Kerberos realm crossover

Caño Bellatriu, O.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Security Group

Kerberos Realm Crossover

Master Thesis

Oriol Caño Bellatriu

Supervisors:
Dr. Boris Skoric
Dr.ir. Rick van Rein
Dr.ir. L.A.M. Berry Schoenmakers

version 1.0

Eindhoven, May 2016

Abstract

Kerberos is a well-known and widely used authentication protocol that uses a ticket-based system to authenticate clients and services to each other. The clients and services are organised in so-called realms, which are controlled by a secure central service, called Key Distribution Center. Kerberos offers a way for clients from a realm to contact services from a different realm. However, this communication requires the interaction of the administrators of both realms. In order to set it up, the administrators need to agree on a shared secret key and store it in the Kerberos database of both realms.
In this thesis, we introduce a protocol, named KXOVER, that allows clients to request services from remote realms without any administrator interaction. KXOVER will use DANE and the PKINIT Kerberos plugin to authenticate the Key Distribution Centers to each other and to perform an Elliptic Curve Diffie-Hellman key exchange, used to agree on the shared secret key with Perfect Forward Secrecy properties.
Besides the design of the KXOVER protocol, we also provide a proof-of-concept implementation in order to show that realm cross-over can be done without the interaction of the administrators. In order to complete our work, we also provide a security analysis of both the design and the proof-of-concept implementation of KXOVER.

Acknowledgements

I would like to thank Rick for his constant supervision, his invaluable help, and his fast answers at unbelievable times of the day. You are the mastermind of this project and I am glad I had the opportunity to participate in it.
I would like to thank Berry for his supervision and for his help on writing this thesis. It got a lot better thanks to your comments and suggestions.
Finally, I would like to thank E.S.T. Suca and all its members for countless hours of distractions and fun, which, even though they are not part of a thesis, help a lot with the accumulated stress that comes with it.

Quiero darle las gracias a Nathalia, que me ha ayudado a superar momentos difíciles y siempre ha estado allí. Sin tí mi vida sería más aburrida y monótona.

També vull donar les gràcies als meus pares, que m'han suportat econòmicament durant tota la meva trajectòria acadèmica i m'han donat ànims per acabar.

Contents

List of Figures

Listings

Glossary

AP-REP Application Reply. 8

AP-REQ Application Request. 8

AS-REP Authentication Service Reply. 5

AS-REQ Authentication Service Request. 5

ASN.1 Abstract Syntax Notation One. 12

BER Basic Encoding Rules. 12

CMS Cryptographic Message Syntax. 22

DANE DNS-based Authentication of Named Entities. 12

DER Distinguished Encoding Rules. 11

DNS Domain Name System. 12

DNSSEC Domain Name System Security Extensions. 12

KDC Key Distribution Center. 3

KXOVER Kerberos Realm Crossover. 15

RFC Request For Comments. 1

SRV Service record. 12

TGS-REP Ticket Granting Service Reply. 7

TGS-REQ Ticket Granting Service Request. 6

TXT Text record. 12

Chapter 1

Introduction

Kerberos is a network authentication protocol. It allows client and server applications to authenticate to each other over an insecure network, using so-called tickets as credentials. In order to do so, Kerberos relies on a trusted party set up in the network, which shares secret keys with all clients and servers. Another benefit of the protocol is the fact that it provides a single sign-on service for users, which makes it really convenient for the user experience. In a typical Kerberos scenario, users only need to log in by typing their passwords once a day.
Kerberos was initially developed by the Massachusetts Institute of Technology (MIT). Several versions of the protocol were developed internally, until they released Kerberos version 4 in the late 1980s. A version 5 of Kerberos was released in 1993, which was later obsoleted by RFC 4120 [4] in 2005.
Kerberos is a widely used and known protocol, being featured in many Unix-like operating systems. Microsoft also developed their own implementation of the protocol, called Active Directory. There are several implementations of the protocol, like Heimdal [5], Shishi [21], the one developed by Microsoft, and the one released by MIT.
The protocol is constantly being updated and improved, thanks to the creation of the Kerberos Consortium in 2007 by MIT. This consortium receives funding from well known companies such as Oracle, Apple Inc., Google, Microsoft and others. Thanks to this consortium, the Kerberos community is really broad, and there are lots of projects being developed constantly to expand the functionalities of the protocol, keeping its security model intact.

1.1 Goal

The main goal of this thesis is to design a Kerberos protocol that will allow automatic communication between two different realms. On top of designing the protocol, a prototype is built in order to show that the designed protocol functions properly. The prototype also helps in the further refinement of the protocol itself, pointing out bits and pieces that could be improved, and modifications that need to be done in the current Kerberos implementations. The ultimate goal, though, is the formal definition of this protocol in an RFC, and the later inclusion of the protocol into MIT's Kerberos code base. However, these are long-term goals that are beyond the scope of this thesis.

1.2 Motivation

This thesis is part of a bigger project, called ARPA2 [1]. The main goal of ARPA2 is to improve the current architecture of the Internet. In order to do so, they plan on providing a hosting platform that can be installed by hosting companies. This platform would give users more control, centralising all their online identities into one, controlled by the users themselves. It would offer better security by using cryptography instead of passwords to authenticate the users to online services. And it would also offer better privacy, by letting the user own all their data, instead of relying on third-party systems.

In order to do all that, they have designed a so-called *Identity Provider*, which will provide users with a way of managing their online identities. This whole idea of the system is to let users authenticate themselves once they log on to their systems, and from there, the system will authenticate the user automatically to online servers. This way, the user only has to enter one password on a trusted machine, ownend by the user, and all the following authentication processes are done through cryptographic means. This will be convenient for users, since they will not need to create a new password for each online service they use. On top of that, the Identity Provider will unite all the currently different online identities of a user in a single one. For example, nowadays, you have to create different users to access different services, like *Facebook*, *Twitter*, *IRC*, and many more. With the Identity Provider, a user will have one online identity, controlled by him, and use all these different services using that identity. On top of that online identity, the user could also make use of pseudonyms or aliases.

However, sometimes it is desirable to have different identities on different services. That is why the Identity Provider will let users use different roles or groups, in order to have the freedom of choosing what they want their online identity to be.

As can be seen, the Identity Provider system has been designed around Kerberos, but in order to apply it to the Internet as a whole, realm crossover is necessary.

Chapter 2

Background

2.1 Kerberos

Kerberos is a protocol that authenticates clients and services over an insecure network, and also provides a single sign-on point for its users. In order to achieve that, it uses a trusted party called Key Distribution Center (KDC) and three exchanges, separated into Authentication Service, Ticket-Granting Service and Client/Server Authentication. These exchanges will be explained in the following sections.

Kerberos is organized in so-called Realms, which represent networks controlled by a KDC. Inside a Realm you can find both clients that want to use a service, and servers that provide those services. Realms are identified by their Realm name. Realm names are case sensitive, and the most used style for realm names is the domain style. In this style, realm names must look like domain names and it is recommended by convention that all characters are uppercase. All realm names used in this thesis will follow this style.

The entity controlling a realm is the Key Distribution Center. It acts as a trusted party, since it shares long-lasting secret keys with all clients and services of its network. These long-lasting secret keys are stored in a secured database. The database holds unique identities of all clients and services of the network, which are called Principals. Each entry in the database contains a principal and the attributes and policies associated with that principal. The database also stores secret keys used for special instances in Kerberos. One type of those instances are the secret keys used to perform cross-realm authentication, which will be explained later on. The KDC must be hosted by a machine reachable by everyone in the realm. However, that machine must have strict security measures to shield it from intrusion. This is necessary because the KDC controls all authentications on the system, and compromising this authentication infrastructure would allow an attacker to impersonate any principal on the network after stealing its long-term key.

As we have mentioned before, Kerberos works with so-called tickets. Tickets represent short-term session keys between a client and a service. Tickets are issued by the KDC, and they contain two copies of the session key, one can only be read by the client and the other can only be read by the service. A special instance of a ticket is the Ticket Granting Ticket,

which represents a session key shared between a client and the KDC itself, this is explained in Section 2.1.2.

The KDC is divided into two different servers, the Authentication Server(AS), and the Ticket Granting Server(TGS). The Authentication Server is the one handling the login process of clients. It receives log in requests from clients, and hands out ticket granting tickets.
On the other hand, the Ticket Granting Server receives the ticket granting ticket from a client, and hands out tickets for the servers that reside in the realm.

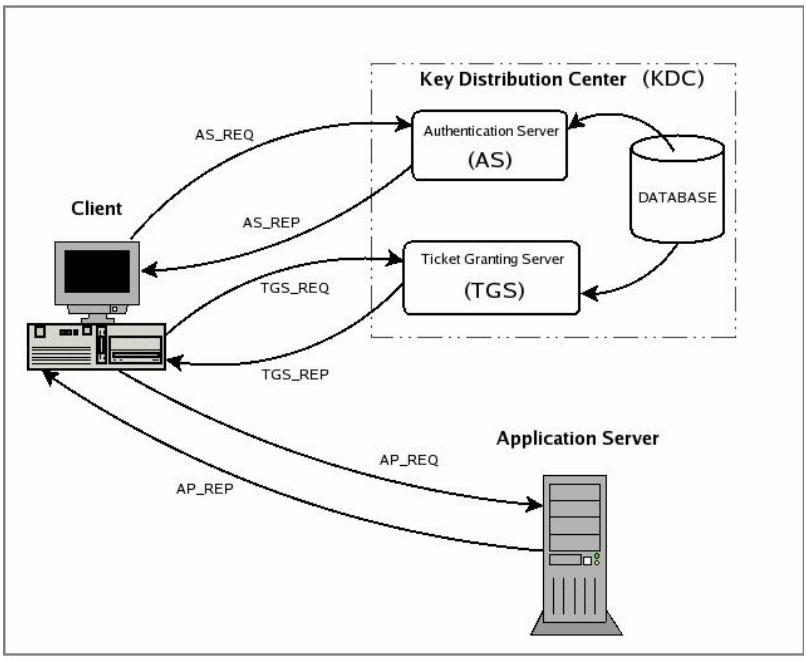


Figure 2.1: Kerberos Design

Source: `http://www.kerberos.org/images/krbmsg.gif`

2.1.1 Authentication Service Exchange

The first step of the Kerberos workflow is the client authentication. As has been mentioned before, Kerberos provides a single sign-on mechanism, meaning that users do not need to login to every service they want to use.
In this case, Kerberos provides the client with a Ticket Granting Ticket that can be used to request tickets to the services in that realm. Kerberos provides authentication over insecure networks, which means that the secret key of the user is not sent directly over the network. Instead, the KDC encrypts part of the response with the secret key of the user. This way, only the client with the correct secret key will be able to authenticate itself to the KDC. This is the only step on the whole process where the exchange is encrypted using the long-term secret key of the user. This means that an attacker could use a captured reply message in order to try and guess the long-term secret key of a user.

The client authentication process is represented by the two top-most arrows in Figure 2.1, and works as follows.

- First of all, the client generates an AS-REQ message that contains the client's name and realm, amongst others, and sends it to the Authentication Server.
  A specification of the AS-REQ message can be seen in Listing 2.1. This specification has been taken from RFC 4120 [4]. When the AS receives the request it checks if the client name exists, and then retrieves the long-lasting secret key of the client from the database.

- The KDC generates an AS-REP message that has two main components.
  A *Client/TGS Session Key*, which will be encrypted using a secret key derived from the client's password. A *Ticket Granting Ticket (TGT)*, encrypted using the TGS's secret key. This ticket contains information identifying the client, and specifying the expiration date of the ticket. It also contains a copy of the same *Client/TGS Session Key*. A specification of the AS-REP message can be seen in Listing 2.2.

```
AS-REQ          ::= [APPLICATION 10] KDC-REQ

TGS-REQ         ::= [APPLICATION 12] KDC-REQ

KDC-REQ         ::= SEQUENCE {
        -- NOTE: first tag is [1], not [0]
        pvno            [1] INTEGER (5) ,
        msg-type        [2] INTEGER (10 -- AS -- | 12 -- TGS --),
        padata          [3] SEQUENCE OF PA-DATA OPTIONAL
                            -- NOTE: not empty --,
        req-body        [4] KDC-REQ-BODY
}

KDC-REQ-BODY    ::= SEQUENCE {
        kdc-options             [0] KDCOptions,
        cname                   [1] PrincipalName OPTIONAL
                                    -- Used only in AS-REQ --,
        realm                   [2] Realm
                                    -- Server's realm
                                    -- Also client's in AS-REQ --,
        sname                   [3] PrincipalName OPTIONAL,
        from                    [4] KerberosTime OPTIONAL,
        till                    [5] KerberosTime,
        rtime                   [6] KerberosTime OPTIONAL,
        nonce                   [7] UInt32,
        etype                   [8] SEQUENCE OF Int32 -- EncryptionType
                                    -- in preference order --,
        addresses               [9] HostAddresses OPTIONAL,
        enc-authorization-data  [10] EncryptedData OPTIONAL
                                    -- AuthorizationData --,
        additional-tickets      [11] SEQUENCE OF Ticket OPTIONAL
                                        -- NOTE: not empty
}
```

Listing 2.1: Request message ASN.1 specification[4]

When the client receives the reply, it will try to decrypt the session key with a secret key derived from the password entered by the user. This session key will be needed for a later step in the protocol. The TGT is encrypted using a secret key unknown to the client, which means that the client cannot access the information inside the ticket. The TGT can only be read by the correct recipient, if it is sent unaltered.

2.1.2 Ticket-Granting Service Exchange

The Ticket-Granting Service Exchange is used to request mutual authentication to a service in the realm. It can only be performed by clients that have authenticated themselves using the Authentication Service Exchange. The exchange is represented by the two middle arrows in Figure 2.1, and works as follows.

- In order to request authentication to a service, the client will need to generate a TGS-

REQ message and send it to the Ticket Granting Service.
The TGS-REQ message is similar to the AS-REQ one, and so they share the same specification (Listing 2.1). However, in this request the client has to include two extra components.
First, the client needs to include the Ticket Granting Ticket obtained from the AS.
Finally, it also includes an authenticator, which contains the client name and a timestamp, and is encrypted using the *Client/TGS Session Key*, also obtained from the AS. This authenticator is used to prove the client's authentication to the TGS and to avoid replay attacks.

- When the Ticket Granting Service receives the request, it first obtains the TGT, and decrypts it with its own secret key. From the ticket, it gets the *Client/TGS Session Key*, which will then use it to decrypt the authenticator. After checking the authenticator, the TGS generates a TGS-REP message. The TGS-REP specification can be seen in Listing 2.2.
  Again, the reply message contains two main components.
  A *Client/Server Session Key*, which is encrypted using the *Client/TGS Session Key*.
  A *Client-to-Server Ticket*, encrypted using the Server's secret key. This ticket contains information about the client, identifying it and specifying the expiration date of the ticket. It also contains a copy of the *Client/Server Session Key*.

Once the client receives the TGS-REP, it decrypts the *Client/Server Session Key* using the *Client/TGS Session Key*.

```
AS-REP          ::= [APPLICATION 11] KDC-REP

TGS-REP         ::= [APPLICATION 13] KDC-REP

KDC-REP         ::= SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (11 -- AS -- | 13 -- TGS --),
        padata          [2] SEQUENCE OF PA-DATA OPTIONAL
                                -- NOTE: not empty --,
        crealm          [3] Realm,
        cname           [4] PrincipalName,
        ticket          [5] Ticket,
        enc-part        [6] EncryptedData
                                -- EncASRepPart or EncTGSRepPart,
                                -- as appropriate
}
```

Listing 2.2: Reply message ASN.1 specification[4]

2.1.3 Client/Server Authentication Exchange

When a client has obtained a ticket to a service, it can contact the service directly to finalize the mutual authentication. In order to do so, the last Kerberos exchange needs to happen.

This exchange is represented with the two bottom-most arrows in Figure 2.1. The exchange works as follows.

- The client has to craft an AP-REQ message, specified in Listing 2.3. This message is composed of two parts. First, the ticket that the client obtained from the TGS, the one that the client could not decrypt. Then, a new Authenticator, which includes information about the client and a timestamp. The Authenticator is encrypted using the *Client/Server Session Key*.

- The service receives the message, and uses its long-term secret key to decrypt the ticket. From the decrypted ticket, it obtains the *Client/Server Session Key*, which it then uses to decrypt the Authenticator, which is used to prove that the client has the other half of the ticket, and to chek for replay attacks. Finally, the server creates an AP-REP message, specified in Listing 2.3, and sends it to the client. The AP-REP message contains the timestamp found in the Authenticator in order to prove that the service is the correct one, ensuring mutual authentication.

When the client receives the AP-REP message, it checks whether the received timestamp is correct or not. If it is correct, the client can be sure that it has authenticated itself to the server, and the normal Client/Server interaction can start. The interaction between the client and the server can be encrypted using the shared *Client/Server Session Key*.

```
AP-REQ          ::= [APPLICATION 14] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (14),
        ap-options      [2] APOptions,
        ticket          [3] Ticket,
        authenticator   [4] EncryptedData -- Authenticator
}

Authenticator   ::= [APPLICATION 2] SEQUENCE  {
        authenticator-vno       [0] INTEGER (5),
        crealm                  [1] Realm,
        cname                   [2] PrincipalName,
        cksum                   [3] Checksum OPTIONAL,
        cusec                   [4] Microseconds,
        ctime                   [5] KerberosTime,
        subkey                  [6] EncryptionKey OPTIONAL,
        seq-number              [7] UInt32 OPTIONAL,
        authorization-data      [8] AuthorizationData OPTIONAL
}

AP-REP          ::= [APPLICATION 15] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (15),
        enc-part        [2] EncryptedData -- EncAPRepPart
}

EncAPRepPart    ::= [APPLICATION 27] SEQUENCE {
        ctime           [0] KerberosTime,
        cusec           [1] Microseconds,
        subkey          [2] EncryptionKey OPTIONAL,
        seq-number      [3] UInt32 OPTIONAL
}
```

Listing 2.3: AP exchange messages ASN.1 specification[4]

2.1.4 Special Message Exchanges

Kerberos also allows clients to send messages using the benefits of Kerberos. These messages are not part of the main interaction of the Kerberos system. This messages can be used by clients and services of the realm, and its use depends on the requirements of the message exchange. There are three types of messages.

- **KRB_SAFE**. This message exchange can be used by clients requiring the ability to detect modifications of the messages they exchange. In order to achieve it, the messages include a keyed collision-proof checksum of the used data and some control information. The message exchange requires an established session key between the two parties exchanging the messages.
- **KRB_PRIV**. This message exchange can be used by clients requiring confidentiality and authenticity of the messages they exchange. In order to achieve it, the messages

are encrypted, and control information is included. The message exchange requires an established session key between the two parties, in order to encrypt and sign the messages.

- **KRB_CRED**. This message exchange can be used by clients requiring the ability to send Kerberos credentials from one machine to another. In order to achieve it, it sends the tickets together with encrypted data containing the session keys and other information associated with the tickets. This exchange also needs an established session key between the two parties exchanging the messages.

2.2 Realm Crossover

Everything we have seen so far about Kerberos concerns a single realm. However, a KDC in a realm may also authenticate a user in one realm to a service in another realm. This is called Cross-Realm Authentication.
Currently, Cross-Realm Authentication needs the interaction of administrators in both realms. In order to set it up, a common principal has to be created in both databases. All principals for Cross-Realm Authentication need to have a common prefix, namely *krbtgt*, followed by the service realm and the client realm, separated by an @. An example of such principal will be seen later in this section.
These principals allow for one-way authentication, which means that two principals have to be created in order to allow the authentication to work both ways.
The process for setting up Cross-Realm Authentication involves the administrators of the two realms to get in contact with each other using a trusted channel. This is necessary because the principals that have to be created in both databases need to have the same secret key, key version number and encryption types on both sides.
It is also really important to use good secret keys, MIT recommends passwords of at least 26 random ASCII characters.

As an example, if we want the clients of a realm called RHCP.DEV.ARPA2.ORGto be able to authenticate to services provided by a realm called EXAMPLE.COM, the administrators of both realms will need to contact each other and create a principal named `krbtgt/EXAMPLE.COM@RHCP.DEV.ARPA2.ORG` in both databases with the same secret key.
If the administrators wish to let a client from the realm EXAMPLE.COM to access a service provided by the realm RHCP.DEV.ARPA2.ORG, then a new principal would need to be created, this time named `krbtgt/RHCP.DEV.ARPA2.ORG@EXAMPLE.COM`.

2.3 Pre-authentication

Kerberos version 5 introduced pre-authentication. In order to introduce pre-authentication, a new field was added to the requests. This field is the one called *padata* (pre-authentication data). There exists a framework that defines how pre-authentication should work ([20]). However, since its creation, the use of the padata field has evolved, and nowadays it is also used to carry extensions to Kerberos that have nothing to do with proving the identity of the

user.
The padata field is composed of a sequence of padata-type and padata-value pairs, that is the reason why it is called a typed-hole. The padata-type element indicates how the padata-value needs to be interpreted. The padata-type contains an Integer and the padata-value usually contains the DER encoding of another type. The padata-type element is used as an identifier for the Kerberos extension being used. Since it works as an identifier, it has to be registered by IANA. In order to register a new Kerberos extension, an expert review is necessary. A list of the current Kerberos extensions and pre-authentication data types can be found in [6].

Kerberos has numerous extensions in the form of plugins. One of them is called *PKINIT*[9], and it offers Public Key Cryptography as a means of pre-authentication.
PKINIT integrates the usage of Public Key Cryptography into the initial authenticate exchange. Instead of prompting users for their password, this plugin allows the usage of either Diffie-Hellman Key exchange or Public Key Encryption.
In both cases, it uses a field already defined in the KDC-REQ(2.1) and KDC-REP(2.2) messages, called *padata*. This pre-authentication data contains all the necessary information to facilitate the authentication exchange. Concretely, the data holds an ASN.1 element called *SubjectPublicKeyInfo* that allows to include a public key, and to specify the algorithm you wish to use in conjunction with that key.
In the request, the client provides an AuthPack ASN.1 element, which contains an authenticator, the client public key information, the Cryptographic Message Syntax ([16]) encryption types supported by the client and an optional Diffie-Hellman nonce.
Besides this AuthPack element, the client may also provide a list of Certificate Authorities trusted by the client that can be used to certify the KDC, and a KDC public key that the client already has.
When the KDC receives the request, it has to validate the client using all the information provided in the request. After validating, the KDC has to generate a shared secret that will be used as a session key, either using Diffie-Hellman Key exchange, or Public Key Encryption, depending on the client's choice.
After generating the shared secret, the KDC has to make sure that the client also obtains the session key. It does so by using the padata field on the KDC-REP. This reply will either contain the Diffie-Hellman key information necessary to derive the session key or an encrypted KeyPack structure directly containing it.

There also exists an extension to PKINIT that adds support for Elliptic Curve Cryptography[10]. This extension adds Elliptic Curve Diffie-Hellman key exchange, and describes how to specify it.
In order to use it, a new element is introduced to PKINIT, called *ECParamenters*, that gives the choice to include a named curve, an implicit curve or a specified curve. Besides specifying the curve, the extension explains how you need to use the *SubjectPublicKeyInfo* element to specify the point acting as public key for the Elliptic Curve Diffie-Hellman exchange.

2.4 ASN.1

'ASN.1 is a formal notation used for describing data transmitted by telecommunications protocols, regardless of language implementation and physical representation of these data, whatever the application, whether complex or very simple.'[8]
As we have seen before in this chapter, Kerberos specifies its messages using the ASN.1 notation. This notation provides a certain number of pre-defined basic types, like integers, bit strings, character strings, etc. Also, it allows to construct self-defined types by providing elements like structures, lists and choices, amongst others.

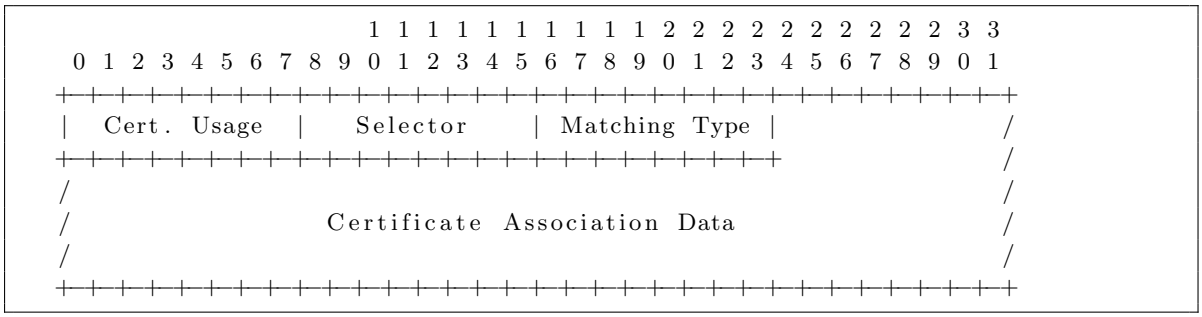
ASN.1 [7] defines the abstract syntax of the messages. However, it is also associated to several standardized encoding rules, that define how the data has to be encoded when including it into the ASN.1 syntax. These encoding rules include BER (Basic Encoding Rules), DER (Distinguished Encoding Rules), and many more.
Kerberos explicitly specifies the use of DER for its encodings. This encoding rules provide exactly one way of encoding an ASN.1 value. This is valuable for cryptographic situations, and it is also widely used for digital certificates such as X.509.

2.5 DNSSEC + DANE

The Domain Name System Security Extensions (DNSSEC) is a series of specifications that secure certain kinds of information provided by DNS.
DNSSEC was designed to protect applications from using forged or modified DNS data. In order to achieve this, all data obtained using DNSSEC is digitally signed. The system used by DNSSEC is similar to PKI. However, DNSSEC does not provide encryption, only authentication.

In the protocol designed in this thesis, all DNS look-ups will be executed using DNSSEC. This is necessary in order to maintain the security of the protocol. The protocol uses three different look-ups, the first two are an SRV and a TXT look-up. These two look-ups provide information about the location of the remote KDC, and the names of the realms it handles. The location of the server needs to be secured, otherwise when you contact a remote KDC for the first time you could be contacting a rogue KDC posing as the original one.

The third look-up will regard TLS Trust Anchors (TLSAs). This look-up is the most important one, since it will provide a means of authenticating signatures made by remote KDCs. The TLSA resource record is defined on the DANE TLSA protocol [15], and it is used to associate a TLS server certificate or public key with the domain name where the record is found. In the protocol designed in this thesis, we are not using DANE to associate a domain name to a TLS server, we associate it to a KDC instead. The structure of a TLSA record can be seen in Listing 2.4.



Listing 2.4: TLSA Format

A TLSA record is composed by 4 different fields.

- **Certificate Usage**. This field specifies the usage of the certificate association data. There are 4 values defined for this field.

  0. Certification usage 0 is used to specify a CA certificate.
  1. Certification usage 1 is used to specify an end entity certificate. The target certificate must pass PKIX certification path validation.
  2. Certification usage 2 is used to specify a certificate that must be used as the trust anchor when validating the end entity certificate.
  3. Certificate usage 3 is used to specify an end entity certificate as well. However, unlike usage 1, it does not need to pass PKIX certification validation.

- **Selector**. This field specifies which part of the TLS certificate presented by the server will be matched against the association data.

  0. Full certificate.
  1. *SubjectPublicKeyInfo*.

- **Matching Type**. This field specifies how the certificate association is presented.

  0. Exact match on selected content.
  1. SHA-256 hash of selected content.
  2. SHA-512 hash of selected content.

- **Certificate Association Data**. This field specifies the certificate association data to be matched. These bytes are either raw data or the hash of the raw data, depending on the *matching type* field.

The KXOVER protocol uses these TLSA records in order to perform KDC-to-KDC authentication. This authentication process will be explained later on.

Chapter 3

Design

The goal of this thesis is to automate Cross-Realm Authentication. In order to do so, we will be using mechanisms that already exist within Kerberos, and these mechanisms will be slightly modified for our purposes.
The main idea is to replicate the authentication process that takes place between a client and its Key Distribution Center. However, this authentication process will now take place between two remote KDCs.
Since the final goal of the protocol is to be applied Internet-wide, we have to assume that the connection between the two KDCs will be done through the Internet. We are also assuming that the two KDCs have no prior knowledge of each other. This means that the normal authentication process is not suitable here. Instead, we will be using the Public Key Cryptography protocol extension PKINIT. We will specifically be using the Elliptic Curve extension to PKINIT [10] to cryptographically sign an Elliptic Curve Diffie-Hellman key exchange between the KDCs to agree on an authenticated shared secret.

All communication between KDCs is done through the Internet. In order to authenticate each party, the protocol will rely on DNS records secured with DNSSEC, in relation to the certificates provided inside the native Kerberos messages.

The resulting protocol that we have developed has been named **Kerberos Realm Crossover**, which from now on will be abreviated as **KXOVER**.

3.1 Setup

In order for our protocol to work, some prior setup has to be done. This is so because the protocol relies on DNS records in order to do some of its main tasks.
This means that the administrators of the Kerberos realms involved in the cross-realm communication need to add some information to their DNS zones.
The DNS records being used by the protocol are the following.

- **TXT**. This record is used to obtain the realm name of the server we are trying to access. A client could be requesting a server either by its domain name, or by its host name. The administrators of the realm will need to add records to DNS for all the public

servers inside the realm that they wish to make available through KXOVER. The use of TXT records to declare Kerberos realm names has been introduced by Rick van Rein in [17]. The structure of the TXT record is the following.

```
_kerberos.server IN TXT realmName
```

- **SRV**. This record is used to obtain the address and the port of the KDC itself. These address and port are the ones that will be used to send all messages to the remote KDC. The structure of the SRV record is the following.

```
_kerberos._protocol.realmName IN SRV priority weight port target
```

- **TLSA**. The last record used by the protocol is a TLSA record. This record is used to obtain a certificate for the remote KDC. This is the basis of the authentication process, and it will be explained on Section 5.2. Since we want to allow self-signed certificates, the Certificate Usage field is set to 3. The selector field is set to 0, because we are checking a full certificate, instead of a *SubjectPublicKeyInfo*. The matching type field depends on how the data is being stored, which can be either raw data, in which case the matching type would be set to 0, or the hash of the raw data, in which case the matching type could be set to 1 or 2. The port and realm name on this record are the same ones that can be found in the SRV and TXT records respectively. The structure of the TLSA record is the following.

```
_port._realmName IN TLSA 3 0 matching data
```

All DNS records used for this protocol have to be secured using DNSSEC. As it has been said before, this ensures the validity of the records, and eliminates some possible attacks like DNS spoofing. This type of attacks will be argued in Section 5.4.

Another thing that the administrator of the KDC needs to do before running the protocol is to setup a principal in the database that will manage the KXOVER interaction. This principal has to be named *kxover/admin@REALM-NAME*.
This principal needs to have admin privileges on the Kerberos database, since it will be the one creating the cross-over principals in the database during the execution of our protocol.
After creating the principal in the database, a keytab file for the principal needs to be created. This keytab file will allow the automatic authentication of the *kxover* principal.

3.2 KXOVER Protocol

The protocol has been designed with security in mind. However, our second goal when designing the protocol was to avoid requiring to modify the clients or services of a realm. With our protocol design, only the KDC needs to be modified, which allows us to add our protocol to Kerberos without needing to modify all client applications that use Kerberos.
The MIT Kerberos implementation that we are using has an unsuitable structure for long

remote queries,which would result in the whole process being blocked. That is the reason why all the logic of the protocol is being done in an external process (daemon) instead of in the KDC itself. With this design, the KDC only needs to redirect messages to the daemon when they are related to Realm Crossover.

The workflow of the protocol can be seen in Figure 3.1.

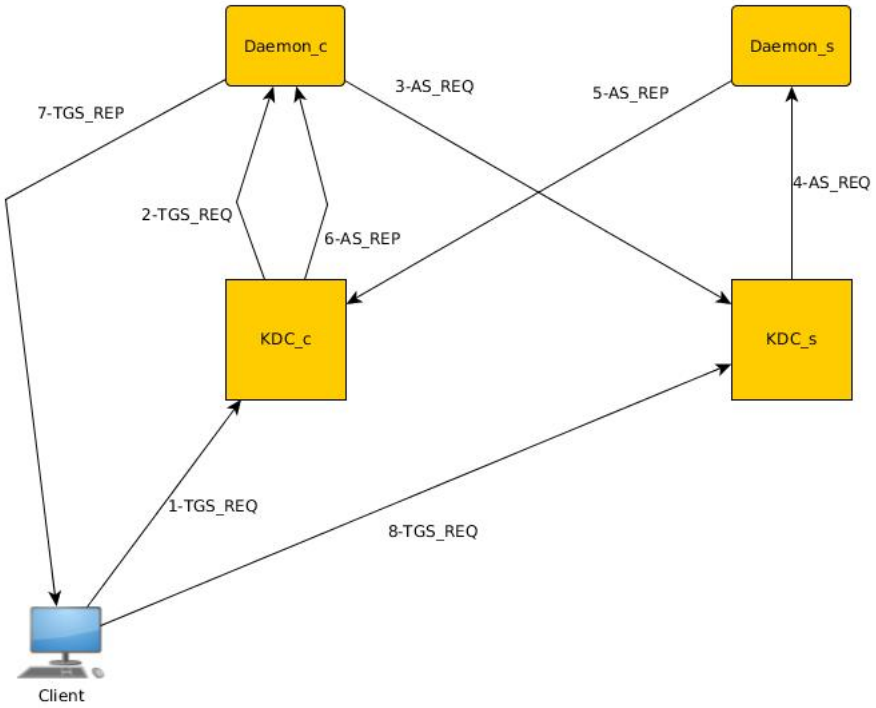


Figure 3.1: KXOVER

1. When a Client wants to request a service, it sends a TGS-REQ to its KDC, from now on, KDC_c.

2. When KDC_c receives a TGS-REQ for a service that is not in one of its realms, it searches the database for a *krbtgt* principal for the remote realm. When it does not find said principal, it redirects the TGS-REQ to the KXOVER daemon, from now on, $Daemon_c$.

3. When $Daemon_c$ receives a TGS-REQ, it extracts the server name from the request, and uses it to lookup the realm name. When the daemon has the realm name, it obtains the location of the remote KDC.
   When the daemon has the location of the remote KDC, it generates an AS-REQ message, and sends it to the remote KDC, from now on, KDC_s. The request incorporates all the necessary PKINIT initialization parameters. The request also has a distinctive marker that identifies it as a KXOVER message.

4. When KDC_s receives an AS-REQ, it checks whether it is part of the KXOVER protocol or not. If the message comes from a remote KDC instead of a client in its own realm, it redirects the AS-REQ to the KXOVER daemon, from now on, $Daemon_s$.

5. When $Daemon_s$ receives the AS-REQ, it first retrieves the realm name from the request and looks up the initiating KDC's location. With that location, the remote daemon looks up for the server certificate of the initiating KDC, and uses it to validate it against the PKINIT certificate of the KDC.
   Once the client KDC has been validated, the daemon obtains the Diffie-Hellman information from the PKINIT data, and uses it to generate the shared secret. Then it uses the shared secret to create a *krbtgt* in the database. After creating the principal, it generates an AS-REP and sends it to KDC_c.

6. When KDC_c receives the AS-REP, it redirects it to $Daemon_c$.

7. When $Daemon_c$ receives the AS-REP, it first extracts the PKINIT information from it. Then, it validates the KDC_s certificate using a TLSA look-up. After validating the certificate, it generates the shared secret from the Diffie-Hellman information of the remote KDC. Using the shared secret, it creates the *krbtgt* principal in the database as well.
   Finally, the $Daemon_c$ Generates and sends a TGS-REP to the Client, providing the Ticket Granting Ticket for the remote realm.

8. Finally, the client can use the cross-realm Ticket Granting Ticket to generate a TGS-REQ directed to the remote KDC, requesting again for the service.

3.2.1 Message Specification

In KXOVER, we are using custom-made messages, based mostly on the AS-REQ and AS-REP. However, some modifications have been made, mainly by removing some unnecessary and undesirable parts of the message that do not contribute to our protocol, like the ticket itself. Our messages can be compared with the standard Kerberos messages, shown on Listings 2.1 and 2.2 from Section 2.1.

In Listing 3.1, the custom AS-REQ message, renamed as KX-AS-REQ, can be seen. The initial part of the message does not change. However, the *req-body* has been changed. That is the reason why it has been renamed to KX-REQ-BODY
The *kdc-options* field has been removed, because the exchange will not produce a ticket. This means that no ticket options need to be used in the exchange. The encryption types field *etype* has also been removed for the same reason. Since no ticket is being generated through the exchange, no encryption type describing the encryption of the ticket is needed. Similarly, the *rtime* field has also been removed since it indicates the maximum lifetime of the ticket in the case of it being a renewed ticket.
The *enc-authorization-data* and *additional-tickets* fields have been removed because they are only used in the TGS exchange.

Since the addresses being used in the KXOVER exchange are obtained via DNS look-ups, the *addresses* field is also not needed in the KXOVER request.

The usage of the fields in our custom made AS-REQ messages is the following.

- **pvno**. This field specifies the Kerberos protocol version number. The current Kerberos version is version 5.

- **msg-type**. This field specifies the message type. Since we are using a custom made AS-REQ message, the msg-type is KRB_AS_REQ, which is represented with the number 10.

- **padata**. This field contains the pre-authentication data. In our case it includes our custom message type, and all the PKINIT information that we are using for the Elliptic Curve Diffie-Hellman exchange. Both contents will be explained later on.

- **req-body**. This field contains all the rest of the fields related to the request itself.

- **cname**. This field specifies the client name of the client performing the authentication. In KXOVER, it specifies the name of the *kxover* principal. This is the principal that has access to the database and creates the principals shared with the remote KDCs.

- **realm**. This field specifies the realm of the client performing the authentication. The usage is the same in KXOVER, where it specifies the name of the realm initiating the KXOVER exchange.

- **sname**. This field specifies the name of the service the request is directed to. In KXOVER, it specifies the *kxover* principal of the remote realm that is being contacted.

- **till**. This field contains the expiration date requested by the client in a ticket request. In KXOVER, it contains the expiration date of the cross-over principal created in the database. After that time has passed, the principal has to be removed from the database, and another KXOVER exchange needs to happen.

- **nonce**. This field contains a random number generated by the client. The same number has to be included in the encrypted response from the KDC, it provides evidence that the response is fresh and has not been replayed by an attacker. In KXOVER, the nonce has to be sent back using the PKINIT signed data structure.

```
KX-AS-REQ           ::= SEQUENCE {
        -- NOTE: first tag is [1], not [0]
        pvno             [1] INTEGER (5) ,
        msg-type         [2] INTEGER (10 -- AS -- ),
        padata           [3] SEQUENCE OF PA-DATA,
        req-body         [4] KX-REQ-BODY
}

KX-REQ-BODY     ::= SEQUENCE {
        cname            [1] PrincipalName,
        realm            [2] Realm,
                             -- Server's realm
        sname            [3] PrincipalName,
        till             [5] KerberosTime,
        nonce            [7] UInt32
}
```

Listing 3.1: KXOVER AS-REQ message specification

The reply message being used by KXOVER has also been modified. The specification of our custom-made AS-REP messages, named KX-AS-REP can be seen in Listing 3.2. This modification is relevant, because KXOVER is not using the reply as it was originally intended. Both the AS-REP and the TGS-REP are used to provide the client with a ticket. However, in KXOVER we do not send a ticket with our reply.
For this reason, both the *ticket* and the *enc-part* fields of the original reply are not present in our custom-made KXOVER replies.

The usage of the fields of the custom-made AS-REP message is the following.

- **pvno**.This field specifies the Kerberos protocol version number. The current Kerberos version is version 5.

- **msg-type**. This field specifies the message type. Since we are using a custom made AS-REP message, the msg-type is KRB_AS_REP, which is represented with the number 11.

- **padata**. This field contains the pre-authentication data. In our case it includes our custom message type, and all the PKINIT information that we are using for the Elliptic Curve Diffie-Hellman exchange. Both contents will be explained later on.

- **crealm**. This field contains the name of the realm in which the client is registered and in which initial authentication took place. In KXOVER, we use it to specify the realm name of the remote realm. Note that this usage is the opposite as the intended usage for this field.

- **cname**. This field contains the name part of the client's principal identifier. In KXOVER, it contains the name of the *kxover* principal that belongs to the initiating realm.

```
KX-AS-REP           ::= SEQUENCE {
        pvno                [0] INTEGER (5),
        msg-type            [1] INTEGER (11 -- AS --),
        padata              [2] SEQUENCE OF PA-DATA,
        crealm              [3] Realm,
        cname               [4] PrincipalName
}
```

Listing 3.2: KXOVER AS-REP message specification

As we have just seen, our custom messages are using the same identifiers as the original AS exchange messages. Therefore, the KDC receiving these messages will identify them as AS-REQ or AS-REP messages.
However, we need to provide a way for KDCs to detect that the AS-REQ comes from a remote KDC instead of from one of its clients.
This is done with the use of the *padata* field. This field is the same one that contains all the information used by PKINIT to complete the Diffie-Hellman Exchange.
In order to identify the messages related to the KXOVER protocol, a padata-type and padata-value pair will be added to both the AS-REQ and AS-REP messages.
A 32-bit integer will be allocated as the type for PA-KXOVER, and it will be used as the padata-value.
As mentioned in Section 2.3, the padata-type used by KXOVER has to be assigned by an expert from IANA. Temporarily, the protocol is using one of the unassigned values for the padata-type element.

3.2.2 Diffie-Hellman Exchange

The KXOVER protocol uses Elliptic Curve Diffie-Hellman key exchange to generate a shared secret. This exchange is done using a modified version of the Kerberos plugin PKINIT.
PKINIT provides options for both Diffie-Hellman key exchange, and public key encryption. For KXOVER, the optional public key encryption part of the messages is dropped, making the resulting specification simpler than the original one.
In Listing 3.3 you can see the request message of the PKINIT plugin, after being customized for the protocol needs.
The top-level request has been modified to contain only a signed AuthPack. The original specification also contains two other values, called *trustedCertifiers* and *kdcPkId*. These two fields are related to the Public Key part of PKINIT, and have been removed for KXOVER.
We are using the original AuthPack specification. No modifications were needed. The fields of the AuthPack are the following.

- **pkAuthenticator**. This is used to prove to the receiving party that the requester has recent knowledge of its signing key. This is done by providing a checksum over the KDC-REQ-BODY sequence.

- **clientPublicValue**. This is used to transmit the Elliptic Curve information to the receiving party. It uses the *SubjectPublicKeyInfo* type to do so.

- **supportedCMSTypes**. This field specifies the list of CMS algorithm identifiers that are supported in order of decreasing preference.

- **clientDHNonce**. This field is used when one wishes to reuse a previously agreed Diffie-Hellman Key. This nonce must be chosen randomly.

The EC Diffie-Hellman public key is mapped to the *subjectPublicValue* field of the *SubjectPublicKeyInfo* sequence. The *SubjectPublicKeyInfo* type has not been modified, the fields are the original ones and the use of those fields is also the original one. The *algorithm* field allows to specify the *ECParameters*, which represent the curve the algorithm is working with. The public key, represented by a point in the Elliptic Curve, is transmitted using the *subjectPublicKey* field.

```
PA-PK-KX-AS-REQ ::= SEQUENCE {
  signedAuthPack        [0] IMPLICIT OCTET STRING
}

AuthPack ::= SEQUENCE {
  pkAuthenticator       [0] PKAuthenticator,
  clientPublicValue     [1] SubjectPublicKeyInfo OPTIONAL,
  supportedCMSTypes     [2] SEQUENCE OF AlgorithmIdentifier OPTIONAL,
  clientDHNonce         [3] DHNonce OPTIONAL,
  ...
}

PKAuthenticator ::= SEQUENCE {
  cusec                 [0] INTEGER (0..999999),
  ctime                 [1] KerberosTime,
  nonce                 [2] INTEGER (0..4294967295),
  paChecksum            [3] OCTET STRING OPTIONAL,
  ...
}

SubjectPublicKeyInfo  ::= SEQUENCE {
  algorithm             AlgorithmIdentifier,
  subjectPublicKey      BIT STRING
}

AlgorithmIdentifier   ::= SEQUENCE {
  algorithm             OBJECT IDENTIFIER,
  parameters            ECParameters OPTIONAL
}

ECParameters ::= CHOICE {
  namedCurve            OBJECT IDENTIFIER
  implicitCurve         NULL
  specifiedCurve        SpecifiedECDomain
}
```

Listing 3.3: Custom PKINIT ASN.1 specification for Request

The reply message used in the protocol is much simpler than the request. This is so because the PKINIT protocol was designed to be an exchange between a client and its KDC. In that case, the KDC is assumed to be an already secured and trusted party. In the case of a KDC-to-KDC communication, this assumption is not completely correct, and should be dealt with later on. This will be discussed later on with the creation of a new message type.

The customized reply message of the PKINIT plugin has only one main component instead of two. Therefore, the type has changed from a CHOICE to a SEQUENCE and the *encKeyPack* field has been dropped. The *dhInfo* field contains the signed sequence that contains the ECDH public key, and an optional nonce that is only present if and only if *dhKeyExpiration* is present.
The *KDCDHKeyInfo* type contains the *subjectPublicKey*, which is the Diffie-Hellman public value, the *nonce* that was present inside the *pkAuthenticator* (only if the DH keys are not reused, otherwise the nonce value is 0), and the *dhKeyExpiration* which specifies the expiration time in the case that the DH keys are reused.

```
PA-PK-KX-AS-REP ::= SEQUENCE {
   dhInfo                  [0] DHRepInfo
}

DHRepInfo ::= SEQUENCE {
   dhSignedData            [0] IMPLICIT OCTET STRING,
   serverDHNonce           [1] DHNonce OPTIONAL,
   ...
}

KDCDHKeyInfo ::= SEQUENCE {
   subjectPublicKey        [0] BIT STRING,
   nonce                   [1] INTEGER (0..4294967295)
   dhKeyExpiration         [2] KerberosTime OPTIONAL,
   ...
}
```

Listing 3.4: Custom PKINIT ASN.1 specification for Reply

Both the reply and the request in the PKINIT plugin are signed. This is necessary to guarantee the authenticity of the messages being sent. PKINIT uses a well known method for sending signed messages, called Cryptographic Message Syntax (CMS)[16]. This syntax can be used to sign, authenticate, or encrypt arbitrary message content.
In the case of PKINIT, the content type being used is *SignedData*(3.5).
This content type specifies how to send signed data, and it incorporates a field to store the data, *encapContentInfo*, and a field to store signatures over the data being sent, *signerInfos*. There can be multiple signatures over the same data.
On top of that, the content type allows the specification of certificates, using the *certificates* field. The set of certificates specified in this content type should be sufficient to contain certification paths from a recognized "root" or "top-level certification authority" to each of the signers of the content. Also, the signer's certificate may be included. This allows for self-signed certificates.

This field is useful for KXOVER because it allows to send the certificate that will identify the KDC itself. This certificate is the certificate that can be found in the TLSA records that the protocol looks up. This is the main authentication method of the protocol, and will be discussed in a later chapter of this document.

```
SignedData ::= SEQUENCE {
  version              CMSVersion,
  digestAlgorithms     DigestAlgorithmIdentifiers,
  encapContentInfo     EncapsulatedContentInfo,
  certificates         [0] IMPLICIT CertificateSet OPTIONAL,
  crls                 [1] IMPLICIT RevocationInfoChoices OPTIONAL,
  signerInfos          SignerInfos
}
```

Listing 3.5: CMS SignedData ASN.1 Specification

3.3 Daemon

The main workflow of the KXOVER protocol takes place in an separate daemon instead of in the KDC itself.
The main reason for designing a daemon instead of implementing the protocol in the KDC itself is the fact that the KDC has **read-only** access to the database. Since there is a need to add cross-over principals to the database, an external process is needed to modify the database.
Another reason in favor of doing all the processing in a daemon, is the fact that the protocol contains DNS lookups. DNS lookups are considered slow, compared to the normal execution time of a Kerberos request, taking up to a couple of seconds. Since the KDC's process when it receives a request is synchronous, it would need to wait until the lookups are finished, not doing anything meanwhile. This would slow down the whole process, and be a possible target for Denial of Service attacks.
However, this issue can be resolved by the KDC directly. When launching the KDC, an option can be set that specifies the number of processes that will be listening to the KDC ports and processing the requests in parallel. This transforms the KDC into a multi-threaded process, avoiding the issues mentioned earlier. When using this option, the top level process acts as a supervisor, and the specified number of workers are forked from it. This option would allow the KDC not to block while the KXOVER process is taking place. However, this is not a default option, and the system administrator has to specify it when launching the KDC.

3.3.1 Design

The main objective for the design of the KXOVER daemon was to maximize the functionalities of the daemon, and to minimize the modifications to the KDC. This means that all the workflow of the protocol is done in the daemon, and the KDC basically acts as a proxy, redirecting messages towards the daemon.

The design of the KXOVER daemon was inspired by the design of the KDC itself. Even though the workflow of the whole process bounces between KDCs and daemons, the daemon itself is stateless. This means that the daemon handles each request or reply individually. In order to handle the messages, the daemon uses the same strategy used by the KDC, which is having a dispatcher that receives all messages, and depending on the type of the message, it is redirected to the correct handling functions.

The daemon has been designed as a process running on the same host as the KDC. However, its design, and the *KRB-PRIV* communication mechanisms provided by Kerberos, would allow the daemon to be executed in a different machine than the KDC. This would allow an extra degree of flexibility regarding the KXOVER administration..

On a regular Kerberos setup, the machine hosting the KDC has to be secured against any attackers. Since the KDC has access to the database, compromising it would allow an attacker to have full control over the realm, being able to impersonate anyone.
In the case of the KXOVER daemon, the process has both read and write access to the database, making it as worthy as an entry point as the KDC itself. This means that the machine hosting the daemon needs, at least, the same amount of security measures as the machine hosting the KDC. Since the machine hosting the KDC is already secured, it is a recommended choice for hosting the KXOVER daemon as well.

3.4 Daemon communication

As mentioned in last section, the daemon does not necessarily have to be hosted in the same machine as the KDC.
However, hosting the KXOVER daemon in a different machine needs some additions both in the daemon and the KDC.

If the daemon is hosted in the same machine as the KDC, the communication method can be the simplest possible. No extra security measures need to be taken, since the machine is assumed to be secure.
In this case, the communication between the KDC and the daemon can be done through UNIX Domain Sockets. The messages are sent through the sockets as clear text, with no need for encryption.
Since the machine is also hosting the KDC, an attacker with the ability to read the Socket communication between the daemon and the KDC would also be able to access both the KDC and the database, having complete control over the network.

In the case of the daemon being hosted by a remote machine, the communication method needs to be secured.
In order to do that, there is a method already available within the Kerberos protocol, the **KRB-PRIV** Exchange. This exchange allows clients to send messages requiring confidentiality and the ability to detect modifications done by rogue third parties. Both clients need to share an encryption key in order for this mechanism to work.
This exchange method would allow the KDC to redirect the requests and replies to the daemon in an encrypted way. In order for this to work, the KDC would need to access the secret

key of a principal in the database representing the daemon, and encrypt the message with it. Afterwards, this message could be sent over an insecure network to the daemon, which would be able to decrypt the message using its own copy of the secret key.
Even though this mechanism already exists in Kerberos, it is a message exchange directed to clients, and not to the KDC itself. This means that the KDC would need to be modified in order to encrypt the messages before sending them to the KXOVER daemon.

3.5 Key Distribution Center Modifications

Having the whole workflow of the protocol inside the KDC's code was not desirable. At least a daemon to access the database had to be designed.
Having this in mind, the design process of the protocol tried to achieve the complete opposite. The daemon is in charge of the whole mechanics of KXOVER, and the KDC simply acts as a proxy, redirecting the messages it receives towards the daemon.

As we have seen in Section 3.1, there are three different messages that the KDC has to redirect towards the daemon.

- The first message is the **TGS-REQ**. This is the initiating message of KXOVER. When the KDC receives a request directed to a service from another realm, it first tries to locate the crossover principal in the database. If that check fails, the KDC sends the message to the daemon instead of returning an error. The required modifications for this step are minimal, since no extra checks have to be created.

- The second message is the **AS-REQ**. This is the first message generated by the daemon. This means that the message will include the "PA-KXOVER" padata element. This padata element identifies all messages that are part of the KXOVER protocol. The KDC has to check whether the request has that element or not, and only redirect to the Daemon the messages that have it. Since the KDC receives AS-REQ messages regularly, and the added padata element is following the standards, the needed modifications to the KDC are minimal.

- The final message is the **AS-REP**. The KDC currently never receives AS-REP messages, according to the protocol description. This means that there is no implemented handling functions for this message type. A message handling function should be added, that checks whether the message is related to KXOVER or not, and in the positive case, the KDC redirects the message to the daemon.

Chapter 4

Implementation

In order to prove that the protocol designed in the previous chapter works and to learn about any oversights in the Internet Draft specifying it, a proof-of-concept implementation has been done. A KXOVER daemon has been implemented, and the necessary modifications to the KDC code have been made. This Chapter provides an overview of the proof-of-concept implementation, and discusses some of the challenges and problems that were found while developing it.
The Kerberos implementation used for the development of our proof of concept is the MIT one. The daemon has been named **kxoverd**.

Due to the time constraints of this project, and having in mind that the implementation is merely a proof of concept for the protocol, some parts of the protocol have not been implemented. These parts will be explained in Section 4.3.

Even with some unimplemented features, the proof of concept developed in this thesis has managed to correctly perform a Diffie-Hellman exchange, and to create the cross-over principal in both KDCs. These two features enable realm cross-over, thus proving that the protocol works.

4.1 Overview

As we can see in Figure 4.1, the Sequence Diagram of the implementation closely matches the protocol design described in the previous chapter.

However, there are some small differences that are worth describing.
First of all, we can see that the client KDC returns a NOT_FOUND error at the begining of the diagram. Ideally, the client would receive a TGS-REP message containing the cross-over principal when the protocol finishes its work. However, as we will explain later, including this in our proof-of-concept requires a lot of modifications to the KDC, and we leave it for future work. Instead, our proof-of-concept implementation will return the NOT_FOUND error to the client and it will execute the whole protocol in the background. Finally, when the client sends a second TGS-REQ message, the cross-over principal will already be in the Kerberos Database, and the TGS-REP with the correct cross-over information will be sent to the client. The latency of our proof-of-concept implementation is slightly higher than the Kerberos usual

execution times. This means that making the KDC wait until KXOVER has finished could cause timeouts on the clients. A way of improving the latency of our proof-of-concept would be to precompute a pool of different Diffie-Hellman public keys, and consume one each time a request comes. This would reduce the latency of our implementation while still keeping the forward secrecy provided by Diffie-Hellman.

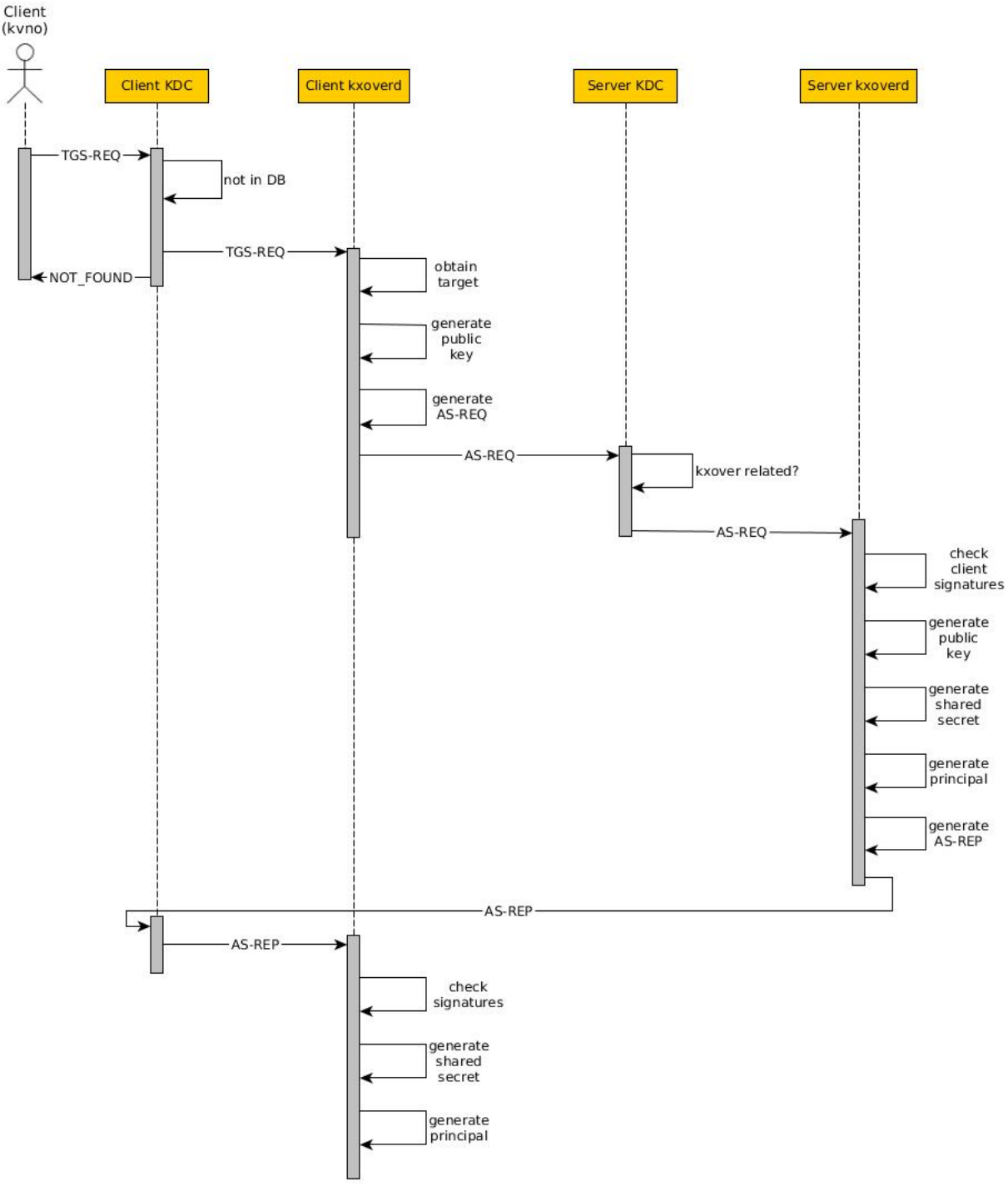


Figure 4.1: KXOVER Protocol Sequence Diagram

4.2 Challenges

There have been several challenges in the implementation process of this protocol.
These challenges needed to be addressed since they were main parts of the protocol, and needed to be added to the proof of concept.

The main challenge encountered in the development process was the difficulty of adapting the message encoders and decoders.
The encoding and decoding functions used by the Kerberos MIT implementation are generated rather than explicitly defined, and adapting them to the needs of our protocol was not considered feasible. For that reason, functions to encode and decode the authentication exchange messages had to be created. A new function had to be created for each process and for each message. Even though the messages are really similar and the functions are basically doing the same process, the different specification of the request and reply messages did not allow the creation of a more general function.

In order to create the functions to encode and decode both the AS-REQ and AS-REP, an ASN.1 library was used, called GNU Libtasn1[12]. This library was used to encode and decode the messages directly. However, the library is not really flexible, and does not implement all of the features of the ASN.1 standard. For exampe, the library does not recognize the OPTIONAL keyword, and requires all elements of the specification to be present in the ASN.1 structure before encoding it. This fact hindered the implementation process, which affected the duration of the whole project.
The problems regarding the ASN.1 library triggered the development of a more suitable library for our purposes. The library was developed by Rick van Rein and it is called Quick DER[19]. There are plans to replace the Libtasn1 library by the Quick DER one, since it is more suited to the needs of the project. However, due to the time constraints of the thesis, this replacement is kept as future work.

4.3 Remote Key Distribution Center Authentication

Due to a lack of time, and the fact that we were building a prototype and not a final product, one of the main features of the protocol was not implemented. That feature is the signing of messages and the verification of the signature and the KDC's certificate. These two steps compose the Authentication process for remote KDCs. Since it is an integral part of the KXOVER protocol, we will explain below how this has to be implemented. On Section 5.2 we will discuss why this method is good enough to provide authentication over the Internet.

As we have seen before in Listing 3.5, KXOVER uses the *SignedData* content type. Both the AS-REQ and AS-REP messages being sent by KXOVER use the same content type, which means that the process of signing and checking will be the same on both sides of the protocol. On top of that, the certificates being used in this step are the same ones that have to be setup in a TLSA record on DNS.

4.3.1 Signing

The signing process is the following.

1. The initiator computes a message digest over the data being sent. In our case, the data being signed is the *AuthPack* element.
2. The message digest is digitally signed using the private key of the signer.
3. The signature value is added to the *SignerInfo* element, as well as the signer's certificate identifier and other signer-specific information.
4. The digest algorithm identifier and the *SignerInfo* value are collected into the *SignedData* value, as well as the content itself (*AuthPack*). The KDC's certificate is added to the *certificates* field, along with any certificates needed to validate it under DANE.

4.3.2 Checking

The checking process is the following.

1. The receiver of the message extracts the signature from the *SignerInfo* value, and the certificate from the *certificates* field.
2. The receiver computes the message digest from the received *AuthPack*.
3. The receiver verifies the signature using the certificate and the computed message digest.
4. The receiver verifies the certificate or certificate chain by checking it against the certificate obtained with the TLSA look-up.

4.4 Key Distribution Center Modifications

Kerberos Version 5 was released many years ago, and it has been updated through the years. This means that the source code is extensive and difficult to follow at first. Understanding the code at a deep enough level to modify it was a challenge in itself.

However, the changes that had to be done to the source code were few, and small. This fact made it easier to finish, and no major problems were found. No new dependencies were introduced, which means that the Makefile did not need to be updated, making the compilation process a lot easier.
The implemented changes are the ones that were planned in the design process. A total of three files were modified, *dispatch.c*, *do_tgs_req.c* and *do_as_req.c*,

- When the KDC receives a TGS-REQ, it checks whether the server being requested is from its realm or not. If the server is from a different realm than the client, the KDC searches for a cross-over principal on the database. If the cross-over principal is found in the database, the Kerberos execution continues. Otherwise, if the principal is not found in the database, we connect to the UNIX Domain Socket, and send the request

from the client through it. After it, the normal execution of Kerberos continues, which means that the KDC sends an error to the client, stating that the requested principal was not found.

- When the KDC receives an AS-REQ, it checks whether the client requesting the authentication is in the database or not. If the client is found, the normal execution of Kerberos continues. Otherwise, if the client is not found in the database, we check whether the request is a KXOVER request or not. To do that, we check the *padata* element introduced to identify all KXOVER messages. If the message is indeed KXOVER-related, we connect to the UNIX Domain Socket, and send the request through it. After that, the KDC continues with its normal execution, and reports the principal not found error.

- When the KDC receives an AS-REP, it usually discards it, reporting an error on the message type. Instead of reporting the error, we connect to the UNIX Domain Socket and we send the reply through it.

As can be seen from the modifications to the KDC, we still report the original errors at the end of our additions to the code. In order to provide the correct ticket, the client has to ask for it again. This is an intentional choice, since the prototype is just a proof-of-concept of the protocol. In order to offer a complete implementation of our protocol, new error types and messages would need to be introduced into Kerberos. This would need to be done with the acceptance of the Kerberos community.

Another aspect to note is that we are not checking whether the AS-REP belongs to KXOVER. This happens because the messages we use are custom-made, and slightly different from the originals. The problem with using custom-made messages is that the KDC's decoder cannot decode our messages. This means that we cannot access the message from the KDC, and have to do all the checking in the daemon instead. This problem could be solved with the introduction of our custom messages as new message types, which will be discussed on Section 7.3.

4.5 Dependencies

We managed to modify the Kerberos code without introducing any new dependencies, which means that the compilation process was not modified at all. This simplified the process greatly, since introducing dependencies in a project of this size can generate a lot of problems.
Regarding the daemon, we have introduced three major dependencies. These dependencies are inherent to the protocol, and need to be resolved in order to implement it. However, the implementation of the daemon is done in such a way that the libraries used could be easily changed by different ones providing the same functionalities.

- **DNSSEC**. The KXOVER protocol requires DNS lookups using DNSSEC. This means that the implementation of the protocol needs a library able to issue DNS lookups, and with the ability of requiring DNSSEC. Concretely, the three DNS lookups described earlier have to be implemented, requiring DNSSEC in each of them. In the prototype

that we developed, the library used to issue this lookups is *getdns*[22]. Getdns is a modern asynchronous DNS API. It offers a modern and flexible way to access DNS security (DNSSEC). The open source implementation is developed and maintained by NLnet Labs, Verisign Labs and No Mountain Software.

- **ASN.1**. Kerberos uses ASN.1 to specify its messages. If we want to craft our own messages, or access the requests and replies redirected by the KDC, an ASN.1 library is needed. The library used for the prototype is *GNU Libtasn1*[12]. It is the library used by GnuTLS, GNU Shishi and some other packages. It was written by Fabio Fiorina, but is currently maintained by Simon Josefsson and Nikos Mavrogiannopoulos. Even though this is the library used in the prototype, it is likely to change in the future.

- **TLS/SSL**. The KXOVER protocol needs a cryptographic library to deal with its main parts. It needs a library capable of performing an Elliptic Curve Diffie-Hellman exchange, and it needs a library capable of signing messages, and checking a certificate against a hash of it. In our prototype, we have used the same library for these two cases. The library we have used is *OpenSSL*[14]. It is not necessary to use the same library for both tasks, but doing so reduces the number of dependencies of the project.

As we have said before, all these dependencies introduced in the prototype can be easily changed thanks to the prototype's design. Changing one of the libraries we have used for a different one offering the same functionality is an easy task, and no modifications to the code need to be done besides the ones regarding the libraries themselves.

Chapter 5

Security Analysis

In this Chapter we will discuss about some security aspects of the KXOVER protocol and the proof-of-concept implementation that we have developed.

The addition of KXOVER to the Kerberos protocol does not modify any of the existing message exchanges. This means that the core features of Kerberos are not modified at all. On top of that, thanks to our design, the clients or the services residing in the realm do not need any modifications either. This means that the cross-over communication between a client and a remote server is not modified by KXOVER, and the current cross-over process is executed as-is.
Since the core features of Kerberos are not modified, the security of the message exchanges is not compromised at all by the introduction of KXOVER. This means that an exhaustive security analysis is not needed for Kerberos after the application of the KXOVER protocol.

There are, however, some issues that need to be addressed regarding security, and even though we are not providing an exhaustive security analysis to Kerberos, we have to analyse the KXOVER exchange, the authentication process for remote KDCs, and some potential attacks to KXOVER.
After that, we will also analyse our proof-of-concept implementation because it has some issues that need to be addressed.

5.1 KXOVER Exchange Analysis

The AS-REQ/AS-REP exchange between the two KDCs is the only non-standard exchange that KXOVER is adding to Kerberos. The main goal of these messages is performing an Elliptic Curve Diffie-Hellman key exchange between two identified and authenticated KDCs. In order to do so, the authentication process needs to happen separately from the Diffie-Hellman key exchange, since Diffie-Hellman does not provide authentication at all. In order to perform authentication, the TLSA record provided by the DNS lookups will be used together with the certificate, or chain of certificates, provided in the PKINIT plugin. The authentication method will be analysed in the next Section.

One of the main benefits of using a Diffie-Hellman key exchange is that, if the public keys used by the exchange are not reused, the exchange provides perfect forward secrecy.

We benefit from this fact, and KXOVER has to generate new keys each time an exchange happens. This way, we provide perfect forward secrecy to the two parts involved on the exchange.

During the KXOVER exchange the Diffie-Hellman shared key is not included in any way in the messages, which means that no information about the key can be leaked. On top of that, the shared key is not used to encrypt anything, it is only stored in a local protected database. This fact is relevant, because there is no way for an attacker to obtain a message encrypted using that key and perform an offline brute-force attack on it.
The only sensitive bit of information being sent during the exchange are the public keys of both KDCs, which do not expose any information about the private keys or the shared secret key generated afterwards.

As said before, KXOVER uses the Elliptic Curve cryptography. A benefit of using Elliptic Curve Diffie-Hellman is that key sizes can be smaller than modular Diffie-Hellman, providing the same security. For example, to achieve the same security as a 112-bit symmetric key, at least a 2048-bit Diffie-Hellman key is needed. On the other side, only a key of around 224 bits is needed if Elliptic Curve is used [3]. This fact reduces the sizes of the messages being exchanged, and potentially reduces the execution times of the shared key generation.

5.2 Certificate Validation

The main part of the KXOVER protocol is the KDC-to-KDC authentication process. As we have said before, we are using a modified PKINIT plugin, which allows the KDCs to exchange a secret key. However, since the PKINIT plugin was designed to authenticate clients to KDCs, it offers functionalities that are useful for our authentication process.

When a client authenticates to its KDC using PKI, the KDC already has information about that client, and can use that information to verify the client with the PKINIT exchange. In our scenario, a KDC does not previously know the other KDC that is trying to authenticate itself, which means that we need another way of obtaining information about that KDC. Our way of obtaining information about a KDC is through DNS. If an administrator can modify DNS records of a domain, it is safe to assume that he has some degree of control over that domain. This fact authenticates that administrator as a legitimate one. KXOVER assumes that only a legitimate administrator of a KDC or the domain hosting it, will be able to modify its DNS records. This assumption requires the DNS records to be validated through DNSSEC.

In order to verify the authenticity of a KDC, we will be using the certificate or its hash provided in the TLSA record, and the certificate and signature provided in the KXOVER message.
First thing the KDC needs to do is verify that the signature sent on the AS-REQ or AS-REP is a valid one. To do so, the certificate of the KDC is also sent in the message. Since only the KDC should be able to sign using that certificate, verifying the signature authenticates that the message comes from the KDC.
Then, to check that the KDC from which the signature comes is the correct one, the certificate

included in the AS-REQ has to be validated against the certificate or its hash obtained as the TLSA record.

If both checks are passed, and under the assumption that only a legitimate administrator can edit DNS records, the KDC is authenticated as the legitimate one.

5.3 Access Control

Authorization in Kerberos is done in the services themselves. However, Kerberos helps by providing a common naming of clients to the whole realm, and only authenticating clients introduced by a Kerberos administrator. This makes access control easier for the services, and it may even be the case where access control is centralized somewhere inside the realm, instead of each service doing its own.

In KXOVER, we have replicated the Kerberos certificate-based authentication process with the goal of authenticating a KDC to another one. With the current design of the protocol, any Kerberos realm with the correct certificates would be able to authenticate itself to another one. In theory, this is not a problem, because as we just mentioned, Kerberos helps with access control, and services need to check whether they know the client trying to access them, even if it comes from a remote realm.
However, the process of setting up a shared secret between two KDCs is time consuming, and limiting it would be desirable.

Creating a traditional access control for KXOVER, where only known realms are allowed, would not be really useful, since the ultimate goal was to avoid the administrators having to contact each other. It would also conflict with the main objective of KXOVER for the ARPA2 project. Instead of the traditional access control, what could be applied is a blacklisting system. In a blacklisting system, the administrator could specify realms, or realms belonging to certain domains, that are not allowed to perform the authentication process with the local KDC.

When a KXOVER request arrives, the first things done by the daemon are looking up the remote realm name and the domain name of the remote KDC. This information would be enough to check whether the KXOVER process can continue or not. The blacklisting of realms could be done in several ways.

- Blocking realms by realm name. A single realm could be blocked, by whatever reason, by blacklisting its realm name.
- Blocking domain names. A single realm, or a realm hierarchy could be blocked, by blacklisting a domain name. In this case, you could block the top domain name and all its sub-domains.
- Blocking by certification path. If a certificate provided by a KDC is not a self-signed certificate, a certification path to a trusted root certificate must be available. In this case, an administrator would be able to block certificates that have a certain untrusted CA in its certification path.

5.4 DNSSEC Analysis

We have repeated several times that all the DNS lookups done in KXOVER have to be done with DNSSEC. We have argued that it is necessary, since it is part of the authentication process used by KXOVER. Now we will show the risks of not using it, by explaining a possible attack to our system in a case where DNSSEC is not enforced.

Let's imagine a scenario where you want to contact a remote KDC using KXOVER. In this scenario, we are assuming that you do not enforce DNSSEC on the DNS lookups. If you don't enforce DNSSEC, you cannot verify that the records were generated by the remote KDC's administrator. In this concrete case, you could be a victim to a DNS spoofing attack. In a DNS spoofing attack, the attacker has modified the DNS records that you access in order to point you to a different location than the one you were expecting. Usually these different locations are malicious, and controlled by the attacker.
In a DNS spoofing attack using KXOVER, an attacker could pair your KDC with a malicious KDC, replicating the original remote KDC that you were trying to contact. Once the KXOVER connection with the malicious KDC has been finished, all your clients will contact the malicious KDC thinking that it is the original one. The attacker could set up replicas of the services that the original remote realm was offering, and use them to steal information from your users.
Another attack that the attacker could do is to set a Man In The Middle (MITM) attack, by contacting the original remote KDC, setting up a cross-over connection with it, and redirecting your clients to the original remote KDC's services. In this case, the attacker would be able to listen on all the communication between your users and the remote services.

There are some differences in the effects of spoofing each DNS record that KXOVER uses. The previous case assumes that all three records have been spoofed. However, spoofing individual records could also lead to problems. These three cases show what would happen if an attacker would spoof a single DNS record type.

- **TXT**. The TXT records hold the realm name of the services that are being looked up. In the case that an attacker spoofs the TXT records, the attacker could point the initiating KDC to a realm name different than the intended one. In this case, when issuing the second lookup, the initiating KDC would request the location of a different KDC, which could be controlled by the attacker.

- **SRV**. The SRV record holds the domain name and the port of the KDC. If an attacker spoofs an SRV record, he could redirect the initiating KDC to another KDC under its control.

- **TLSA**. The TLSA record holds the certificate of the KDC. If an attacker spoofs this record, the certificate on the TLSA record would not match the one being sent by the KDC, and KXOVER would not authenticate the remote KDC.

As we just discussed, attacking only one of the DNS records can already cause some major trouble. Spoofing the two first records would help create a MITM attack, and spoofing the third one would block the protocol from pairing the two KDCs. With the enforcement of

DNSSEC in all the lookups performed by the protocol, we are negating these kinds of attacks altogether.

5.5 Implementation Analysis

We have created a prototype that shows that the designed protocol works. Even though the realms are not verifying each other, we have described how the checks have to be done, and it could be easily implemented in the future.
Even though our goal has been accomplished, there are certain things that need to be analyzed. We are introducing new functionalities to a well-known security protocol, and this could affect its stability. We could also be introducing a potential entry point for attackers to exploit.

One of the main assumptions of our prototype is that the machine hosting the KDC is secured. This is an important assumption, because it gives us a foundation to base a security model on. Our prototype sends the messages between the KDC and the Daemon on the same host in plaintext, without using any encryption mechanisms. Thanks to the assumption we are making, this is not a problem. If the machine hosting the KDC gets compromised by an attacker, the entire Kerberos setup is compromised, and the fact that the KXOVER messages are sent as cleartext is not an issue anymore.

A potential problem of our prototype is the fact that the KDC is not checking the KXOVER messages it receives. In the case of a KXOVER AS-REQ, we redirect the message as soon as the KDC detects it is not in the database. In this case, the KDC has only checked the *message type* of the message, and whether the *client* and *server* fields of the request are null or not. After that, the KDC checks the database and if the client is not found, we check whether the message is KXOVER related or not.
In the case of the KXOVER AS-REP, the KDC does not even have a handling function for this message type, so no checks whatsoever are done by the KDC. Since we cannot check if the message is KXOVER related in the KDC, we directly send it to the daemon.

Another potential problem of the prototype is its vulnerability to Denial of Service (DoS) attacks. The prototype is not multi-threaded, and the whole process takes a significant amount of time to finish, if compared to the normal Kerberos execution times. This means that the daemon itself is a potential target for these kind of attacks.
However, the most successful DoS attack that the daemon could suffer, would need to come from inside of the realm itself. The KXOVER process can only be started by a client of the realm, and most of the workflow of the prototol is done in the initiating side. On top of that, the attacker would be a known client, which means that the KDC would not discard the requests, and would send all of them to the Daemon.
In order to perform an internal DoS attack, a rogue user of the realm would need to request cross-realm tickets continuously, which would saturate the daemon and effectively deny service to all other users of the system. However, in a real-world Kerberos environment, the administrator of the realm would be able to detect this uncommon behaviour of one of the realm users and deal with it, limiting the potential damage.

A DoS attack from a remote client would need to be done from a customized rogue KDC. There could be several options for this attack.

- Sending multiple copies of a correct AS-REQ message. This attack could try to force the KDC to perform the DNS look-ups over and over. However, if the messages are correct, the first exchange will succeed in creating a cross-over principal. In our prototype, this situation is not checked, but it would be easily avoidable by just checking whether the cross-over principal already exists or not, and sending a message to the initiating KDC directly, without issuing any DNS look-ups.

- Sending multiple copies of incorrect AS-REQ messages. This attack would be a refinement of the previous one, forcing the KDC to perform the DNS look-ups and not allowing it to create a cross-over principal in the database. This attack would be a bit more complex to detect, but keeping track of the requests and limiting them when they come from the same realm would avoid most of the damage of this attack.

- Sending several different AS-REQ messages. This would be the equivalent of a Distributed Denial of Service attack. This would require the attacker to control a network of KDCs. In this case, our prototype would be completely vulnerable. A mitigation to this problem would be upgrading the daemon to a multi-threaded implementation. Another security mechanism to prevent this would be the master-slave mechanism available in Kerberos. When setting up Kerberos in a real-world environment, it is recommended to set up several slave KDCs, to ensure the availability of the system. In our case, the KXOVER daemon would also need to be replicated in the slave KDCs. However, this solution is not inherent to the daemon itself.

Chapter 6

Related Work

In this chapter we will talk about some projects that are related to our thesis, but that are not directly influencing it.

6.1 TLS-KDH

TLS-KDH is another sub-project of ARPA2[2]. It is currently being developed by a Kerckhoffs Institute student, and even though it is orthogonal to KXOVER, both projects enhance each other's potential enormously..
As the name points out, this project is separated into two different parts, KDH and TLS.

First, KDH aims to pair Perfect Forward Secrecy with Kerberos. It does that by adding Diffie-Hellman to Kerberos. This is interesting because Kerberos provides authentication without providing Perfect Forward Secrecy, whereas Diffie-Hellman provides an attractive encryption solution without authentication properties. In order to do that, it adds a Diffie-Hellman key exchange to the AP-REQ and AP-REP messages. From that moment onward, instead of using the session key provided by the KDC, the two parts communicating will use the shared key generated through the Diffie-Hellman exchange. This way, not even the KDC could intercept the messages being shared between the two parties, and the DH exchanges provides Perfect Forward Secrecy.

As we said before, Kerberos is a widely used protocol. However, integration of Kerberos with some Internet protocols, like HTTP, is not really good. On top of that, the security of Kerberos over HTTP is especially weak. This is where the second part of the project comes in. It intends to integrate KDH into TLS. One of the options to do that could be to do Kerberos over TLS directly. However, it fails to provide Perfect Forward Secrecy. Specifically, once a user's credential would be guessed, all his past TLS interactions would be at risk of decrypting. TLS-KDH aims at incorporating the KDH idea directly into the TLS implementation.

The main goal of this project is to provide the benefits of Kerberos authentication over HTTPS, which is an Internet protocol. This could be achieved inside the same realm with the current implementation of Kerberos. However, it gets interesting when a client wants to access an HTTPS server that resides outside of his realm. Here comes in play the KXOVER protocol that we have designed, since it would allow automatic authentication to remote

realms.

6.2 PKCROSS

PKCROSS[13] is a draft proposed in 2014 that also aims at performing automatic realm crossover. However, the starting point of PKCROSS differs from KXOVER.

The idea behind PKCROSS is that the clients have to perform the cross-over authentication themselves. In order to do so, PKCROSS uses a kx509 service that has to run inside the client realm that generates certificates that identify the user. After obtaining this certificate, the client asks the remote TGS directly, using a PKINIT exchange and the client certificate that it just obtained.

The main difference between PKCROSS and KXOVER is that in PKCROSS, each user needs to support client-driven PKCROSS, and the exchange with the remote realm happens each time a clients needs a TGT. With KXOVER, the exchange happens only once, between the two KDCs, which means that the clients do not need any modifications, and multiple clients can reuse the crossover principal once it is created.

6.3 Pseudonimity Support for Kerberos

KXOVER allows to connect to previously unencountered remote realms. It might happen that the client does not trust those realms regarding their privacy, or the client simply does not want to give away its login information to a remote party. In this case, Kerberos offers anonymity[11], which completely conceals the client's principal name and possibly also its realm. However, that might not always be the best option.

In an Internet-Draft presented by Rick van Rein [18], the concept of pseudonymity is introduced. With pseudonymity, instead of completely concealing the principal name of a user, a pseudonym is used. Using a pseudonym would conceal the real identity of the client, and it would allow the remote realm to distinguish return visits. The remote realm would treat the pseudonym as a normal client identity, but that identity would not match with the login name of the client. On the other hand, the client might use specific pseudonyms for each remote realm, or set up online identities for certain types or groups of remote realms.

Another advantage of using pseudonyms instead of the client's login identity is the ability to act in behalf of another entity for which the user has been authorised. This can be done by replacing the client's identity by that of a group or role. This would be helpful to remote realms because they would not need to know which users are allowed to do a certain task, they would only need to recognize a certain role.
As an example, a realm might use an online pseudonym for everyone working in a certain department. In this case, the remote realm would know they are contacting with the correct department, but not know the specific person they are communicating with.

Chapter 7

Future Work

7.1 Implementation Improvements

As we mentioned in Chapter 4, the implementation of KXOVER made during this thesis is merely a proof-of-concept to show that the protocol works as it is intended to. Being a proof-of-concept piece of software, it lacks several parts that need to be added to the implementation in order to adhere to the protocol completely.
This improvements are left as future work, but an explanation for each of them is provided.

- **KDC validation**. As said in Section 4.3, the signing and validating of remote KDCs was not implemented for the proof-of-concept. This is a core concept of KXOVER and has to be the first addition as future work.

- **Client response**. The current implementation of KXOVER, with the current modifications to the KDC, does not reply to the client directly with a positive answer, instead, it replies with a negative answer and expects the client to request again in order to obtain the positive reply. The implementation needs to be changed in order to reply positively to the first client request. This will require more modification to the KDC and a speed-up of the KXOVER Daemon.

- **Multi-threading**. An important modification to the Daemon would be the introduction of multithreading. This allow to scale the Daemon to bigger realms with more users. It would also provide some slight DoS protection.

- **Concurrency**. Ordering the actions taken by the Daemon differently would allow for a speedup on the whole process, limiting the appearance of timeouts.

- **Exception handling**. The proof-of-concept implementation shows that the protocol works by responding to the normal execution of KXOVER. It needs to be expanded with special cases, error cases, and a correct handling of errors and exceptions.

- **Timeouts**. The Daemon needs to be able to react to timeouts, since the clients requesting the cross-over TGT have a timeout of around 1 second. The whole process

could take more than 1 second to finish due to the DNS queries, which means that the clients should be notified about it.

- **Key Management**. The cross-over keys established by KXOVER have a limited lifetime. After that lifetime has finished, they cannot be used anymore and the key has to be re-established. This re-establishment of keys affects all the clients and services that were using them, and needs to be dealt with properly. On top of that, keys could be revoked by one of the KDCs, which would disallow all communications with the other realm.

7.2 Request for Comments

In order to present a new Internet protocol to the community, it first needs to be completely laid out and specified in an Internet Draft. Internet Drafts are documents published by the Internet Engineering Task Force (IETF) containing preliminary technical specifications, results of networking-related research, or other technical information.
Internet Drafts are considered as work in progress documents, and their goal is to be published as Request for Comments (RFC). An RFC can eventually become an Internet Standard.

In order to publish an RFC, a draft has to be published first. Internet Drafts can be published either by IETF working groups or by individuals. An Internet Draft has a validity of six months, after which they have to be updated, or published as an RFC.

The KXOVER protocol will be written into an Internet Draft, and it will be submitted to the Kitten WG of the IETF in order to be reviewed. This Working Group deals with new authentication technology, including Kerberos. One of the goals of this thesis is to show that the protocol works, and it helpin the writing of the next iteration of the draft itself.
After the draft is published, if it generates enough interest in a IETF working group, an RFC will be written and published.

7.3 New message type

As mentioned in Section 3.2.1, the messages being sent by KXOVER are custom-made modifications of the AS-REQ and AS-REP messages, both having custom-made PKINIT extensions. We also mentioned that the official encoders and decoders provided by the MIT implementation of Kerberos do not work with our messages.
This facts started the idea of creating new message types for KXOVER, separate from the current Kerberos messages.

Relying on plugins that were not designed for the purpose we are giving them is usually not a good idea. Since those protocols were not specifically designed for the usage we are giving them, problems may arise in the future. Creating a new message type would allow a finer control over the protocol. The messages would only contain the data necessary to perform the authentication between KDCs, instead of hacking the messages aimed at client-KDC authentication.

Furthermore, the PKINIT exchange was specifically designed to allow client-KDC authentication, and creating the new message type would benefit KXOVER by only having the options related to KDC-to-KDC authentication. So, we believe that creating new message types to allow KDC-to-KDC authentication would make Cross-Realm authentication easier to implement and more robust.

However, Kerberos is a well-known and established protocol, and adding a message type to it is not an easy task. First of all, the protocol would need to be perfectly defined, and accepted by the Kerberos community. Then, all different implementations of Kerberos would need to build their own version of the KXOVER protocol, following the accepted protocol specification.

This future goal is not only beyond the scope of this thesis, but it is also a goal that has to be achieved with the help and acceptance of all the Kerberos community.

Chapter 8

Conclusions

In this thesis we have developed a protocol that allows Kerberos to setup realm crossover automatically. Thanks to this protocol, the administrator's interaction is no longer needed. The protocol, which we called Kerberos Realm Crossover (KXOVER), uses DANE to authenticate an Elliptic Curve Diffie-Hellman key exchange in order to authenticate the KDCs to each other and to create a shared realm cross-over key between them.
The mechanics of the protocol have been shown to work in a proof-of-concept solution. Our solution works with the MIT Kerberos implementation, which has been modified to suit our needs. We have created a separate daemon that handles the protocol itself, and it is capable of creating a cross-over principal on both sides of the exchange.
With our prototype we have shown that our protocol works, and that it could be added to Kerberos.

In order to add our protocol to Kerberos, it first has to be completely specified, and a draft has to be written about it. Then this draft will be presented to the Kerberos community in order to discuss its benefits and weaknesses, and maybe to discuss its implementation as a protocol extension.
Since Kerberos is a widely-used and well-known protocol, the relevance of this thesis could increase significantly if our design is accepted as a Kerberos extension. However, this could take some time, since the Kerberos community first has to discuss it.

One of the direct implications of our thesis is the fact that the ARPA2 project relies heavily on the work described in it. This means that the KXOVER protocol will be further improved and detailed, and care will be put on it to make it perfectly secure and usable. In order to add our design to the ARPA2 project, a proper implementation would need to be done, since our prototype is merely a proof-of-concept and it does not include all the features necessary to be used in a real system.

Bibliography

[1] ARPA2. ARPA2 Project website. https://arpa2.net/. Accessed: 03-03-2016.

[2] ARPA2. TLS-KDH project website. http://tls-kdh.arpa2.net/. Accessed: 21-04-2016.

[3] BlueKrypt. Cryptographic Key Length Recommendation. https://www.keylength.com/en/compare/. Accessed: 12-05-2016.

[4] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, MIT, July 2005.

[5] Heimdal. Heimdal Kerberos webpage. https://www.h5l.org/index.html. Accessed: 29-04-2016.

[6] IANA. List of Pre-authentication and Typed Data. http://www.iana.nl/assignments/kerberos-parameters/kerberos-parameters.xhtml#pre-authentication. Accessed: 29-03-2016.

[7] ITU. ASN.1 Project website. http://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx. Accessed: 31-03-2016.

[8] ITU. ITU ASN.1 Introduction. http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx. Accessed: 05-04-2016.

[9] L. Zhu, and B. Tung. Public Key Cryptography for Initial Authentication in Kerberos (PKINIT). RFC 4556, Microsoft Corporation, June 2006.

[10] L. Zhu, K. Jaganathan, and K. Lauter. Elliptic Curve Cryptography (ECC) Support for Public Key Cryptography for Initial Authentication in Kerberos (PKINIT). RFC 5349, Microsoft Corporation, September 2008.

[11] L. Zhu, P. Leach, and S. Hartman. Anonymity Support for Kerberos. RFC 6112, Microsoft Corporation, Painless Security, April 2011.

[12] Libtasn1. GNU Libtasn1 library website. https://www.gnu.org/software/libtasn1/. Accessed: 31-03-2016.

[13] N. Williams. Public Key-Based Kerberos Cross Realm Path Traversal Protocol Using Kerberized Certification Authorities (kx509) and PKINIT. draft, Cryptoconector, October 2014.

[14] OpenSSL. OpenSSL project webpage. https://www.openssl.org/. Accessed: 06-04-2016.

[15] P. Hoffman, and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA. RFC 6698, VPN Consortium and Kirei AB, August 2012.

[16] R. Housley. Cryptographic Message Syntax (CMS). RFC 3852, Vigil Security, July 2004.

[17] R. Van Rein. Declaring Kerberos Realm Names in DNS (_kerberos TXT). I-D, ARPA2.net, March 2016.

[18] R. Van Rein. Pseudonymity Support for Kerberos. I-D, ARPA2.net, April 2016.

[19] R. Van Rein. Quick DER Library Github website. https://github.com/vanrein/quick-der. Accessed: 31-03-2016.

[20] S. Hartman, and L. Zhu. A generalized Framework for Kerberos Pre-Authentication. RFC 6113, Painless Security, Microsoft Corporation, April 2011.

[21] GNU Shishi. Shishi Kerberos webpage. http://www.gnu.org/software/shishi/. Accessed: 29-04-2016.

[22] Unbound Security. Getdns API webpage. https://getdnsapi.net/. Accessed: 06-04-2016.