesting to address on the next steps of our project:

- Design and implementation of the **SAML assertion → Kerberos ticket** conversion. It should be similar to the X.509 → Kerberos translation. The most difficult issue could be the establishment of trust between STS and client, since the client does not have a certificate on this case.

- **SAML assertion ↔ X.509 certificate** conversions. Design of the translation for each direction. The mapping of credentials seems to be easy, but the trust relationship between client and STS should be investigated to find the most flexible way.

- Further investigation of the integration with **Shibboleth** should be performed when more information and new versions of the Shibboleth system are available. The feasibility of the model proposed on this report should be studied, modifying it or proposing alternative ways after carefully read the new information.

- Now, the parameters of client and service (e.g., path to the key stores and certificates, Kerberos parameters, etc.) are specified in properties files on each side. A more intuitive way of doing it, adding a GUI for the service installation, could be interesting.

- We have only addressed identity issues. The STS should be able to issue **attribute assertions** about a client given her identity. For example, a client could present her Kerberos ticket or X.509 certificate and the STS, after checking the client's identity and its policy, could issue certain attributes depending on some roles or groups the client is into (e.g. "student").

- Our solution is not using functionalities given by WS-Federation. The application of that standard for **Attributes/Name-space mapping** would be necessary in a real environment. The recent *WS-Federation v1.1* released in December 2006 should be studied.

- It would be good to be able to establish **dynamic trust relationships** between STS/Client and STS/Resource. This would make our solution more flexible.

- The solution should be able to execute on **different platforms** with the same Java virtual machine and libraries. A study of the needed changes, and a set of tests to prove that it can be executed in other operating systems and architectures should be carried out.

In summary, the final goal of our project is to construct a complete system based on WS-Trust and WS-Federation specifications. In that system, illustrated in Figure 2.5, the client will have a local IP/STS, that gives him a credential valid for another IP/STS in the resource's realm (both STS trust each other). The latter IP/STS will provide the requestor with a temporal credential valid for the resource. This would solve the trust issues between the STS and the other two parties. As there is one local STS at each realm (so trust can be easily established by the local administrator), just one trust relationship between both STS is needed.

Appendix A

Glossary

These are some terms which appear during the whole document. It is necessary for the audience to be familiar with them when reading the paper. Most of them are specific to the security area, quoted from the security literature.

Access Control: *"Protection of system resources against unauthorized access; a process by which use of system resources is regulated according to a security policy and is permitted by only authorized entities (users, programs, processes, or other systems) according to that policy"* [5].

Asymmetric cryptography is a *"branch of cryptography (popularly known as "public key cryptography") in which the algorithms employ a pair of keys (a public key and a private key) and use a different component of the pair for different steps of the algorithm [such as encryption and decryption, or signature creation and signature verification]"* [5]. Common asymmetric key cryptography algorithms are `RSA` and `Diffie-Hellman`.

Authentication is a way to prove that an entity is who it says. Formally, it *"is the process by which a subject proves its identity to a requestor, typically through the use of a credential. Authentication in which both parties (i.e., the requestor and the requestee) authenticate themselves to one another simultaneously is referred to as mutual authentication"* [32].

Authorization *"is the process by which we determine whether a subject is allowed to access or use an object"* [32]. It states what the entity is allowed to do. It may be specified in a **security policy**.

A **Claim** *"is a declaration made by an entity (e.g. name, identity, key, group, privilege, capability, attribute, etc.)"* [26].

A **Client** is *"a system entity that requests and uses a service provided by another system entity, called a **server**"* [5].

Confidentiality is *"the property that information is not made available or disclosed to unauthorized individuals, entities, or processes"* [5].

A **Credential** *"is a piece of information that is used to prove the identity of a subject"* [32]. It is a token that gives the entity a set of claims (its identity, some attributes, ...) and a way to check that it and only it (ideally) could be the holder (e.g., a private secret). The credential must be trusted by the party the entity is authenticating to. Examples of credentials are passwords, Kerberos tickets and X.509 certificates. In this document, I will use the terms **Security Token** and credential referring the same concept.

Cryptography is *"the mathematical science that deals with transforming data to render its meaning unintelligible (i.e., to hide its semantic content), prevent its undetected alteration, or prevent its unauthorized use. If the transformation is reversible, cryptography also deals with restoring encrypted data to intelligible form"* [5].

A **Digital Signature** is *"a value computed with a cryptographic algorithm and appended to a data object in such a way that any recipient of the data can use the signature to verify the data's origin and integrity"* [5]. A typical usage of the digital signature is encrypting with a private key the hash value (see below) of a message, so that it can be verified with the public key.

Encryption is the *"cryptographic transformation of data (called "plaintext") into a form (called "ciphertext") that conceals the data's original meaning to prevent it from being known or used. If the transformation is reversible, the corresponding reversal process is called* ***decryption****, which is a transformation that restores encrypted data to its original state"* [5]. Common encryption algorithms include `DES`, `Triple DES`, `AES`, and `RSA`.

A **Hash Function** is *"an algorithm that computes a value based on a data object (such as a message or file; usually variable-length; possibly very large), thereby mapping the data object to a smaller data object (the "hash result") which is usually a fixed-size value"* [5]. The term **Digest Function** will be used having the same meaning. A secure hash function must have two properties: It is not computationally feasible to find the original data object given its hash result, neither to find two data objects with the same hash result (finding "collisions"). Hash result, **Hash Value**, and **Digest Value** will also be treated as synonyms during the rest of the thesis. The two most-commonly used hash functions are `MD5` and `SHA-1`.

Identification is *"an act or process that presents an identifier to a system so that the system can recognize a system entity and distinguish it from other entities"*. Do not confuse with authentication. Here, the entity just presents its identity. The authentication process *proves* that this identity corresponds to the entity.

Identity: Who an entity is.

Integrity is *"the property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner"* [5].

A **Key** is *"an input parameter that varies the transformation performed by a cryptographic algorithm"* [5].

An **Object** *"is a resource that is being protected by the security policy"* [32].

Privacy is *"the right of individuals to control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed"* [5].

Repudiation: *"Denial by a system entity that was involved in an association (especially an association that transfers information) of having participated in the relationship"* [5].

A **Security Policy** is *"a set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources"* [5]. The term **Policy** will be used as an abbreviation for this concept.

Single Sign-On (SSO) is *"A system that enables a user to access multiple computer platforms (usually a set of hosts on the same network) or application systems after being authenticated just one time"* [5].

A **Subject** *"is a participant in a security operation. In grid systems, a subject is generally a user, a process operating on behalf of a user, a resource (such as a computer or a file), or a process acting on behalf of a resource"* [32].

Symmetric cryptography is *"A branch of cryptography involving algorithms that use the same key for two different steps of the algorithm (such as encryption and decryption, or signature creation and signature verification)"* [5]. Common symmetric key cryptography algorithms are `DES`, `Triple DES`, `AES`.

Trust *"is the characteristic that one entity is willing to rely upon a second entity to execute a set of actions and/or to make set of assertions about a set of subjects and/or scopes"* [25].

A **Trust Domain/Realm** *"is a logical, administrative structure within which a single, consistent local security policy holds. Put another way, a trust domain is a collection of both subjects and objects governed by single administration and a single security policy"* [32].

Appendix B

Abbreviations

AS	Authentication Server, see Section 2.2
CA	Certification Authority, see Section 2.3
CN	Common Name, a field of a Distinguished Name
CR	Certification Request, see Section 2.3
DN	Distinguished Name
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol or Identity Provider
IPsec	Internet Protocol Security
KCA	Kerberized Certificate Authority, see Section 4.2
KCT	Kerberized Credential Translator, see Section 4.3
KDC	Key Distribution Center, see Section 2.2
PKI	Public Key Infrastructure, see Section 2.3
REL	Rights Expression Language
RST	Request Security Token, see Section 2.7.2
RSTR	Request Security Token Response, see Section 2.7.2
SAML	Security Assertion Markup Language, see Section 2.5
SOAP	originally Simple Object Access Protocol, now it is not used as an acronym, although it is still capitalized
SP	Service Provider
SSL	Secure Sockets Layer
SSO	Single Sign-On
ST	Service Ticket, see Section 2.2
STS	Security Token Service
TGS	Ticket Granting Server, see Section 2.2
TGT	Ticket Granting Ticket, see Section 2.2
TLS	Transport Layer Security
URI	Uniform Resource Identifier
VO	Virtual Organization
WS	Web Service, see Section 2.6
WSDL	Web Services Description Language, see Section 2.6

WSS Web Services Security 2.7
XML Extensible Markup Language

Appendix C

Messages exchanged during Tests

C.1 Kerberos $\Longrightarrow$ X.509 Test

C.1.1 RST Message

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
  <wsse:Security
   soapenv:mustUnderstand="1"
   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-wssecurity-secext-1.0.xsd">
   <wsse:BinarySecurityToken
    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
        -200401-wss-soap-message-security-1.0#Base64Binary"
    ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
        token-profile-1.1#GSS_Kerberosv5_AP_REQ"
    wsu:Id="KrbID-6885751"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
        wss-wssecurity-utility-1.0.xsd">
    YYH8MIH5oAMCAQWhDRsLTkFEQS5LVEguU0WiJTAjoAMCAQGhHDAaGwNz
    dHMbE21laHJhbmEubmFkYS5rdGguc2WjgbswgbigAwIBA6EDAgEBooGr
    BIGoTx0GlQrRrMKro/S9KSWlBmsiIhK3dhaw0xf2+QM+AOFRgW0hWY5g
    WIXHoqZHIdN1OK9hApYYMcTdnlNsnPZTiT0YOnGtAYvYFsKaSOewQLh5
    Lj0altKmM9QYPl+miNyAkbUrLOeuOqtXM8uzVCGzC7P4D1TlAeCzkGbH
    pOpcAe+rxKMcDPJuZ4XaeTH/RwYHLqcsLH9gm3VLQZ4yUYmcWl0r/8Ws
    cttn
   </wsse:BinarySecurityToken>
   <ds:Signature Id="Signature-5076660"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
     <ds:CanonicalizationMethod Algorithm="http://www.w3.org
         /2001/10/xml-exc-c14n#"/>
     <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
         xmldsig#hmac-sha1"/>
     <ds:Reference URI="#id-28623319">
```

```
28        <ds:Transforms>
29         <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
               c14n#"/>
30        </ds:Transforms>
31        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
              #sha1"/>
32        <ds:DigestValue>qy+xc3zmJXg/uD5W4Yu8x/xKL0w=</ds:DigestValue>
33       </ds:Reference>
34      </ds:SignedInfo>
35      <ds:SignatureValue>P7cg0LmLDb85oBdflf9dRZEFxUs=</
           ds:SignatureValue>
36      <ds:KeyInfo Id="KeyId-9144903">
37       <wsse:SecurityTokenReference
38        wsu:Id="STRId-12470752"
39        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis
              -200401-wss-wssecurity-utility-1.0.xsd">
40        <wsse:Reference
41         URI="#KrbID-6885751"
42         ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos
              -token-profile-1.1#GSS_Kerberosv5_AP_REQ"/>
43       </wsse:SecurityTokenReference>
44      </ds:KeyInfo>
45     </ds:Signature>
46    </wsse:Security>
47  </soapenv:Header>
48  <soapenv:Body
49   wsu:Id="id-28623319"
50   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
         wss-wssecurity-utility-1.0.xsd">
51   <wst:RequestSecurityToken xmlns:wst="http://schemas.xmlsoap.org/
         ws/2005/02/trust">
52    <wst:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/
           Issue</wst:RequestType>
53    <wst:TokenType>
54     http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-
           token-profile-1.0
55    </wst:TokenType>
56    <wst:Claims Dialect="http://DialectNameSpace">
57     <wsse:BinarySecurityToken
58      EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-soap-message-security-1.0#Base64Binary"
59      ValueType="http://www.pdc.kth.se/oasis-200401-wss-x509-token-
           profile-1.0#PKCS10"
60      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-wssecurity-secext-1.0.xsd">
61      MIIBpDCCAQ0CAQAwZDENMAsGA1UEChMER3JpZDEeMBwGCSqGSIb3DQE
62      JARYPZXRnQE5BREEuS1RILlNFMQ8wDQYDVQQGEwZTd2VkZW4xDDAKBg
63      NVBAMTA2V0ZzEUMBIGA1UECxMLTkFEQS5LVEguU0UwgZ8wDQYJKoZIh
64      vcNAQEBBQADgY0AMIGJAoGBAKxPwuWoa3XueBurAntwFPeZiUcJ52FY
65      akNIIXSRIwepFLt184Zc4VFgj7tnmX8DercVZCX1LaQ0RpoKGhbi0o0
66      SA7HCLpM52VpBzDE/TAyqlj5L0PXz0wnTH7OWgd70TcGWXUoRZamW9t
67      OBbC0oeZqqQ/VUvvli+Z7NoMdD0R8VAgMBAAGgADANBgkqhkiG9w0BA
68      QUFAAOBgQBRh75gfvYcRd47YJnfPFUOYb2jVWUjzHnmDBvXMJEF3GMD
69      lV+KKrx5orzVNUROQd0EW+b0bSFtCIBOFUbcMFpCnZxiC5KCe688wWz
```

```
70       XxJzEKxKfVGfezc0V5x22seXQybd/Yw/SOBpXw/yGx3Z9OrCCFJ+Qsq
71       KFVKxzV8zQdny3Rg==
72      </wsse:BinarySecurityToken>
73      <wsse:BinarySecurityToken
74       EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-soap-message-security-1.0#Base64Binary"
75       ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
             token-profile-1.1#GSS_Kerberosv5_AP_REQ"
76       xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-wssecurity-secext-1.0.xsd">
77       YYH8MIH5oAMCAQWhDRsLTkFEQS5LVEguU0WiJTAjoAMCAQGhHDAaGwN
78       zdHMbE21laHJhbmEubmFkYS5rdGguc2WjgbswgbigAwIBA6EDAgEBoo
79       GrBIGopDm7M0rW4NckB+5ulnWfSEplXRB2ZX9ekODoXeJPnaT2TWcW9
80       6hSC0s+WOy65sz/IIFSez/FqjaSEkMEQQ4lzxudC5s9nkvIAJU2SFf5
81       Di6OItctO97idRDSdDXEi/tcqZoKW66yhE4aM2yqbSC0fcMXeTC2dCh
82       iGH55a38+g4xw9y9jxQnRFJYjPs7hGpK2ryNSAehpGuEaSqRrl0RYFX
83       9tD1/5vA57
84      </wsse:BinarySecurityToken>
85     </wst:Claims>
86    </wst:RequestSecurityToken>
87   </soapenv:Body>
88 </soapenv:Envelope>
```

Listing C.1. RST Message in Kerberos → X.509 translation

C.1.2 RSTR Message

```
1  <soapenv:Envelope
2   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <soapenv:Header>
6    <wsse:Security
7     soapenv:mustUnderstand="1"
8     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-wssecurity-secext-1.0.xsd">
9     <ds:Signature
10     Id="Signature-6059828"
11     xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
12     <ds:SignedInfo>
13      <ds:CanonicalizationMethod Algorithm="http://www.w3.org
            /2001/10/xml-exc-c14n#"/>
14      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
            xmldsig#hmac-sha1"/>
15      <ds:Reference URI="#id-16765237">
16       <ds:Transforms>
17        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
              c14n#"/>
18       </ds:Transforms>
19       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
             #sha1"/>
20       <ds:DigestValue>M5+nbOa1ZkVlFr3P1CLB5/NlJGw=</ds:DigestValue>
21      </ds:Reference>
22      <ds:Reference URI="#SigConf-27785692">
```

```
23        <ds:Transforms>
24         <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
               c14n#"/>
25        </ds:Transforms>
26        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
              #sha1"/>
27        <ds:DigestValue>lpP2608mk4yNTw8AnydxvYpTXAc=</ds:DigestValue>
28       </ds:Reference>
29      </ds:SignedInfo>
30      <ds:SignatureValue>m8/h0LEoGwyHsZ6G+ZpmJMv8TPk=</
            ds:SignatureValue>
31     </ds:Signature>
32     <wsse11:SignatureConfirmation
33      Value="P7cg0LmLDb85oBdflf9dRZEFxUs="
34      wsu:Id="SigConf-27785692"
35      xmlns:wsse11="http://docs.oasis-open.org/wss/2005/xx/oasis-2005
            xx-wss-wssecurity-secext-1.1.xsd"
36      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
            wss-wssecurity-utility-1.0.xsd"/>
37   </wsse:Security>
38  </soapenv:Header>
39  <soapenv:Body
40   wsu:Id="id-16765237"
41   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
        wss-wssecurity-utility-1.0.xsd">
42   <wst:RequestSecurityTokenResponse
43    xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
44    <wst:TokenType>
45     http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-
           token-profile-1.0
46    </wst:TokenType>
47    <wst:RequestedSecurityToken>
48     <wsse:BinarySecurityToken
49      EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-soap-message-security-1.0#Base64Binary"
50      ValueType="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-x509-token-profile-1.0#X509v3"
51      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-wssecurity-secext-1.0.xsd">
52      MIIB6DCCAZKgAwIBAgIDEAABMA0GCSqGSIb3DQEBBAUAME8xCzAJBg
53      NVBAYTAlNFMRIwEAYDVQQIEwlTdG9ja2hvbG0xDDAKBgNVBAoTA0tU
54      SDEMMAoGA1UECxMDUERDMRAwDgYDVQQDEwdFc3RlYmFuMB4XDTA3MD
55      IxNTE5MDAxMloXDTA3MDIxNjA3MDUxMlowZDENMAsGA1UEChMER3Jp
56      ZDEeMBwGCSqGSIb3DQEJARYPZXRnQE5BREEuS1RILlNFMQ8wDQYDVQ
57      QGEwZTd2VkZW4xDDAKBgNVBAMTA2V0ZzEUMBIGA1UECxMLTkFEQS5L
58      VEguU0UwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAKxPwuWoa3
59      XueBurAntwFPeZiUcJ52FYakNIIXSRIwepFLt184Zc4VFgj7tnmX8D
60      ercVZCX1LaQ0RpoKGhbi0o0SA7HCLpM52VpBzDE/TAyqlj5L0PXz0w
61      nTH7OWgd70TcGWXUoRZamW9tOBbC0oeZqqQ/VUvvli+Z7NoMdD0R8V
62      AgMBAAEwDQYJKoZIhvcNAQEEBQADQQBdmH2m++bmWEQGLEuyGBQmtv
63      DPfNFCeyQumq+plW0a7/7ggv3dMAP66Fm6inSrJHZbSXkJQqlZiq7U
64      MB7shK7b
65     </wsse:BinarySecurityToken>
66    </wst:RequestedSecurityToken>
```

```
67   </wst:RequestSecurityTokenResponse>
68  </soapenv:Body>
69 </soapenv:Envelope>
```

Listing C.2. RSTR Message in Kerberos → X.509 translation

C.1.3 Returned Certificate

```
1 Version: 3
2  SerialNumber: 1048577
3  IssuerDN: C=SE,ST=Stockholm,O=KTH,OU=PDC,CN=STS
4  Start Date: Thu Feb 15 20:00:12 CET 2007
5  Final Date: Fri Feb 16 08:05:12 CET 2007
6  SubjectDN: O=Grid,E=etg@NADA.KTH.SE,C=SE,CN=etg,OU=NADA.KTH.SE
7  Public Key: RSA Public Key
8   modulus: ac4fc2e5a86b75ee781bab027b7014f799894709e761586a4
9            3482174912307a914bb75f3865ce151608fbb67997f037ab7
10           156425f52da434469a0a1a16e2d28d1203b1c22e9339d95a4
11           1cc313f4c0caa963e4bd0f5f3d309d31fb39681def44dc196
12           5d4a1165a996f6d3816c2d28799aaa43f554bef962f99ecda
13           0c743d11f15
14   public exponent: 10001
15  Signature Algorithm: MD5WithRSAEncryption
16          Signature: 5d987da6fbe6e65844062c4bb2181426b6f0cf7c
17                     d1427b242e9aafa9956d1aeffee082fddd3003fa
18                     e859ba8a74ab24765b49790942a9598aaed4301e
19                     ec84aedb
```

Listing C.3. Returned certificate in Kerberos → X.509 translation

C.2 Kerberos ⟹ SAML Test

C.2.1 RST Message

```
1 <soapenv:Envelope
2  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5  <soapenv:Header>
6   <wsse:Security
7    soapenv:mustUnderstand="1"
8    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
         wss-wssecurity-secext-1.0.xsd">
9    <wsse:BinarySecurityToken
10    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
          -200401-wss-soap-message-security-1.0#Base64Binary"
11    ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
          token-profile-1.1#GSS_Kerberosv5_AP_REQ"
12    wsu:Id="KrbID-20348456"
13    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-wssecurity-utility-1.0.xsd">
14    YYH8MIH5oAMCAQWhDRsLTkFEQS5LVEguU0WiJTAjoAMCAQGhHDAaGwNz
15    dHMbE21laHJhbmEubmFkYS5rdGguc2WjgbswgbigAwIBA6EDAgEBooGr
16    BIGoNbPZAFDRqTu6qXcUTkM48Ln1OFCqlXEutE8GxjR7+JLq7Vmd0hgE
```

```
17      RMPDH1oIvDb5dqQ3kmCMtLXd2gq4f8gElw2ubSnTKvM8UwrSQTht6esF
18      ZPDwtTyEyVT7pI+srm+lG5alpJhsl4Pn2JjZfji+jMcYvds1U3kEmjOl
19      DxYIHvtRXy45adGGhNWZ2hSDf42055OPYna4kFetsTzRmSgXv0HQU03p
20      eRmv
21     </wsse:BinarySecurityToken>
22     <ds:Signature Id="Signature-25865024" xmlns:ds="http://www.w3.
           org/2000/09/xmldsig#">
23      <ds:SignedInfo>
24       <ds:CanonicalizationMethod Algorithm="http://www.w3.org
             /2001/10/xml-exc-c14n#"/>
25       <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
             xmldsig#hmac-sha1"/>
26       <ds:Reference URI="#id-26870275">
27        <ds:Transforms>
28         <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
               c14n#"/>
29        </ds:Transforms>
30        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
             #sha1"/>
31        <ds:DigestValue>g+RyF4xHeQXcC1Gw0c445+eZP0s=</ds:DigestValue>
32       </ds:Reference>
33      </ds:SignedInfo>
34      <ds:SignatureValue>x7A//wuYkyzLKVsK7DK/XrjpQMg=</
           ds:SignatureValue>
35      <ds:KeyInfo Id="KeyId-18135083">
36       <wsse:SecurityTokenReference
37        wsu:Id="STRId-876215"
38        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-wssecurity-utility-1.0.xsd">
39        <wsse:Reference
40         URI="#KrbID-20348456"
41         ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos
               -token-profile-1.1#GSS_Kerberosv5_AP_REQ"/>
42       </wsse:SecurityTokenReference>
43      </ds:KeyInfo>
44     </ds:Signature>
45    </wsse:Security>
46   </soapenv:Header>
47   <soapenv:Body
48    wsu:Id="id-26870275"
49    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
         wss-wssecurity-utility-1.0.xsd">
50    <wst:RequestSecurityToken
51     xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
52     <wst:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/
           Issue</wst:RequestType>
53     <wst:TokenType>
54      http://docs.oasis-open.org/wss/2004/XX/oasis-2004XX-wss-saml-
           token-profile-1.0
55     </wst:TokenType>
56     <wst:Claims
57      Dialect="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
           token-profile-1.1#GSS_Kerberosv5_AP_REQ">
58      <wsse:BinarySecurityToken
```

```
59      EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
            -200401-wss-soap-message-security-1.0#Base64Binary"
60      ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
            token-profile-1.1#GSS_Kerberosv5_AP_REQ"
61      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
            -200401-wss-wssecurity-secext-1.0.xsd">
62      YYH8MIH5oAMCAQWhDRsLTkFEQS5LVEguU0WiJTAjoAMCAQGhHDAaGwN
63      zdHMbE21laHJhbmEubmFkYS5rdGguc2WjgbswgbigAwIBA6EDAgEBoo
64      GrBIGon5eLOthbiKkvNJdXHAkdV9u6H5nIwQ/eJL4AzhZMcpstbcA+z
65      4sL5yh4DuW1I5WMcgPloGMZsehlW1XARNh8NKB8T5yg8BGkpkbXZmau
66      QkjJSVzd37WQ+v1wJ9s7g/ruXw/plRVFO3SJBe4vJZ2OHq+UDgwKz1k
67      ogL/f2RmO1nCNPOmHmhzBuacRQl6jmldSyoJaOw7eACAZEOu6upn9ze
68      mVYjA3Ofpk
69     </wsse:BinarySecurityToken>
70     <ns1:KeyInfo
71      xmlns="http://www.w3.org/2000/09/xmldsig#"
72      xmlns:ns1="http://www.w3.org/2000/09/xmldsig#">
73      <ns1:KeyValue>
74       <ns1:RSAKeyValue>
75        <ns1:Modulus>
76         ofkJg9hyozd04bgKQiKW0/VbioNXEfgdewPNJAjsOLf/k6K8Ux+J
77         25JgOeuxUD2W3gbrDvg8itPzE0/eTiUK17FLzIQ9zGW3D4FInB03
78         8JxmD4qJJULgM7M61XN9Tw7TYJchMws44C9oZIgsuEgUyW1YQlxT
79         ALZAW4SroM6D/WU=
80        </ns1:Modulus>
81        <ns1:Exponent>AQAB</ns1:Exponent>
82       </ns1:RSAKeyValue>
83      </ns1:KeyValue>
84     </ns1:KeyInfo>
85    </wst:Claims>
86   </wst:RequestSecurityToken>
87  </soapenv:Body>
88 </soapenv:Envelope>
```

Listing C.4. RST Message in Kerberos → SAML translation

C.2.2 RSTR Message

```
1  <soapenv:Envelope
2   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <soapenv:Header>
6    <wsse:Security
7     soapenv:mustUnderstand="1"
8     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-wssecurity-secext-1.0.xsd">
9     <ds:Signature
10     Id="Signature-10923886"
11     xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
12     <ds:SignedInfo>
13      <ds:CanonicalizationMethod Algorithm="http://www.w3.org
            /2001/10/xml-exc-c14n#"/>
```

```
14      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
            xmldsig#hmac-sha1"/>
15      <ds:Reference URI="#id-21465645">
16       <ds:Transforms>
17        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
              c14n#"/>
18       </ds:Transforms>
19       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
             #sha1"/>
20       <ds:DigestValue>ipBF5sk/PlQCBH2vRmfVqbdNKZU=</ds:DigestValue>
21      </ds:Reference>
22      <ds:Reference URI="#SigConf-8585506">
23       <ds:Transforms>
24        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
              c14n#"/>
25       </ds:Transforms>
26       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
             #sha1"/>
27       <ds:DigestValue>zEW4UK1tJ17m0pM58f7FVuorhRk=</ds:DigestValue>
28      </ds:Reference>
29     </ds:SignedInfo>
30     <ds:SignatureValue>i2hKnuOvH0FxNWt6bI2x29dm9Kk=</
           ds:SignatureValue>
31    </ds:Signature>
32    <wsse11:SignatureConfirmation
33     Value="x7A//wuYkyzLKVsK7DK/XrjpQMg="
34     wsu:Id="SigConf-8585506"
35     xmlns:wsse11="http://docs.oasis-open.org/wss/2005/xx/oasis-2005
           xx-wss-wssecurity-secext-1.1.xsd"
36     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
           wss-wssecurity-utility-1.0.xsd"/>
37   </wsse:Security>
38 </soapenv:Header>
39 <soapenv:Body
40  wsu:Id="id-21465645"
41  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-wssecurity-utility-1.0.xsd">
42  <wst:RequestSecurityTokenResponse
43   xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
44   <wst:TokenType>
45    http://docs.oasis-open.org/wss/2004/XX/oasis-2004XX-wss-saml-
          token-profile-1.0
46   </wst:TokenType>
47   <wst:RequestedSecurityToken>
48    <saml:Assertion
49     AssertionID="_bac2462fd101f41567cabda33dbda1f6"
50     IssueInstant="2007-02-15T19:08:00.255Z"
51     Issuer="http://localhost:8080/axis/services/sts"
52     MajorVersion="1"
53     MinorVersion="1"
54     xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
55     xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">
56     <saml:Conditions
57      NotBefore="2007-02-15T19:08:00.255Z"
```

```
58        NotOnOrAfter="2007-02-16T07:13:00.255Z"/>
59       <saml:AuthenticationStatement
60        AuthenticationInstant="2007-02-15T19:08:00.255Z"
61        AuthenticationMethod="urn:ietf:rfc:1510">
62        <saml:Subject>
63         <saml:NameIdentifier
64          Format="urn:oasis:names:tc:SAML:1.1:nameid-
              format:unspecified">
65          etg@NADA.KTH.SE
66         </saml:NameIdentifier>
67         <saml:SubjectConfirmation>
68          <saml:ConfirmationMethod>
69           urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
70          </saml:ConfirmationMethod>
71          <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
72           <ds:KeyValue>
73            <ds:RSAKeyValue>
74             <ds:Modulus>
75              ofkJg9hyozd04bgKQiKW0/VbioNXEfgdewPNJAjsOLf/k6K8
76              Ux+J25JgOeuxUD2W3gbrDvg8itPzE0/eTiUK17FLzIQ9zGW3
77              D4FInB038JxmD4qJJULgM7M61XN9Tw7TYJchMws44C9oZIgs
78              uEgUyW1YQlxTALZAW4SroM6D/WU=
79             </ds:Modulus>
80             <ds:Exponent>AQAB</ds:Exponent>
81            </ds:RSAKeyValue>
82           </ds:KeyValue>
83          </ds:KeyInfo>
84         </saml:SubjectConfirmation>
85        </saml:Subject>
86       </saml:AuthenticationStatement>
87       <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
88        <ds:SignedInfo>
89         <ds:CanonicalizationMethod Algorithm="http://www.w3.org
              /2001/10/xml-exc-c14n#"/>
90         <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
              xmldsig#rsa-sha1"/>
91         <ds:Reference URI="#_bac2462fd101f41567cabda33dbda1f6">
92          <ds:Transforms>
93           <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig
                #enveloped-signature"/>
94           <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc
                -c14n#">
95            <ec:InclusiveNamespaces
96             PrefixList="code ds kind rw saml samlp typens #default
                  xsd xsi"
97             xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
98           </ds:Transform>
99          </ds:Transforms>
100         <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/
              xmldsig#sha1"/>
101         <ds:DigestValue>mGlM/PFNp5y4CsnF5Sn8vvTKy/I=</
              ds:DigestValue>
102        </ds:Reference>
103       </ds:SignedInfo>
```

```
104       <ds:SignatureValue>
105        xcUhn/vBUuzvNkLjEIWd9ZRurll3rFGcvGlgR1O31cyFtst1XlZLX
106        Oebm4bHBHfDkVJrUAk+L7JPf9cxbOhnYQ==
107       </ds:SignatureValue>
108       <ds:KeyInfo>
109        <ds:X509Data>
110         <ds:X509Certificate>
111          MIIBrTCCAVcCAxAAATANBgkqhkiG9w0BAQQFADByMQwwCgYDVQQ
112          KEwNLVEgxDDAKBgNVBAsTA1BEQzESMBAGA1UEBxMJU3RvY2tob2
113          xtMRIwEAYDVQQIEwlTdG9ja2hvbG0xCzAJBgNVBAYTAlNFMR8wH
114          QYDVQQDExZDZXJ0aWZpY2F0aW9uQXV0aG9yaXR5MB4XDTA2MTAw
115          NTEzMzgxM1oXDTA3MTAwNTEzMzgxM1owTzELMAkGA1UEBhMCU0U
116          xEjAQBgNVBAgTCVN0b2NraG9sbTEMMAoGA1UEChMDS1RIMQwwCg
117          YDVQQLEwNQREMxEDAOBgNVBAMTB0VzdGViYW4wXDANBgkqhkiG9
118          w0BAQEFAANLADBIAkEA0iheU3hQcaYHq4YrtVIYfyJnoNJKNBmb
119          TAkfV6CVOq7g10WBmxLzPjLwLuR7oF8vw9e9dv9QV1La+ubd6o5
120          llwIDAQABMA0GCSqGSIb3DQEBBAUAA0EAEaWtYqjankvren2DmA
121          QuOmB/fsc71T84FbgcoATFvVYB9SHfy2A1u9cMT7zCOEXkTg6RL
122          Tt2yac6SB/QaqXkWw==
123         </ds:X509Certificate>
124        </ds:X509Data>
125       </ds:KeyInfo>
126      </ds:Signature>
127     </saml:Assertion>
128    </wst:RequestedSecurityToken>
129   </wst:RequestSecurityTokenResponse>
130  </soapenv:Body>
131 </soapenv:Envelope>
```

Listing C.5. RSTR Message in Kerberos → SAML translation

C.2.3 Returned Assertion

```
1  <saml:Assertion
2   AssertionID="_bac2462fd101f41567cabda33dbda1f6"
3   IssueInstant="2007-02-15T19:08:00.255Z"
4   Issuer="http://localhost:8080/axis/services/sts"
5   MajorVersion="1"
6   MinorVersion="1"
7   xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
8   xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">
9   <saml:Conditions
10   NotBefore="2007-02-15T19:08:00.255Z"
11   NotOnOrAfter="2007-02-16T07:13:00.255Z"/>
12  <saml:AuthenticationStatement
13   AuthenticationInstant="2007-02-15T19:08:00.255Z"
14   AuthenticationMethod="urn:ietf:rfc:1510">
15   <saml:Subject>
16    <saml:NameIdentifier
17     Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">
18     etg@NADA.KTH.SE
19    </saml:NameIdentifier>
20    <saml:SubjectConfirmation>
21     <saml:ConfirmationMethod>
```

```
22      urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
23     </saml:ConfirmationMethod>
24     <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
25      <ds:KeyValue>
26       <ds:RSAKeyValue>
27        <ds:Modulus>
28         ofkJg9hyozd04bgKQiKW0/VbioNXEfgdewPNJAjsOLf/k6K8
29         Ux+J25JgOeuxUD2W3gbrDvg8itPzE0/eTiUK17FLzIQ9zGW3
30         D4FInB038JxmD4qJJULgM7M61XN9Tw7TYJchMws44C9oZIgs
31         uEgUyW1YQlxTALZAW4SroM6D/WU=
32        </ds:Modulus>
33        <ds:Exponent>AQAB</ds:Exponent>
34       </ds:RSAKeyValue>
35      </ds:KeyValue>
36     </ds:KeyInfo>
37    </saml:SubjectConfirmation>
38   </saml:Subject>
39  </saml:AuthenticationStatement>
40  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
41   <ds:SignedInfo>
42    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/
        xml-exc-c14n#"/>
43    <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig
        #rsa-sha1"/>
44    <ds:Reference URI="#_bac2462fd101f41567cabda33dbda1f6">
45     <ds:Transforms>
46      <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#
          enveloped-signature"/>
47      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
          c14n#">
48       <ec:InclusiveNamespaces
49        PrefixList="code ds kind rw saml samlp typens #default xsd
            xsi"
50        xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
51      </ds:Transform>
52     </ds:Transforms>
53     <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#
         sha1"/>
54     <ds:DigestValue>mGlM/PFNp5y4CsnF5Sn8vvTKy/I=</ds:DigestValue>
55    </ds:Reference>
56   </ds:SignedInfo>
57   <ds:SignatureValue>
58    xcUhn/vBUuzvNkLjEIWd9ZRurll3rFGcvGlgR1O31cyFtst1XlZLX
59    Oebm4bHBHfDkVJrUAk+L7JPf9cxbOhnYQ==
60   </ds:SignatureValue>
61   <ds:KeyInfo>
62    <ds:X509Data>
63     <ds:X509Certificate>
64      MIIBrTCCAVcCAxAAATANBgkqhkiG9w0BAQQFADByMQwwCgYDVQQ
65      KEwNLVEgxDDAKBgNVBAsTA1BEQzESMBAGA1UEBxMJU3RvY2tob2
66      xtMRIwEAYDVQQIEwlTdG9ja2hvbG0xCzAJBgNVBAYTAlNFMR8wH
67      QYDVQQDExZDZXJ0aWZpY2F0aW9uQXV0aG9yaXR5MB4XDTA2MTAw
68      NTEzMzgxM1oXDTA3MTAwNTEzMzgxM1owTzELMAkGA1UEBhMCU0U
69      xEjAQBgNVBAgTCVN0b2NraG9sbTEMMAoGA1UEChMDS1RIMQwwCg
```

```
70      YDVQQLEwNQREMxEDAOBgNVBAMTB0VzdGViYW4wXDANBgkqhkiG9
71      w0BAQEFAANLADBIAkEA0iheU3hQcaYHq4YrtVIYfyJnoNJKNBmb
72      TAkfV6CVOq7g10WBmxLzPjLwLuR7oF8vw9e9dv9QV1La+ubd6o5
73      llwIDAQABMA0GCSqGSIb3DQEBBAUAA0EAEaWtYqjankvren2DmA
74      QuOmB/fsc71T84FbgcoATFvVYB9SHfy2A1u9cMT7zCOEXkTg6RL
75      Tt2yac6SB/QaqXkWw==
76     </ds:X509Certificate>
77    </ds:X509Data>
78   </ds:KeyInfo>
79  </ds:Signature>
80 </saml:Assertion>
```

Listing C.6. Returned assertion in Kerberos → SAML translation

C.3 X.509 ⟹ Kerberos Test

C.3.1 RST Message

```
1  <soapenv:Envelope
2   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <soapenv:Header>
6    <wsse:Security
7     soapenv:mustUnderstand="1"
8     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-wssecurity-secext-1.0.xsd">
9     <ds:Signature
10     Id="Signature-8140933"
11     xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
12     <ds:SignedInfo>
13      <ds:CanonicalizationMethod Algorithm="http://www.w3.org
            /2001/10/xml-exc-c14n#"/>
14      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
            xmldsig#dsa-sha1"/>
15      <ds:Reference URI="#id-78219">
16       <ds:Transforms>
17        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
              c14n#"/>
18       </ds:Transforms>
19       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig
             #sha1"/>
20       <ds:DigestValue>IfCArIoKxBfWWWONm+kRNLgW770=</ds:DigestValue>
21      </ds:Reference>
22     </ds:SignedInfo>
23     <ds:SignatureValue>
24      Wfs6HbSx0LuOr+lhf2WAnHmLnjOWhKXYO2QeceRO5sx+hlS7JhjB/w==
25     </ds:SignatureValue>
26     <ds:KeyInfo Id="KeyId-17423963">
27      <wsse:SecurityTokenReference
28       wsu:Id="STRId-20843194"
29       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-wssecurity-utility-1.0.xsd">
30       <ds:X509Data>
```

```
31        <ds:X509IssuerSerial>
32         <ds:X509IssuerName>CN=Esteban,OU=NADA,O=KTH,C=SE</
               ds:X509IssuerName>
33         <ds:X509SerialNumber>1171582785</ds:X509SerialNumber>
34        </ds:X509IssuerSerial>
35       </ds:X509Data>
36      </wsse:SecurityTokenReference>
37     </ds:KeyInfo>
38    </ds:Signature>
39   </wsse:Security>
40  </soapenv:Header>
41  <soapenv:Body
42   wsu:Id="id-78219"
43   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-wssecurity-utility-1.0.xsd">
44   <wst:RequestSecurityToken
45    xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
46    <wst:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/
         Issue</wst:RequestType>
47    <wst:TokenType>
48     http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile
          -1.1
49    </wst:TokenType>
50    <wst:AppliesTo xmlns:wst="http://schemas.xmlsoap.org/ws/2004/09/
         policy">
51     <wsa:EndpointReference
52      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
53      <wsa:Address>mehrana.nada.kth.se</wsa:Address>
54     </wsa:EndpointReference>
55    </wst:AppliesTo>
56    <wst:Claims
57     Dialect="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-x509-token-profile-1.0#X509v3">
58     <wsse:BinarySecurityToken
59      EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-soap-message-security-1.0#Base64Binary"
60      ValueType="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-x509-token-profile-1.0#X509v3"
61      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
           -200401-wss-wssecurity-secext-1.0.xsd">
62      MIIBrTCCAVcCAxAAATANBgkqhkiG9w0BAQQFADByMQwwCgYDVQQKEwN
63      LVEgxDDAKBgNVBAsTA1BEQzESMBAGA1UEBxMJU3RvY2tob2xtMRIwEA
64      YDVQQIEwlTdG9ja2hvbG0xCzAJBgNVBAYTAlNFMR8wHQYDVQQDExZDZ
65      XJ0aWZpY2F0aW9uQXV0aG9yaXR5MB4XDTA2MTAwNTEzMzgxM1oXDTA3
66      MTAwNTEzMzgxM1owTzELMAkGA1UEBhMCU0UxEjAQBgNVBAgTCVN0b2N
67      raG9sbTEMMAoGA1UEChMDS1RIMQwwCgYDVQQLEwNQREMxEDAOBgNVBA
68      MTB0VzdGViYW4wXDANBgkqhkiG9w0BAQEFAANLADBIAkEA0iheU3hQc
69      aYHq4YrtVIYfyJnoNJKNBmbTAkfV6CVOq7g10WBmxLzPjLwLuR7oF8v
70      w9e9dv9QV1La+ubd6o5llwIDAQABMA0GCSqGSIb3DQEBBAUAA0EAEaW
71      tYqjankvren2DmAQuOmB/fsc71T84FbgcoATFvVYB9SHfy2A1u9cMT7
72      zCOEXkTg6RLTt2yac6SB/QaqXkWw==
73     </wsse:BinarySecurityToken>
74    </wst:Claims>
75   </wst:RequestSecurityToken>
```

```
 </soapenv:Body>
</soapenv:Envelope>
```

Listing C.7. RST Message in X.509 → Kerberos translation

C.3.2 RSTR Message

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
  <wsse:Security
   soapenv:mustUnderstand="1"
   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-wssecurity-secext-1.0.xsd">
   <ds:Signature
    Id="Signature-3373197"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
     <ds:CanonicalizationMethod Algorithm="http://www.w3.org
         /2001/10/xml-exc-c14n#"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
          xmldsig#dsa-sha1"/>
      <ds:Reference URI="#id-12608429">
       <ds:Transforms>
        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
            c14n#"/>
       </ds:Transforms>
       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/
           xmldsig#sha1"/>
       <ds:DigestValue>fxPQLRFEjT13odhj4PkQ2Ja5dJ0=</ds:DigestValue
           >
      </ds:Reference>
      <ds:Reference URI="#SigConf-176713">
       <ds:Transforms>
        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
            c14n#"/>
       </ds:Transforms>
       <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/
           xmldsig#sha1"/>
       <ds:DigestValue>SZI63lNXlSLJqgtpNfAjMfK9KLU=</ds:DigestValue
           >
      </ds:Reference>
     </ds:SignedInfo>
     <ds:SignatureValue>
      ZWKwh1ykPpGCkUEgq8gQsm5V6J9ZSLVCw7qh2sUNIItk6i59lh4QwQ==
     </ds:SignatureValue>
    <ds:KeyInfo Id="KeyId-33078541">
     <wsse:SecurityTokenReference
      wsu:Id="STRId-21217085"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis
          -200401-wss-wssecurity-utility-1.0.xsd">
      <ds:X509Data>
```

```
38          <ds:X509IssuerSerial>
39           <ds:X509IssuerName>CN=STS,OU=PDC,O=KTH,C=SE</
                 ds:X509IssuerName>
40           <ds:X509SerialNumber>1171583176</ds:X509SerialNumber>
41          </ds:X509IssuerSerial>
42         </ds:X509Data>
43        </wsse:SecurityTokenReference>
44       </ds:KeyInfo>
45      </ds:Signature>
46      <wsse11:SignatureConfirmation
47       Value="Wfs6HbSx0LuOr+lhf2WAnHmLnjOWhKXYO2QeceRO5sx+hlS7JhjB/w==
             "
48       wsu:Id="SigConf-176713"
49       xmlns:wsse11="http://docs.oasis-open.org/wss/2005/xx/oasis-2005
             xx-wss-wssecurity-secext-1.1.xsd"
50       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
             wss-wssecurity-utility-1.0.xsd"/>
51    </wsse:Security>
52  </soapenv:Header>
53  <soapenv:Body
54   wsu:Id="id-12608429"
55   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
         wss-wssecurity-utility-1.0.xsd">
56   <wst:RequestSecurityTokenResponse
57    xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
58    <wst:TokenType>
59     http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile
             -1.1
60    </wst:TokenType>
61    <wst:RequestedSecurityToken>
62     <wsse:BinarySecurityToken
63      EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-soap-message-security-1.0#Base64Binary"
64      ValueType="http://docs.oasis-open.org/wss/oasis-wss-kerberos-
             token-profile-1.1#GSS_Kerberosv5_AP_REQ"
65      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
             -200401-wss-wssecurity-secext-1.0.xsd">
66      YYIBBDCCAQCgAwIBBaEIGwZQREMuU0WiJTAjoAMCAQKhHDAaGwNzdHM
67      bE21laHJhbmEubmFkYS5rdGguc2WjgccwgcSgAwIBEKEDAgEBooG3BI
68      G0DS14xARZj59t7OH62A3u1Sin5PenBUV90nuYLI7cCJqzi1gYw6BDM
69      iABVfBmz5L6KAmSM5DT/wata8XJ427ljNWd0A/0eqt1GNjVDiPLLEPj
70      HBl9FvSvTBrCva4tGxC/IIDZQqjieAIhVN7nMYzG84t0e271Oqsn3XW
71      czH7fH6ecZuoPBOSC7lJqyiFygdRRHC/Qx0vJjy23mgWQRPfUPhI7IT
72      rM83UheTyd8X9186ifufSe
73     </wsse:BinarySecurityToken>
74    </wst:RequestedSecurityToken>
75    <wst:RequestedProofToken>
76     <xenc:EncryptedKey
77      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
78      <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/
             xmlenc#rsa-1_5"/>
79      <xenc:CipherData>
80       <xenc:CipherValue>
81        IKBvXHK1pihpPyke6k3aWveMNUEszsj/9t6Gu6XwzhvFIwHPcEPis
```

```
82         00sKsdGheBPCMSs2eGk6UHmmiTTX72FmA==
83        </xenc:CipherValue>
84       </xenc:CipherData>
85      </xenc:EncryptedKey>
86     </wst:RequestedProofToken>
87   </wst:RequestSecurityTokenResponse>
88  </soapenv:Body>
89 </soapenv:Envelope>
```

Listing C.8. RSTR Message in X.509 → Kerberos translation

C.3.3 Returned TGT

```
1  Target Service's name: sts/mehrana.nada.kth.se
2  Issuer's Realm: PDC.SE
3
4  Encrypted part of the ticket:
5    Flags: INITIAL;PRE-AUTHENT
6    Session Key:
7      EncryptionKey: keyType=16 kvno=null keyValue (hex dump)=
8        0000: C8 F8 A1 B5 92 F4 DA E3   0B A2 02 FE 26 AD F2 E0
9        0010: 51 97 BF 7C B6 EA 34 20
10   Client's Realm: PDC.SE
11   Client's Principal Name: Esteban
12   Authentication Time: Fri Feb 16 00:54:41 CET 2007
13   Start Time: Fri Feb 16 00:54:41 CET 2007
14   End Time: Fri Feb 16 01:59:41 CET 2007
```

Listing C.9. Returned TGT in X.509 → Kerberos translation

Bibliography

[1] Anthony Nadalin et al. Web Services Security: SOAP Message Security 1.0 (WS-Security 2004). OASIS Standard, OASIS, Mar 2004.

[2] Wikipedia. Grid computing. http://en.wikipedia.org/wiki/Computational_Grid (last accessed Nov 28, 2006).

[3] C. Newman et al. The Kerberos Network Authentication Service (V5). RFC 4120, Internet Engineering Task Force, Jul 2005.

[4] Fulvio Ricciardi. The Kerberos protocol and its implementations. Version 1.0.3, INFN – the National Institute of Nuclear Physics (Italy), Nov 2006. http://www.zeroshell.net/eng/kerberos/ (last accessed Feb 28, 2007).

[5] R. Shirey. Internet Security Glossary. RFC 2828, Internet Engineering Task Force, May 2000.

[6] R. Housley et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280, Internet Engineering Task Force, Apr 2002.

[7] Tim Bray et al. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, World Wide Web Consortium (W3C), Aug 2006. http://www.w3.org/TR/xml/ (last accessed Feb 28, 2007).

[8] Donald Eastlake et al. XML-Signature Syntax and Processing. W3C Recommendation, World Wide Web Consortium (W3C), Feb 2002. http://www.w3.org/TR/xmldsig-core/ (last accessed Feb 21, 2007).

[9] Donald Eastlake et al. XML Encryption Syntax and Processing. W3C Recommendation, World Wide Web Consortium (W3C), Dec 2002. http://www.w3.org/TR/xmlenc-core/ (last accessed Feb 21, 2007).

[10] Eve Maler et al. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V1.1. OASIS Standard, OASIS, Sep 2003.

[11] Scott Cantor et al. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, OASIS, Mar 2005.

[12] Nick Ragouzis et al. Security Assertion Markup Language (SAML) V2.0 Technical Overview. Working Draft 09, OASIS, Jul 2006.

[13] Eve Maler et al. Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML) V1.1. OASIS Standard, OASIS, Sep 2003.

[14] Scott Cantor et al. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, OASIS, Mar 2005.

[15] David Booth et al. Web Services Architecture. W3C Working Group Note, World Wide Web Consortium (W3C), Feb 2004. http://www.w3.org/TR/ws-arch/ (last accessed Feb 21, 2007).

[16] Kelvin Lawrence et al. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). OASIS Standard Specification, OASIS, Feb 2006.

[17] Anthony Nadalin et al. Web Services Security UsernameToken Profile 1.0. OASIS Standard, OASIS, Mar 2004.

[18] Kelvin Lawrence et al. Web Services Security UsernameToken Profile 1.1. OASIS Standard Specification, OASIS, Feb 2006.

[19] Phillip Hallam-Baker et al. Web Services Security SAML Token Profile 1.0. OASIS Standard, OASIS, Dec 2004.

[20] Kelvin Lawrence et al. Web Services Security SAML Token Profile 1.1. OASIS Standard Specification, OASIS, Feb 2006.

[21] Phillip Hallam-Baker et al. Web Services Security X.509 Certificate Token Profile 1.0. OASIS Standard, OASIS, Mar 2004.

[22] Kelvin Lawrence et al. Web Services Security X.509 Certificate Token Profile 1.1. OASIS Standard Specification, OASIS, Feb 2006.

[23] Kelvin Lawrence et al. Web Services Security Kerberos Token Profile 1.1. OASIS Standard Specification, OASIS, Feb 2006.

[24] Paul Madsen. WS-Trust: Interoperate Security for Web Services. *XML.com*, Jun 2003. http://www.xml.com/pub/a/ws/2003/06/24/ws-trust.html (last accessed Feb 21, 2007).

[25] Steve Anderson et al. Web Services Trust Language (WS-Trust). Technical report, BEA, IBM, Microsoft, RSA Security, VeriSign and others, Feb 2005.

[26] Siddharth Bajaj et al. Web Services Federation Language (WS-Federation) 1.0. Technical report, BEA, IBM, Microsoft, RSA Security and VeriSign, Jul 2003.

[27] Tom Scavo et al. Shibboleth Architecture Technical Overview. Working Draft 02, Jun 2005.

[28] Scott Cantor et al. Shibboleth Architecture Protocols and Profiles. Technical report, Sep 2005.

[29] William Doster, Marcus Watts, and Dan Hyde. The KX.509 Protocol. Technical report, Center for Information Technology Integration (CITI), Feb 2001.

[30] Olga Kornievskaia et al. Kerberized Credential Translation: A Solution to Web Access Control. Technical report, Center for Information Technology Integration (CITI), Feb 2001.

[31] Mehran Ahsant et al. Dynamic Trust Federation in Grids. In Proceedings of The 4th International Conference on Trust Management, Pisa, Tuscany, Italy.

[32] Ian Foster et al. A Security Architecture for Computational Grids. Proceedings of the 5th ACM conference on Computer and communications security, Nov 1998.

[33] Bilal Siddiqui. Web Services Security, Part 1. *XML.com*, Mar 2003. http://webservices.xml.com/pub/a/ws/2003/03/04/security.html (last accessed Feb 21, 2007).

[34] Bilal Siddiqui. Web Services Security, Part 2. *XML.com*, Apr 2003. http://www.xml.com/pub/a/ws/2003/04/01/security.html (last accessed Feb 21, 2007).

[35] Bilal Siddiqui. Web Services Security, Part 3. *XML.com*, May 2003. http://www.xml.com/pub/a/ws/2003/05/13/security.html (last accessed Feb 21, 2007).

[36] Bilal Siddiqui. Web Services Security, Part 4. *XML.com*, Jul 2003. http://www.xml.com/pub/a/ws/2003/07/22/security.html (last accessed Feb 21, 2007).

[37] Paul Madsen et al. SAML V2.0 Executive Overview. Committee Draft 01, OASIS, April 2005.

[38] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Technical report, 2001.

[39] V. Welch et al. Security for Grid Services. Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), Jun 2003.

[40] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, Nov 2002.