Department of Mathematics and Computer Science
Security Research Group
&
K3rckhoffs 1nst1tute

Integrating Kerberos into TLS

Master Thesis

T.J.G.M. Vrancken

Supervisors:
Dr. Nicola Zannone (TU/e)
Dr.ir. Rick van Rein (ARPA2)
Dr. Alexander Serebrenik (TU/e)

version 1.1

Eindhoven, August 2016

Abstract

Strong authentication and privacy play a major role in today's digital society. Many protocols and systems have been designed and built but proper integration of the two most common systems, Kerberos and TLS, is still absent. In this thesis we have studied the integration of the Kerberos authentication protocol with TLS for privacy. We analyzed a preliminary specification of a mechanism called TLS-KDH that integrates Kerberos authentication into TLS. Based on this analysis we have proposed some improvements to the original design and built a proof-of-concept to prove that the design actually works. Additionally, a security analyses of the TLS-KDH mechanism has been conducted. It shows that the mechanism is secure if it is deployed under the right conditions. TLS-KDH proves to be a strong alternative for PKI-based authentication. Furthermore, a performance analysis has been conducted that compares the TLS-KDH mechanism with TLS with the X.509 PKI as authentication mechanism. We compared the differences in computational workload based on a theoretical analysis of the designs of both mechanisms. We conclude that TLS-KDH outperforms the X.509 mechanism with a workload factor that lies between 150 and 16000. With our work we have improved the TLS-KDH design and made it one step closer to becoming an Internet Standard. We furthermore showed that it is a secure, efficient and feasible mechanism that elegantly merges the worlds of Kerberos and TLS.

Preface

This work has been written as a master thesis for the Information Security Technology master track offered by the Eindhoven University of Technology (TU/e) and the Kerckhoffs Institute. The master project has been performed as part of the ARPA2 project, which I came across during the "Nationaal Informatica Congres 2014" titled "AnonymIT". The talk of Michiel Leenaars and Rick van Rein appealed very much to me and when Rick said at the end of his performance that the project welcomed graduates I took my chances and applied. Luckily for me there was tons of work to be done and so I was allowed to start my master project as part of the ARPA2 project, under the direct supervision of dr.ir. Rick van Rein, its chief architect. So here I am, a little while later, writing the final chapter of my master thesis or "levenswerk" as Rick likes to call it. Time flies when you are working hard and so this project has come to an end. I have very much enjoyed this project and learnt a lot.

Part of this work was supported with a grant from NLnet foundation, in conjunction with the programme "[Veilig] door innovatie" from the Netherland Center for Cyber Security. I would like to thank them for their support.

I would like to thank dr.ir. Rick van Rein for his daily supervision on behalf of the ARPA2 project. Thank you for your trust in me, your support, your good advise, your lessons in life, your humor and your willingness to help me at the most unbelievable times of day. Thank you for giving me the freedom to plan my own time and for giving me thorough feedback on my work. It is all very much appreciated.

I would like to thank dr. Nicola Zannone for his supervision on behalf of the TU/e. Thank you for giving me the freedom to plan my own time and to manage my own project. Thank you for giving me thorough feedback on my thesis in this short final phase.

I would like to thank dr. Alexander Serebrenik for participating in my graduation committee on such short notice.

Finally, I would like to thank my friends and family for all the moral support and the fun moments that we shared when I was not busy being busy.

T.J.G.M. Vrancken

Eindhoven, August 19, 2016

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Today we live in a world where information technology plays a key role and is omnipresent. We depend more and more on digital systems that support us in our daily lives, replace human tasks or simply make our lives easier. We see a clear shift from a "physical" to a "digital" world. With this shift new design and engineering problems and challenges arise. Digitalization not only brings us "joy and happiness" in the form of new apps, multimedia or smart autonomous cars. It also comes with concerns about privacy, integrity and confidentiality of data. Besides managing their own physical identity, people now have to manage dozens of digital identities as well. Identities linked to among others (web)shops, healthcare services and government services. These identities present access to potentially personal and sensitive information, and should be well protected against misuse.

The benefits of digitalization and digital systems are vast. However, care should be taken when designing such systems. Engineers must not only focus on mere features and functionality but should also take security and privacy into account when designing a proper system. Because of the immense dependency of our society on information technology and digital systems, a breach with respect to security and privacy can have enormous consequences on both individuals and society as a whole. Identity theft, discrimination, failure to perform financial transactions, power outages and flooding are just some examples of what can happen when digital infrastructure is not well protected. This gives rise to questions such as "How can we incorporate good security principles into our systems?", "How can we protect our users from privacy breaches?", "How can we be in control of our data?".

Luckily engineers have not been idle and awareness of the impact and risks of ubiquitous digitalization is slowly pervading the minds of both engineers and end-users. Systems and protocols have been designed, built and put in place to protect us from malicious users of our digital systems and infrastructures. These systems however are far from perfect and work needs to be done to stay up to date. In fact, securing our systems will be a constant arms race against potential attackers. Furthermore, designing new systems will give rise to new security challenges. What complicates matters is that good security and usability are often difficult to unite. Nevertheless, system architects and engineers, cannot neglect the strong need for secure systems.

When we look at information security, the concepts of authenticity and confidentiality play a major role. We often want to know that the party that we are talking to at the other end of the line is actually who he claims to be (authentication). Furthermore, we do not like others eavesdropping on our data and therefore want it to be confidential. We can thus on one hand identify the need for strong authentication mechanisms in order to establish authentic communication between two parties. On the other hand, we need strong cryptographic mechanisms in order to keep our communications confidential to only those parties that we want to grant access. When setting up communications these two concepts are often combined. Communicating entities are first being

authenticated before a secure connection is being set-up. However, proper integration between encryption and authentication mechanisms is not trivial and is usually absent. This gives rise to the main research question addressed in this thesis:

1 *How to design a security solution that combines strong authentication with strong cryptography while maintaining compatibility with existing systems?*

In order to maintain compatibility one should look for systems or mechanisms that are already in the field, preferably widely deployed, and satisfy the properties of being able to provide strong authentication or strong cryptography. There are several network-based authentication systems that are commonly used: TACACS [28], RADIUS [73], Diameter [26] and Kerberos [65]. They are all centrally managed systems based on a client/server architecture. Except for Kerberos, they all implement a so-called AAA protocol. AAA stands for Authentication, Authorization, and Accounting, and represents the primary tasks of such systems. TACACS, RADIUS and Diameter primarily focus on regulating network access (i.e. *device-to-network* authentication or *client-to-network* authentication). Kerberos however, focusses on regulating service access (i.e. client-to-service authentication). It even supports client-to-client authentication, which is useful in the context of a peer-to-peer (P2P) network. Kerberos therefore offers more fine-grained control and flexibility. We will therefore direct our attention to Kerberos. TACACS, RADIUS and Diameter are further discussed in Chapter 7.

Kerberos is a network authentication protocol developed by MIT [59, 64]. It is designed to only facilitate strong authentication, and is therefore not formally ranked among the AAA protocols. Kerberos is however able to pass authorization data along, and can be used as a base for building separate distributed authorization services [63]. Kerberos is designed to authenticate clients to services on a network rather than to authenticate to a network itself. Although, it can also be used as a back-end for RADIUS. It is a mature, secure and well-standardized protocol that is extensively used in industry. Kerberos works both on top of UDP and TCP transport layers. Furthermore, it gives the client a credential (i.e. a Ticket) with which the client can authenticate itself to other services in the realm. The other mechanisms mentioned here only give a "yes" or "no" answer, with respect to the authentication question. Although being really good at authentication, Kerberos lacks some desirable cryptographic properties such as Perfect Forward Secrecy (PFS). Also not all Application Layer communication protocols are well-integrated with Kerberos, most notably HTTP. That makes Kerberos less accessible as a large-scale generic authentication protocol as it potentially could be.

If we look at confidentiality then one of the most widely used cryptographic protocols is Transport Layer Security (TLS). It primarily aims at providing confidentiality and integrity between two communicating parties. TLS is one of the two standardized cryptographic protocols used in industry that operate at the Application Layer (as defined in the Internet Protocol Suite). The other one is SSH (Secure Shell) and has similar capabilities as TLS. They both provide secure network services over an unsecured network. Besides architectural differences, the main difference between the two is the degree of adoption. Compared to SSH, TLS is used more often and more broadly. Another protocol suite that provides network security exists, and is called IPsec [48, 22]. IPsec however, operates on the Internet Layer (as defined in the Internet Protocol Suite) and is by design only capable to provide *host-to-host*, *network-to-network* or *network-to-host* security. It cannot provide *process-to-process* security like SSH and TLS. We therefore direct our attention to TLS. TLS is able to provide strong cryptography by means of various strong ciphers and provides in PFS by implementing Diffie-Hellman key exchange methods. TLS supports several authentication mechanisms which are primarily based on certificates and public-key infrastructures (PKI). While getting the job done they do however have some drawbacks. First of all, they use long-lived credentials which require extra verification and additional mechanisms for revocation. Secondly, because of the hierarchical nature of the PKI, verification of these certificates is more involved. As quoted from the ARPA2 project website [1] these two issues are solved by Kerberos

by "using short-lived tickets for potentially long-lasting identities, instead of long-lived identities for potentially short-lived certainties.".

What we can conclude from this is that if we are able to combine both Kerberos and TLS, that we can benefit from the advantages and strengths of both mechanisms while overcoming their weaknesses. Because both systems are widely used we would maintain compatibility with existing systems and even widen the field of application for both mechanisms. A preliminary design of a mechanism that integrates Kerberos into TLS has been created under the ARPA2 project [1].

The ARPA2 project is an umbrella project that aims at developing tools and systems that contribute to a decentralized global Internet that offers security and privacy by design. An Internet where its users are well-protected and in charge of their own data. The project hosts many subprojects, varying from early research to full-blown designs and systems. Their TLS-KDH project is a first attempt to properly integrate Kerberos into TLS. It aims at combining the best of both worlds. Using the strong authentication properties of Kerberos with its mutual authentication and single sign-on capabilities, with strong cryptographic properties of TLS with Diffie-Hellman for PFS.

In this work we study the integration between Kerberos and TLS and, in particular, the TLS-KDH mechanism developed within the ARPA2 project. In this respect, the main research question can be refined in three subquestions:

1.1 *Is it possible to design a system that combines the strong authentication properties of Kerberos with strong cryptographic properties of TLS?*

First, we analyze the draft specification of TLS-KDH [70]. We investigate how the two protocols are bound together, and how their artefacts are integrated. Based on this analysis we propose some improvements to the original design. Finally, we have built a proof-of-concept to prove that the mechanism actually works.

1.2 *Security analysis of TLS-KDH. What are its weak and strong points?*

Because TLS and Kerberos are security protocols we perform a security analysis of the TLS-KDH mechanism. We discuss the most important security issues for Kerberos and TLS separately. Then, we investigate whether the TLS-KDH mechanism introduces new attack vectors, or solves any of the known problems for Kerberos or TLS.

1.3 *How does TLS-KDH perform with respect to X.509?*

The most common mode of operation for TLS is currently TLS with the X.509 PKI for authentication. TLS-KDH serves as an alternative authentication method. Because it is also certificate-based (i.e. Kerberos uses Tickets as credentials) we compare the performance of these two mechanisms. This analysis will compare the computational complexity of TLS' traditional X.509 authentication mechanism with the new KDH mechanism. That means that we look at the number of cryptographic operations in the execution flow of both mechanisms. These numbers are then related to a workload factor that we found in the literature.

The remainder of this document is structured as follows:

Chapter 2 introduces the reader into some basic security concepts that are needed in order to understand the discussed protocols and mechanisms. Furthermore it describes the fundamentals of Kerberos and TLS, the mechanisms that are be combined in this thesis. When describing the concepts in this chapter we will, for simplicity, abstract as much as possible from details that are not needed for the reader to understand the TLS-KDH mechanism.

Chapter 3 describes the preliminary design of the TLS-KDH mechanism. This design is described, analyzed, and finally some design improvements are proposed.

Chapter 4 describes a security analysis of the improved TLS-KDH design.

Chapter 5 describes a performance analysis of the TLS-KDH mechanism compared to the traditional X.509 mechanism of TLS. This performance analysis focuses on the computational complexity of both mechanisms.

Chapter 6 describes how the proof of concept of the TLS-KDH mechanism has been built. It describes what technologies have been used and how the new mechanism can be utilized.

Chapter 7 describes work that is related to the integration Kerberos and TLS.

Chapter 2

Preliminaries

Before any details of the design and further analysis can be discussed, the reader should be familiar with some basic networking and security concepts. Furthermore, a global understanding of Kerberos and TLS is required to understand the proposed design of the TLS-KDH mechanism. This chapter introduces the reader into the concepts and mechanisms needed to understand the remainder of this thesis. If you are already familiar with the concepts in this chapter feel free to skip them and continue reading in Chapter 3.

2.1 Basic concepts and terminology

This section briefly describes some basic computer networking and information security concepts and terminology.

2.1.1 Security properties

Authentication

Authentication is the act of confirming the truth of a certain property of an object claimed by someone or something. This can be about whether an object really stems from the middle ages, or about whether you really graduated within 10 years. Within the field of information security, the term authentication usually refers to the process of verifying one's claimed identity [79]. This can be the identity of a person or a machine. It's about the question "Is a person actually who he claims to be?". This is important because in the digital world you cannot see who is on the other end of the connection. The authenticity of an entity is usually verified by means of one party proving to the other party that it has some information that only they can have. This can be for example a password or a digital certificate.

In literature there is a distinction between weak authentication and strong authentication. Although several (unofficial) definitions exist depending on context and usage, we adhere to the following definition. Weak authentication systems apply the principle of authentication by assertion. They assume that services and machines cannot be compromised or spoofed and that network traffic cannot be monitored. In such systems, passwords or other secrets would be sent in the clear over the network. Strong authentication systems do not disclose secrets on the network and use encryption and cryptographic protocols to hide sensitive data.

Authentication however, should not be confused with authorization which is about determining and verifying an entity's access rights to certain resources.

Confidentiality

In a broad sense confidentiality deals with limiting access to information [85]. Within the scope of information security, confidentiality is about making sure that only the intended receiver(s) is/are able to disclose confidential information. In this thesis, we restrict ourselves further to the context of computer networking. With confidentiality, we mean that only the sender and the intended receiver are able to read the transmitted information [79, 68]. The sender and receiver can be people, machines or processes. Confidentiality is typically reached by access control, encryption or a combination of both.

Integrity

Data integrity is a concept that relates to the assurance of the accuracy and consistency of data. It is relevant for data storage (i.e. databases), as well as data transmission (i.e. computer networking). In this thesis, we will restrict ourselves to the context of computer networking. Here integrity deals with the fact whether data has been modified unwanted. This can be deliberate modifications by an attacker, or accidental modifications because of in-transit data corruption (e.g. packet loss).

Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) is a property of a cryptographic protocol. It is defined as the property that the compromise of a long-term secret or key, does not enable an attacker to obtain past session keys [68]. These session are therefore "forwardly" protected.

2.1.2 Cryptology

Cryptology is the science of secret writing with the purpose of hiding the meaning of a message. It consists of techniques and mechanisms for secure communications in the presence of adversaries. Although often being linked to modern electronic communications cryptology actually dates back to around 2000 B.C. The ancient Egyptians already used secret hieroglyphs to conceal their messages. Cryptology can be divided into two areas: cryptography and cryptanalysis. Cryptography deals with the construction of mechanisms to establish secure communications, whereas cryptanalysis is concerned with breaking such mechanisms [68, 74]. While both areas cover a vast range of topics which cannot all be addressed here, it is important to know that the fundamentals of modern cryptology lie in mathematics. Typically, a mathematical function is applied to some source data to produce corresponding output data that has one or more of the desired security properties (e.g. confidentiality). This mathematical function is usually easy to compute but difficult to reverse without knowing the secret input parameter called the (secret) key. Only the intended sender and receiver(s) of the message should know the key, making it virtually impossible for any eavesdropper to learn the contents of the concealed message.

Cryptography or cryptographic mechanisms form the cornerstone of most security applications. They are used to accomplish, among other things, data confidentiality, data integrity and authentication.

Symmetric vs asymmetric encryption

Within the field of cryptography, encryption plays a key role. Encryption is the process of transforming an input text, called the plaintext, into an output text, called the ciphertext, such that the output text looks random and does not reveal anything about the contents of the input text. Encryption is often used to accomplish confidentiality and is reversible. Recovering the original plaintext from the encrypted cihpertext is called decryption.

When looking at encryption schemes, two distinct types can be identified. They differ in how the encryption key is used. When an encryption algorithm requires the same key for encryption and decryption the algorithm is called symmetric. Before two parties begin communicating securely they agree on a shared secret key. Only entities in possession of this key are able to encrypt and decrypt the exchanged messages. The main problem in this case is how to agree on a secret key without openly communicating it. This often called the key distribution problem.

When an algorithm requires a different key for encryption than for decryption one speaks of an asymmetric algorithm. The latter uses a public key for encryption and a private key for decryption. It is therefore often called a public-key algorithm or public-key encryption. To start communicating securely, one generates a public/private key pair. It is important to know that the keys of such a pair are mathematically bound together (i.e. they belong to each other in a unique way). One keeps the private key for itself (hence the name private key), and publishes the public key (e.g. on your website). The public key can be used to encrypt messages to the owner of the private key; only the party in possession of the private key is able to decrypt these messages. After publishing your public key everyone can send you encrypted messages while you are the only one that can decrypt them. An asymmetric cryptographic algorithm therefore solves the issue of secret key distribution between two communicating parties.

Figure 2.1 depicts the difference between symmetric and asymmetric encryption graphically. In Figure 2.1a the sender encrypts the plaintext with a shared secret key. The recipient uses this same key to decrypt the ciphertext. In Figure 2.1b the sender encrypts the plaintext with the public key of the recipient. The recipient then uses its private key to decrypt the ciphertext.

Digital signatures

Asymmetric encryption schemes have another application. They can be used to create digital signatures. If you reverse the way the keys are used you can generate an output that belongs to your private key. In other words, if you encrypt a text with your private key then the whole world can decrypt it with your public key (because it is public). Since you are the only one with this unique private key, the generated output can be uniquely bound to you. This is the key principle of a digital signature. It should be noted however, that it is of utmost importance to keep your private key secure. Otherwise, an attacker can pretend to be you by generating signatures with your private key. In practice, since encrypting large documents in order to generate a signature can be cumbersome (see Section 5.1), only the hash[1] of the document is typically signed (i.e. encrypted with the private key). The added benefit of this method is that the receiver can immediately check the integrity of the document by validating this hash.

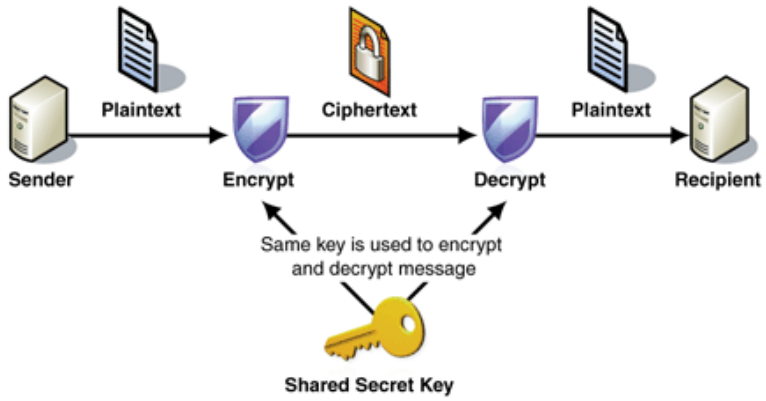
Message Authentication Codes

A message authentication code (MAC) is a, typically short, fixed-length code that is computed over an input text, and can be used to authenticate this input text and detect integrity violations. A MAC algorithm uses a symmetric secret key, only known to the sender and the receiver, to generate an authentication code. MAC algorithms can be constructed from various cryptographic primitives, such as cryptographic hash functions (HMAC) or from block cipher algorithms (e.g. OMAC, CBC-MAC and PMAC).

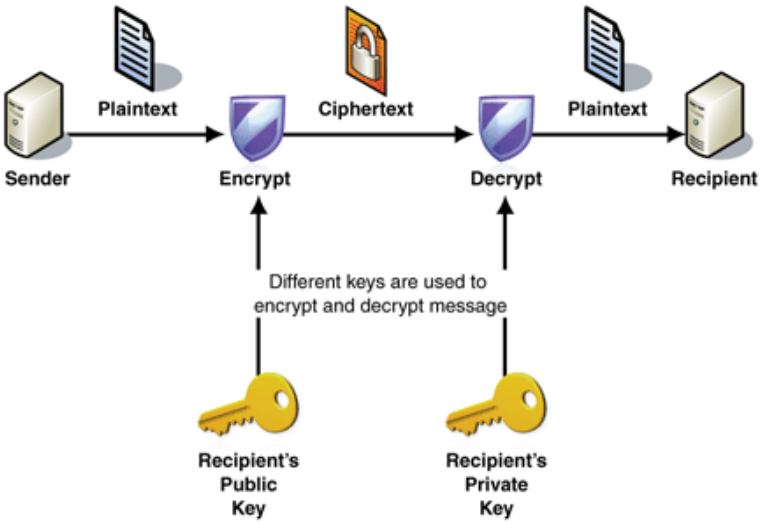
2.1.3 Protocol

Used within the field of computer science the term protocol usually refers to a communication or network protocol. A communication protocol is a set of rules that dictate how two or more entities

[1] A hash can be seen as a unique short-sized fingerprint of a given input.



(a) Symmetric encryption.



(b) Asymmetric encryption.

Figure 2.1: Difference between symmetric and asymmetric encryption.

should exchange messages in order to transmit information. It defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event [50, 84].

Within information security or cryptography, the term protocol is also used to refer to a cryptographic protocol. A cryptographic protocol describes the interactions between two or more entities to achieve one or more security goals (e.g. confidentiality or authentication). It uses cryptographic algorithms as primitives [74, 86]. For example TLS, is such a protocol. A cryptographic protocol that is used to establish the property of authentication is often called an authentication protocol.

2.1.4 Certificate

A certificate or, more accurately, a digital certificate is a digital object that combines (i.e. binds) attributes of its holder. We can distinguish two types of digital certificates: attribute certificates (AC) and public-key certificates (PKC). While the two are much alike, the former is used to characterize its holder whereas the latter is used as a proof of identity. In the remainder of this thesis, when talk about certificates, we mean public-key certificates. A public-key certificate

binds a cryptographic key to some other data. This other data usually refers to or identifies a person, company, machine, process or other type of entity. For example, think about names, addresses, email addresses and website addresses. A certificate is used to prove ownership of a cryptographic key (which is often bound to some form of digital identity). It also contains a digital signature from an entity that claims the authenticity of the certificate, or expresses its trust in the certificate's validity. This could be a trusted third party or the holder itself[2]. These signatures are based on asymmetric cryptography and can be verified by the recipient of the certificate. Besides the verification of the origin, a digital signature also has the cryptographic property that unauthorized changes to the certificate will be detected. It therefore adds a mechanism for data integrity verification and thus a check for whether the certificate has been tampered with [50, 79].

Several types of digital certificates exist that rely on a different trust model. The most widely used certificate standard is X.509 (or PKIX) [79]. Its format is used in for example S/MIME, IP security and SSL/TLS. It is based on asymmetric cryptography and on a hierarchical trust model that uses trusted third parties called Certificate Authorities (CAs) that issue signed certificates. Without delving into all the details of this model it is important for the reader to understand the basic principle of a trust chain. A trust chain, or chain of trust, is obtained when (trusted) root Certificate Authorities issue and sign certificates of other (intermediate) Certificate Authorities, which in turn issue and sign certificates for end-users. There can be any number of intermediate CAs between a root CA and an end-user certificate. Figure 2.2 depicts this architecture graphically. This hierarchical model makes it easy to delegate responsibility and workload with respect to certificate distribution. It also means that, in practice, an end-user is constrained under one trust chain leading to one trusted root.

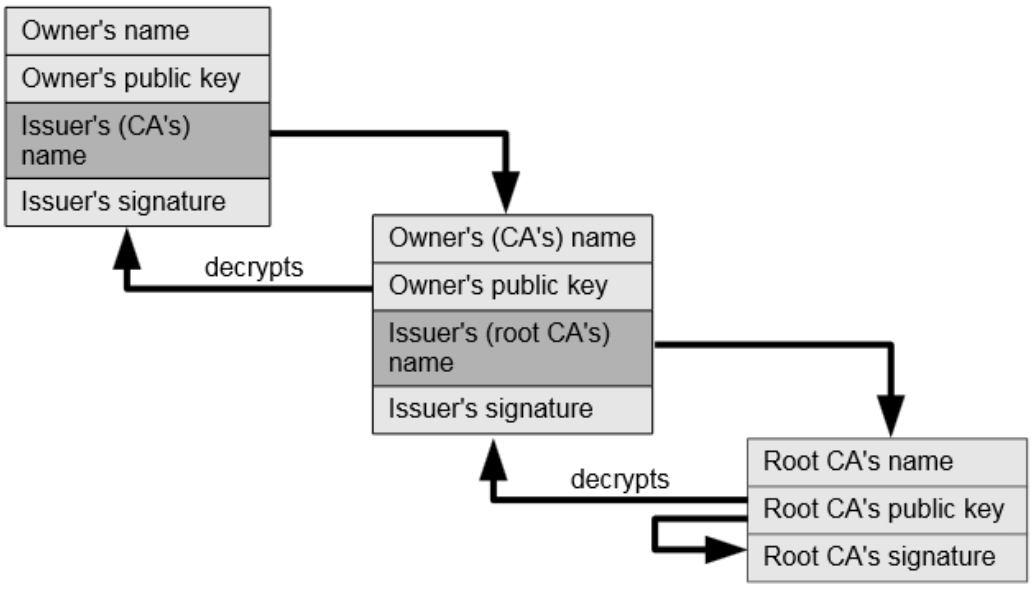


Figure 2.2: Chain of trust [92].

Certificates are widely used for establishing secure connections. Because they rely on asymmetric cryptography they remove the need for two communicating parties to agree on and exchange a common key prior to their communication. Furthermore, they add the possibility to verify the peer's identity. Section 2.1.2 briefly addresses the differences between symmetric and asymmetric cryptography.

[2]The latter is called a self signature and usually doesn't say much about the trustworthiness of the certificate.

2.2 Kerberos

Kerberos is an authentication protocol developed by the MIT to be used over computer networks. It uses the client/server model, symmetric key cryptography and relies on a trusted third party. Kerberos can provide mutual authentication between a client and a server and uses strong cryptography to facilitate authentication, data confidentiality and data integrity. Its name stems from the Greek guard dog of Hades. Since it is an open protocol, several implementations exist. Besides the implementation of MIT [58] itself there is Active Directory from Microsoft [87], Heimdal [37] and GNU Shishi [32]. Kerberos was developed in the 1980's and the first three versions where only used internally. Version 4 was released to the public in the late 1980's and is since then widely used. In the following years several shortcomings where identified and in 1993 version 5 was released with the intention of solving the problems of version 4. At the time of writing Kerberos version 5 (Krb5) [65] is considered to be standard Kerberos [64].

2.2.1 Kerberos architecture

This section gives a brief introduction into Kerberos' architecture. The main artefacts and components are described, which are needed to explain the Kerberos protocol in Section 2.2.2.

Main artefacts

Ticket A Kerberos Ticket can be seen as a specific digital certificate. It contains, among others, the session key for the Client and Application Server with which the Client wants to communicate, the principal's name, the Realm name and an expiration time. A Ticket is used to authenticate a principal to an Application Server (which is the verifier).

Authenticator A Kerberos Authenticator can be seen as a specific digital certificate. It contains, among others, a principal name, a Realm name and a timestamp (see Appendix A). It is used to prove the authenticity of the Client to the Ticket Granting Server or to an Application Server. An Authenticator is created by the Client and verified by the server. Additionally, it can be used to prevent replay attacks (see Section 4.1).

Kerberos components

Client A Client is a process or service that runs on behalf of a specific user (called a principal in Kerberos lingo).

Application Server An Application Server, also called a verifier, is a server that runs a specific application service that Clients might want to use. This can be for example an email service or a print service. Servers are also called principals in Kerberos lingo.

Realm A Realm is the domain for which a specific KDC is responsible. It contains all the Client and Application Server identities (i.e. principals) that the KDC knows and can issue Tickets for.

Key Distribution Center (KDC) The key distribution center forms the core of the Kerberos authentication system. It holds a database[3] with known Client identities and their corresponding cryptographic keys. Similarly, it keeps a list of Application Server identities and corresponding cryptographic keys. The KDC consists of two services:

[3]The original Kerberos design holds a database with principals and their key material. An extension called PKINIT [94] has been developed which facilitates public-key authentication between a Client and the KDC.

Authentication Server (AS) The authentication server is responsible for issuing Ticket Granting Tickets (TGT) to Clients, with which a Client can authenticate itself to the Ticket Granting Server. The TGT, which is valid for a specific amount of time, is used to accomplish single sign-on capabilities.

Ticket Granting Server (TGS) The Ticket Granting Server is responsible for issuing "regular" Application Server Tickets. To that end, a Client can authenticate itself to the Ticket Granting Server with a Ticket Granting Ticket.

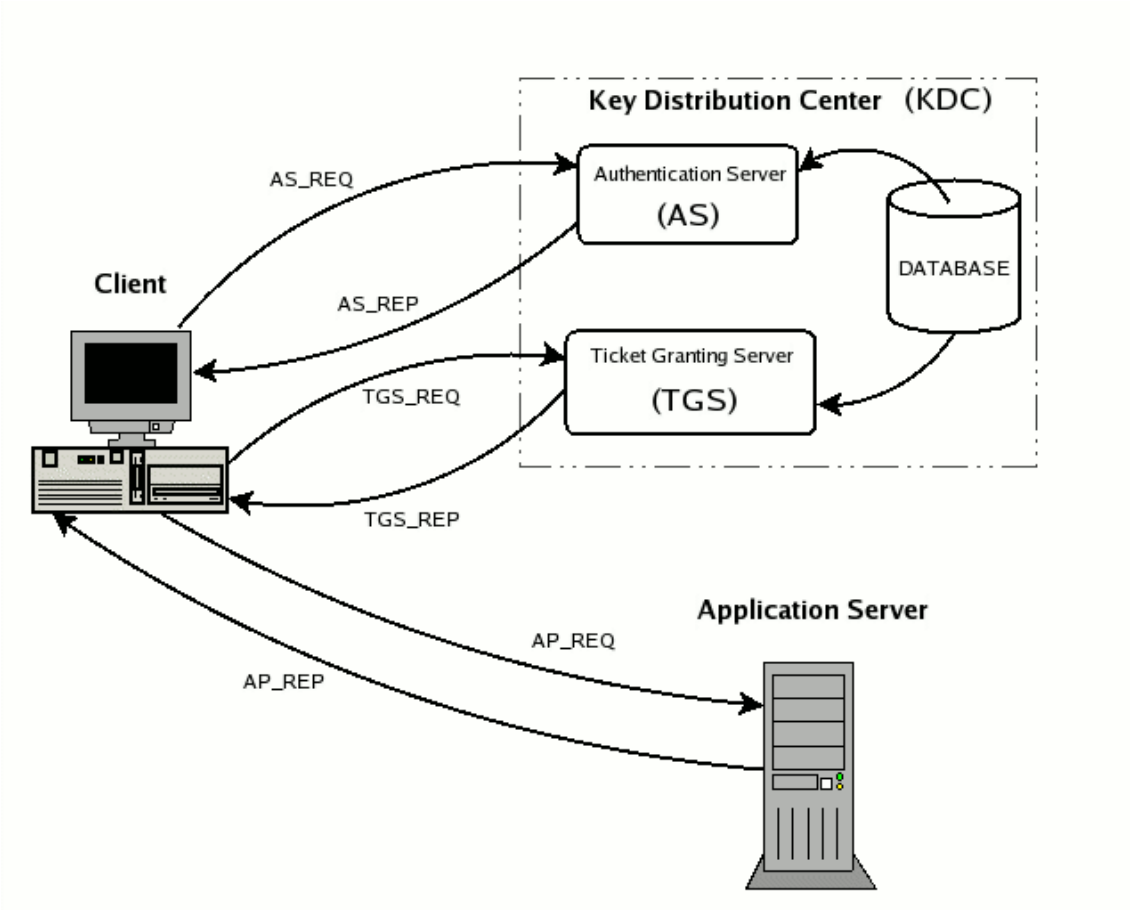


Figure 2.3: Kerberos protocol graphically. (http://www.kerberos.org)

2.2.2 The Kerberos protocol

This section provides a high level overview of the Kerberos authentication protocol. In order to understand the rest of this thesis it is not required for the reader to know all the protocol details. Interested readers are referred to [65] for a full description of the Kerberos protocol.

The Kerberos protocol establishes (mutual) authentication between a Client and an Application Server and relies on a Key Distribution Center (KDC). A Key Distribution Center is responsible for a so called Realm or security domain. A Realm contains all the principals and verifiers that the KDC knows of and can issue Tickets for. Stated otherwise, a KDC controls a Realm and acts as the trusted third party between Clients and Application Servers. Before a principal can be authenticated, it must therefore be registered at the KDC. During registration the KDC and the

principal agree on a shared (long term) secret key. This key is derived from the user's password[4] and bound to a unique principal name with which the user can identify itself. The same principle holds for all verifiers (i.e. Application Servers) in the Realm. The following sections describe the steps in the authentication protocol. It is graphically depicted in Figure 2.3.

Authentication Request

In a typical scenario, a Client wishes to use a specific service offered by an Application Server. The Client therefore needs to authenticate itself to this server. In order to do so the Client issues an authentication request (`AS_REQ`) to the Authentication Server. This `AS_REQ` message contains a principal name, an expiration time (i.e. the time that the Client wants to stay logged in) and a random number. Upon receipt the AS looks up the principal name and checks whether the supplied identity exists. If it exists it looks up the encryption key that corresponds to the principal name and generates an authentication reply (`AS_REP`) message. This message contains:

1. A *Client* $\leftrightarrow$ *Ticket Granting Server* session key. This session key will be encrypted with the Client's key. That is the key that is derived from the principal's password and was stored at the KDC during registration. Only the Client is now able to decrypt this message.
2. A Ticket Granting Ticket that will be encrypted with the Ticket Granting Server's key. This Ticket contains the principal name, the requested validity time and a copy of the *Client* $\leftrightarrow$ *TGS* session key and can only be decrypted by the Ticket Granting Server. The Client is therefore unable to modify it without this being detected.

On receipt the Client is able to decrypt the *Client* $\leftrightarrow$ *TGS* session key from the `AS_REP` message. The Ticket Granting Ticket however can not be decrypted but must be passed on to the TGS in its received form in order to request Application Server Tickets.

Ticket Granting Server Exchange

The Client now has a Ticket Granting Ticket with which it can request Application Server Tickets from the Ticket Granting Server. A TGT has a typical life span of a day[5] which enables the user to request Tickets without typing in its password every time a service is requested. An Application Server Ticket, or just for short Ticket, is used by a Client to authenticate itself to an Application Server (the verifier). If requested by the Client the verifier can also authenticate itself to the Client. In that case mutual authentication has been established. To authenticate to an Application Server the Client first requests a Ticket at the TGS via a `TGS_REQ` message. It uses its TGT for that. A `TGS_REQ` message contains the same data as the `AS_REQ` message plus an Application Server name, the TGT and an Authenticator.

The Application Server name is needed in order for the TGS to know what kind of Ticket must be issued to the Client. The TGT and Authenticator are used to authenticate the Client to the TGS. The TGT can only be decrypted by the TGS since it was encrypted by the AS with the TGS's secret key. The TGT contains information about the Client and the *Client* $\leftrightarrow$ *TGS* session key. The Client now proves its authenticity by sending an Authenticator to the TGS that is encrypted with the *Client* $\leftrightarrow$ *TGS* key. This key was issued by the AS which is trusted by the TGS and therefore the circle of trust is complete. The TGS now verifies the Authenticator and replies with a `TGS_REP` message if everything is allright.

The `TGS_REP` message contains:

[4] Kerberos was originally designed around the use of a user password. Another mechanism using public-key cryptography has also been developed [94] and mitigates some of the problems that arise from password logins.

[5] This can be negotiated during the authentication request, and represents the "single sign-on period". A typical validity time is 8 hours [64].

1. A *Client* ↔ *Application Server* Session Key. This key will be encrypted with the *Client* ↔ *TGS* session key and can therefore be only be decrypted by the Client (and the KDC).
2. A Ticket with which the Client can authenticate itself to the Application Server. It contains information about the Client, a validity time and a copy of the *Client* ↔ *Application Server* session key. This Ticket is encrypted with the Application Server's secret key. The Client is therefore unable to decrypt or modify the Ticket. All fields in the Ticket that are of interest to the Client, are sent as unencrypted copies to the Client.

Application Request

Recall that the main goal of the Client is to use a specific application service. To be able to do so it has to authenticate itself to the Application Server. At this point in the authentication process the Client possesses a *Client* ↔ *Application Server* session key and a Ticket. The Client now proceeds with the authentication by sending an `AP_REQ` message to the Application Server. This message contains a Ticket and an Authenticator. The Ticket was previously obtained from the Ticket Granting Server and the Authenticator is generated by the Client itself. Since the Ticket is encrypted with the secret key of the Application Server it cannot be altered by the Client. The Ticket contains, among other things, a *Client* ↔ *Application Server* session key that was issued by the TGS. The Client also got this key directly from the TGS. The Client now uses this key to encrypt the Authenticator that will be sent to the Application Server.

On receipt of the `AP_REQ` message, the Application Server tries to decrypt the Ticket in order to retrieve the *Client* ↔ *Application Server* session key. If it succeeds, it uses this key to decrypt the Authenticator. If that works, and optionally the checksum inside the Authenticator can be verified, then the Client has successfully authenticated itself to the Application Server. Access to further services or resources can be provided by the Application Server.

If the Client requires mutual authentication, then the Application Server must respond with a `AP_REP` message. This message contains the timestamp from the Authenticator and will be encrypted with the *Client* ↔ *Application Server* session key. On receipt of the `AP_REP` message, the Client tries to decrypt the message with the session key and verifies the timestamp. If the timestamp matches the one that the Client has encapsulated in the Authenticator, then the Application Server has successfully authenticated itself to the Client. Mutual authentication has been established and normal client/server interaction can start.

Overview of Kerberos protocol messages

This section gives a formal overview of the Kerberos protocol messages. There has been abstracted from detailed message contents. Important here to see is the nesting of the session keys, and the keys used to encrypt Tickets and Authenticators. In the description of the messages, K depict cryptographic keys. The subscripts indicate whose key it is, e.g. K_{tgs} is the key of the Ticket Granting Server and $K_{\{c,tgs\}}$ is the session key between the Client and TGS. The subscripts of the Authenticator and Ticket messages indicate their origin and intended recipient. Finally, c represents a Client and v represents a verifier (i.e. an Application Server).

AS_REQ: $c, tgs, time_{exp}, nonce.$

AS_REP: $\{K_{\{c,tgs\}}\}K_c, \{Ticket_{\{c,tgs\}}\}K_{tgs}$

TGS_REQ: $\{Authenticator_{\{c,tgs\}}\}K_{\{c,tgs\}}, \{Ticket_{\{c,tgs\}}\}K_{tgs}$

TGS_REP: $\{K_{\{c,v\}}\}K_{\{c,tgs\}}, \{Ticket_{\{c,v\}}\}K_v$

AP_REQ: $\{Ticket_{\{c,v\}}\}K_v, \{Authenticator_{\{c,v\}}\}K_{\{c,v\}}$

AP_REP: $\{timestamp\}K_{\{c,v\}}$

2.3 TLS

Transport Layer Security (TLS) [21] is a cryptographic protocol standardized by the Internet Engineering Task Force (IETF) and supersedes its now insecure predecessor SSL. TLS aims at providing secure communications between two endpoints that want to communicate over an insecure medium such as the Internet. Its primary security goals are data confidentiality and data integrity. These goals are established by creating an encrypted tunnel between two endpoints so that all exchanged messages remain confidential. Furthermore, it applies message authentication codes (MACs) to ensure data integrity. TLS was originally developed around the public-key infrastructure (PKI) standard X.509 [16, 93] to incorporate authentication, but has since been extended with other authentication mechanisms [55, 77]. TLS is commonly used to "upgrade" a plaintext connection or protocol to a secure one. Common examples of such protocols are HTTP [72], FTP [29], IMAP [66], POP3 [66], ACAP [66] and SMTP [38].

The TLS protocol can be divided into three layers or sub protocols: the record protocol, the handshake protocol and the alert protocol [21] (see Figure 2.4). In the following sections we will give an overview of these protocols. The handshake protocol will be described more extensively because it is needed for the reader to be able to understand the TLS-KDH design.

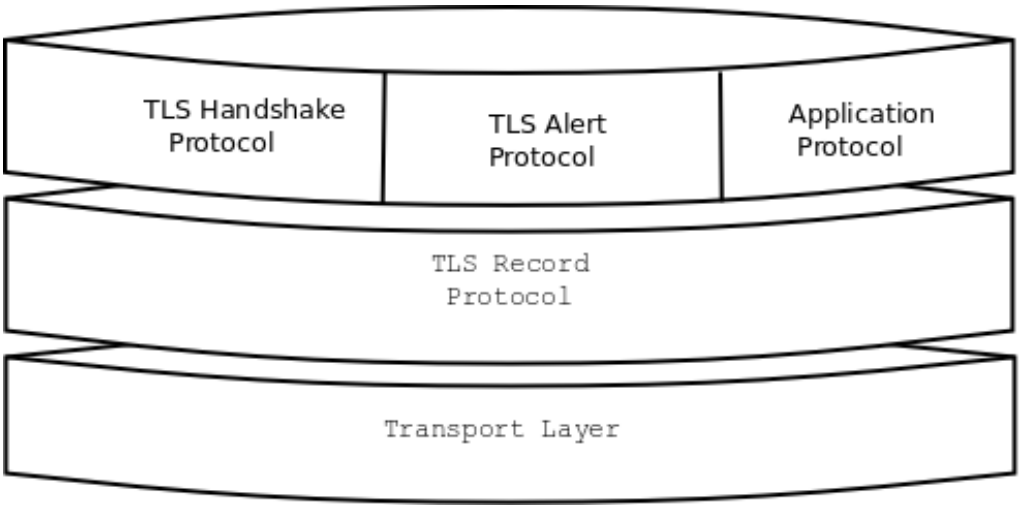


Figure 2.4: TLS internal protocol layers. (from the GnuTLS manual [35])

2.3.1 TLS record protocol

The record protocol is responsible for the actual encryption and transmission of the data. It fragments the data into blocks of the appropriate size, optionally compresses the data, applies a MAC, encrypts, and transmits the result. On the receiving end the data is decrypted, verified, optionally decompressed, reassembled, and then delivered to higher-level layers or processes. It goes beyond the scope of this thesis to discuss all the details of the record protocol. The interested reader can find them in [21]. It is however important to know that the record protocol receives its parameters or "settings" from the handshake protocol. They are called the connection state and include the compression algorithm, an encryption algorithm, a MAC algorithm and all corresponding keys. For the remainder of this thesis, it suffices to acquire a more in-depth understanding of the handshake protocol, which will be discussed in the next section.

2.3.2 TLS handshake protocol

This section describes the TLS handshake protocol. The TLS-KDH mechanism described in Chapter 3 follows the same message flow as a regular TLS handshake, but introduces changes to the message contents. We will therefore describe the flow and the contents of these messages, in order for the reader to be able to understand the differences and extensions that are introduced by the TLS-KDH design.

The handshake protocol is responsible for negotiating the session parameters and operates on top of the record protocol. Before two parties can start communicating over a secure connection, they have to agree on the cryptographic algorithms that are going to be used. Not every client uses the same TLS library version or supports the same cryptographic algorithms. Some clients might allow legacy algorithms for compatibility reasons while others might restrict themselves to more up-to-date or secure algorithms. Therefore, before a secure connection can be established both parties have to negotiate the session's security parameters, and exchange credentials and keys. This is where the handshake protocol comes in. It is worth to mention that initially[6] all handshake messages are sent in the clear. Because we are negotiating encryption parameters, all communication up to the point of agreement (see Section 2.3.2) is sent unencrypted.

TLS uses the client/server model. That means that a TLS-capable server is waiting for a TLS-capable client to initiate a connection. A TLS handshake is performed as follows. A schematic representation of the handshake message flow is depicted in Figure 2.5.

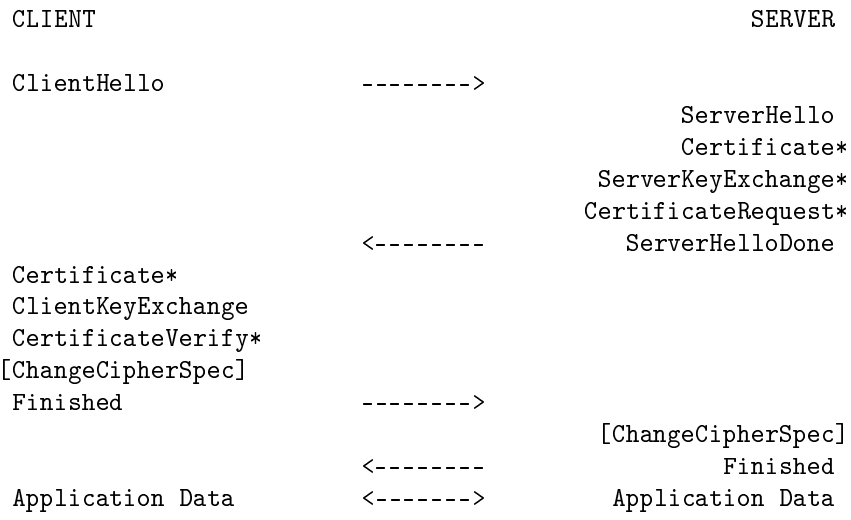


Figure 2.5: Schematic representation of a full TLS handshake. Messages denoted with a * are optional.

Client Hello

The client always first sends a `Client Hello` message. The purpose of this message is to let the server know which cryptographic algorithms the client is capable of using during the session. The `Client Hello` is structured as follows:

[6] All handshake messages are treated by the record layer according to the current connection state. Initially, that state is "unencrypted". This means that the encryption, hash, and compression algorithms are initialized to null. However, it can also occur that during an encrypted session a new handshake will take place. In that case the messages belonging to that rehandshake will be encrypted according to the current connection state.

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-2>;
  CompressionMethod compression_methods<1..2^8-1>;
  select (extensions_present) {
    case false:
      struct {};
    case true:
      Extension extensions<0..2^16-1>;
  };
} ClientHello;
```

Listing 2.1: Client Hello message [21].

client_version The version of the TLS protocol that the client wishes to use during this session.

random Contains 28 bytes of random data generated by the client.

session_id A 32-bit number representing a session ID. This field may be empty if a new session needs to be negotiated, or may contain an existing session ID is the client wants to resume a previously established session.

cipher_suites A list of cipher suite identifiers, sorted in order of preference, that correspond to the cipher suites that the client is willing to use in this session. A cipher suite defines a key exchange algorithm, a bulk encryption algorithm, a MAC algorithm and a Pseudo Random Function (PRF).

compression_methods A list of compression method identifiers, sorted in order of preference, that correspond to the compression methods that the client is willing to use in this session.

extensions This field contains an optional list of extension messages. It can be used to extend the Client Hello message with extra data in order to transmit extra information during the Hello phase. This may be necessary when extending TLS' basic functionality with new functionality [11]. For example the widely used Server Name Indication (SNI) extension [11] and the Signature Algorithms extension[7] [21] use this field. This makes TLS an extensible protocol that can be updated for future needs.

It is important to note that as of TLS 1.2 a Signature Algorithms extension is defined. This extension enables a client to communicate to a server which signature & hash algorithm pairs may be used in digital signatures. This extension may be omitted if the client only supports the default algorithms, but must be present in all other cases. The Signature Algorithms extension adds the following field to the **extensions** list in the `Client Hello`.

```
SignatureAndHashAlgorithm supported_signature_algorithms<2..2^16-2>;
```

supported_signature_algorithms This field contains a list of pairs, where each pair represents a hash and signature algorithm combination (i.e. <HashID, SigID>). The identifiers for these algorithms are maintained by IANA [43].

[7] This extension is actually part of the core TLS 1.2 specification.

Server Hello

When a server receives a `Client Hello`, it first looks whether it has a cached session corresponding to the supplied **session_id**. If it found a match and is willing to resume the session it returns a `Server Hello` message with the same **session_id**. The protocol then continues at the `Finished` messages. If the server could not find a session to resume or does not want to resume the session, a full handshake will be performed. During a full handshake the server will choose the cryptographic algorithms that it is capable of and willing to use during the session from the set of algorithms offered by the client. Since a client only offers algorithms that it is capable of executing, both parties come to a compatible agreement here. The server will always respond with a `Server Hello` message when it was able to find an acceptable set of algorithms. A `Server Hello` message is structured as follows:

```
struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
  select (extensions_present) {
    case false:
      struct {};
    case true:
      Extension extensions<0..2^16-1>;
  };
} ServerHello;
```

Listing 2.2: Server Hello message [21].

server_version This field should contain the highest protocol version that both peers support. For example, if the client offers v1.2 and the server supports this version then the server answers with v1.2. If the client is "newer" and offers v1.2 and the server only supports v1.0, then the server replies with v1.0. The client may the decide whether or not to accept this protocol downgrade and to continue communicating or not. Vice versa when the server is "newer" than the client and the client offers v1.0 and the server supports v1.0 up to v1.2, the server may decide whether or not to continue by replying with v1.0 or terminating with an alert.

random Contains 28 bytes of random data generated by the server. This random must not be derived from the client's random.

session_id A 32-bit identifier for the current session. When a new session is being negotiated the server generates a fresh identifier. When a session is being resumed than the server replies with the session ID of the resumed session. This is the same ID that the client offered in the `Client Hello` with which the client requested a session to resume.

cipher_suite A cipher suite that the server chooses from the list of cipher suites offered by the client. The chosen cipher suite must be supported and allowed by the server. When a session is being resumed than this field contains the cipher suite that was previously negotiated for that session.

compression_method A compression algorithm that the server chooses from the list of compression algorithms offered by the client. The chosen compression algorithm must be supported and allowed by the server. If a session is being resumed then this field contains the compression algorithm that was previously negotiated for that session.

extensions This field is optional and may only contain replies to extensions that where sent by the client.

Certificate

After negotiating a cipher suite during the hello phase it is known to both parties whether authentication is required. If the client and server negotiated one of the anonymous cipher suites then no further authentication is required. In all other cases the server needs to authenticate to the client. Originally, TLS was developed around the X.509 standard. It has been extended however with several other authentication mechanisms such as Pre-shared Keys (PSK) [25, 4] and Secure Remote Passwords (SRP) [80]. In case of traditional X.509 authentication, a server authenticates to a client by sending a digital certificate via a `Certificate` message. Other authentication mechanisms might not, or only partially[8] require to exchange certificates. Certificate authentication has also been extended to support OpenPGP certificates [55], another common cryptographic standard [15].

Client authentication is optional in TLS and can be requested by the server via a `Certificate Request` message. A client is only permitted to send a client certificate when requested by the server. When the client agrees with the authentication request it sends its client certificate to the server via the same type of `Certificate` message. A client may however refuse to authenticate itself. In that case it sends an empty `Certificate` message back to the server. The server may in turn decide whether to continue the session in that case. A `Certificate` message conveys a digital certificate in one of the supported formats and is structured as follows:

```
struct {
  ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

Listing 2.3: Certificate message [21].

certificate_list A certificate chain containing the client's or server's certificate in case of an X.509 certificate type. An OpenPGP certificate or fingerprint in case of an OpenPGP certificate type.

Server Key Exchange

The client and the server need to agree on a Master Secret that will be used for session encryption. One of the inputs for the Master Secret computation is the premaster secret. The premaster secret is either computed on the client and sent (encrypted) to the server, or derived on both sides by executing a Diffie-Hellman (DH) key exchange. Which method is used depends on the negotiated cipher suite. The `Server Key Exchange` message is only sent to the client when the server's `Certificate` message contains not the required data for the client to exchange the premaster secret, or when ephemeral session keys are desired. It typically contains a Diffie-Hellman public key that will be used to derive a DH key that represents the premaster secret, but may also be used to transmit cryptographic data for some other algorithm that aids the establishment of a premaster secret. Additionally, the `Server Key Exchange` is used to transmit cryptographic parameters needed for other authentication methods than certificates [80, 25]. The `Server Key Exchange` message is structured as follows and includes the most common key exchange algorithms:

[8] Here, partially means only from client to server or server to client instead of in both directions simultaneously.

```
struct {
  select (KeyExchangeAlgorithm) {
    case dh_anon:
      ServerDHParams params;
    case dhe_dss:
    case dhe_rsa:
      ServerDHParams params;
      digitally-signed struct {
        opaque client_random[32];
        opaque server_random[32];
        ServerDHParams params;
      } signed_params;
    case rsa:
    case dh_dss:
    case dh_rsa:
      struct {} ; /* message is omitted for rsa, dh_dss, and dh_rsa */
    case ec_diffie_hellman:
      ServerECDHParams params;
      digitally-signed struct {
        opaque sha_hash[sha_size];
      } signed_params;
    case psk:
      opaque psk_identity_hint<0..2^16-1>;
    case srp:
      ServerSRPParams params;
      digitally-signed signed_params;
    /* may be further extended */
  };
} ServerKeyExchange;
```

Listing 2.4: Server Key Exchange message [21, 25, 80, 10].

params Depending on the key exchange algorithm this field contains cryptographic parameters used for the premaster secret computation and/or authentication.

signed_params Depending on the key exchange algorithm this field contains a digital signature over (some of) the exchanged parameters.

Further details about all the data types and sub structures can be found in [21, 25, 80, 10].

Certificate Request

With the `Certificate Request` message, the server asks the client to authenticate by replying with a client certificate. The client is not obliged to do so, e.g. when it has no appropriate certificate. The server may then decide whether to continue the handshake with an unauthenticated client. The `Certificate Request` message may not be sent for anonymous cipher suites, otherwise the handshake will be terminated with a fatal alert.

```
struct {
  ClientCertificateType certificate_types<1..2^8-1>;
  SignatureAndHashAlgorithm supported_signature_algorithms<2^16-1>;
  DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

Listing 2.5: Certificate Request message [21].

certificate_types A list of allowed certificate types that the client may offer in its response.

supported_signature_algorithms A list of signature & hash algorithm pairs, sorted in descending order of preference, that the server is able to verify.

certificate_authorities A list of the distinguished names of acceptable certificate authorities (CA) that are allowed to have signed a certificate with which the client may reply. If the certificate_authorities list in the `Certificate Request` message was non-empty, one of the certificates in the certificate chain should be issued by one of the listed CAs.

Server Hello Done

When the server is finished sending all its messages containing authentication and key exchange information it sends the `Server Hello Done` message to the client. The client then knows that it can proceed with the validation of the received authentication data and can start with his end of the key exchange procedure. The `Server Hello Done` is structured as follows:

```
struct { } ServerHelloDone;
```

Listing 2.6: Server Hello Done message [21].

This message contains no data and simply functions as an end marker.

Client Key Exchange

Contrary to the `Server Key Exchange` message, the `Client Key Exchange` message is always sent by the client. Its goal is to communicate, or facilitate the establishment of, the premaster secret (see Section 2.3.4 for its use). The contents of this message depend on the key exchange algorithm that has been selected. We will only describe the two algorithms that are most common, and that are relevant for this thesis: RSA and Diffie-Hellman.

RSA With the RSA method the client generates a 48-byte long premaster secret, and encrypts it by using the server's public key that was carried in the server's certificate. The encrypted premaster secret is then sent to the server.

Diffie-Hellman With the Diffie-Hellman method both parties communicate Diffie-Hellman parameters such that they both can derive a shared Diffie-Hellman key. This key then represents the premaster secret. Ellictic-curve Diffie-Hellman (ECDH) key exchange algorithms follow the same method.

The `Client Key Exchange` message is structured as follows:

```
struct {
  select (KeyExchangeAlgorithm) {
    case rsa:
      EncryptedPreMasterSecret;
    case dhe_dss:
    case dhe_rsa:
    case dh_dss:
    case dh_rsa:
    case dh_anon:
      ClientDiffieHellmanPublic;
    /* Other extensions*/
  } exchange_keys;
} ClientKeyExchange;
```

Listing 2.7: Client Key Exchange message [21].

EncryptedPreMasterSecret This field contains the RSA encrypted premaster secret that was generated by the client.

ClientDiffieHellmanPublic This field contains the client's Diffie-Hellman public key with which the server can complete the key exchange and derive the premaster secret. If both parties agreed on doing fixed Diffie-Hellman then the client's key was already communicated via the client's certificate. In that case this field must be empty.

Certificate Verify

A `Certificate Verify` will always be sent in conjunction with a client's `Certificate` message if and only if that certificate has signing capabilities[9]. It will be used by the server to verify the validity of the client's certificate. The client's certificate conveys, among other things, the client's public key. With that key the server is able to verify any signature that is set by the client. To prove the validity of the client's certificate the client will sign some data and send it to the server. Because the client and server agree on and share the same data that will be signed, the server can validate the signature and therefore the previously supplied client's certificate.

```
struct {
  digitally-signed struct {
    opaque handshake_messages[handshake_messages_length];
  }
} CertificateVerify;
```

Listing 2.8: Certificate Verify message [21].

handshake_messages This field contains the concatenation of all the exchanged handshake message up to this point in the handshake. The current message is not included. It is important to note that this concatenation is the same for both client and server. The **handshake_messages** are then digitally signed by the client and sent to the server for validation. If the validation fails the server may decide to abort the handshake and to terminate the connection.

Change Cipher Spec

The `Change Cipher Spec` message is used to signal to the other party that the newly negotiated security parameters will henceforth be applied and that the current connection state (see Section 2.3.1) must be replaced with the newly negotiated one. Both client and server send a `Change Cipher Spec` to one another. On receipt, the record protocol will be instructed to immediately change the current read state into the newly negotiated read state. After sending this message the sending party must instruct the record protocol to immediately change the current write state into the newly negotiated write state. Further communication from this point on will continue with the cryptographic algorithms that were negotiated during the handshake. The `Change Cipher Spec` is structured as follows:

```
struct {
  enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

Listing 2.9: Change Cipher Spec message [21].

[9] It is also possible to send a certificate that only contains non-signing key material, such as fixed Diffie-Hellman parameters.

change_cipher_spec This field contains a single byte representing the value 1 and acts as a signal.

The `Change Cipher Spec` message is formally not part of the handshake protocol but has been designated its own protocol[10]. This is done to prevent pipeline stalls. This subtle detail is important to know because `Change Cipher Spec` messages will not be part of **handshake_messages** hashes.

Finished

The `Finished` message is the last message of the handshake protocol and is used to verify the key exchange and authentication steps. It is encrypted under the newly negotiated security parameters and its contents must be validated by both parties. It is important to note that only a legitimate client and server can generate and validate its contents because of the negotiated encryption that has been applied.

The `Finished` message is structured as follows:

```
struct {
  opaque verify_data[verify_data_length];
} Finished;
```

Listing 2.10: Finished message [21].

verify_data This field contains the output of the pseudorandom function (PRF) (see Appendix B). The PRF is applied to the **master_secret**, **finished_label** and the hash of all handshake messages, i.e. PRF(master_secret, finished_label, Hash(handshake_messages)). Here **master_secret** is the master secret that both parties have computed (see Section 2.3.4) and that will be used as a source of entropy for the generation of encryption keys and MAC keys. **finished_label** contains either the string "client finished" or "server finished". Hash(handshake_messages) is the hash computed over all the exchanged handshake messages up to this point[11] excluding the `Finished` message itself.

After successful validation of the `Finished` message the handshake is complete and secure transmission of application data may begin.

2.3.3 TLS alert protocol

The alert protocol facilitates signalling between the communicating peers [21]. It supports two types of alerts: closure alerts and error alerts. Closure alerts are used to signal the other party that no further messages will be send and that the connection can be closed. This type of alert is important in order to prevent truncation attacks [78]. Error alerts are used to communicate error conditions. All messages in the alert protocol contain a description stating what is going on, and an alert level (warning or fatal). When a fatal alert is received the connection must be terminated immediately. A full list of available alert messages can be found in [21] and possible extensions hereof.

[10] This means that it has its own record protocol content type.

[11] The Change Cipher Spec message is not included in this set because it formally belongs to a different protocol [21].

2.3.4 Master Secret computation

The master secret is a 48-byte long shared secret between the two communicating peers. It needs to be negotiated for each session[12] and is used to derive cryptographic keys for the record protocol [21]. The master secret is used as a source of entropy for the generation of encryption keys and MAC keys. It is expanded into a sequence of secure bytes, which is then split into a client-write MAC key, a server-write MAC key, a client-write encryption key, and a server-write encryption key. The master secret computation is based on the premaster secret and the exchanged random values. It is always computed in the same way[13] as stated in Listing 2.11.

```
master_secret = PRF(premaster_secret, "master secret",
                    ClientHello.random + ServerHello.random)[0..47];
```

Listing 2.11: Master Secret computation [21].

[12] It can be restored when a session is resumed.

[13] For security reasons, TLS has been extended with a new master secret computation called the "Extended Master Secret" [9]. For simplicity reasons it will not be discussed here.

Chapter 3

TLS-KDH

As formulated in Research Question 1, we are looking for a mechanism that combines strong authentication capabilities with strong cryptographic capabilities. It should furthermore be compatible with existing systems that are currently used on the Internet. Two systems matching these requirements are Kerberos (described in Section 2.2) and TLS (described in Section 2.3). Both Kerberos and TLS are widely used on the Internet but proper integration of the two mechanisms is unfortunately still missing. Several alternatives have been proposed (see Section 7.2) but each of them has its own shortcomings. Under the ARPA2 project a new mechanism called TLS-KDH has been designed that combines Kerberos with TLS [3]. Its primary goal is to bring the worlds of Kerberos and TLS together, thereby combining the strengths of both mechanisms and broadening their fields of application. More specifically, the strong authentication capabilities of Kerberos will be combined with strong cryptographic capabilities of TLS (Research Question 1.1). TLS-KDH achieves this in the following way:

1. It adds a Kerberos authentication mechanism to TLS,
2. It enables the use of TLS cipher suites to encrypt Kerberos sessions,
3. It adds Perfect Forward Secrecy to Kerberos sessions.

In this chapter we will analyze the draft TLS-KDH specification and propose some improvements to the original design.

3.1 TLS-KDH design

TLS-KDH is an extension of the TLS 1.2 specification and defines two mechanisms for combining Kerberos with TLS. The first one is called KDH-only and uses only Kerberos to achieve mutual authentication between the client and the server. The second one is called KDH-enhanced and uses a hybrid mechanism for authentication. With KDH-enhanced, the server uses traditional certificate-based authentication (i.e. X.509 or OpenPGP) to authenticate itself to the client. The client however uses Kerberos to authenticate itself to the server. Both mechanisms are designed such that they integrate in the standard message flow of TLS. No additional messages needed to be defined, thereby adhering to the core TLS specification. The message flow of TLS-KDH is depicted in Figure 3.2 and will be explained in the following sections.

The TLS-KDH mechanism is designed to protect Kerberos Client to Kerberos Application Server connections. This is depicted graphically in Figure 3.1. During a TLS handshake, a TLS-KDH capable client and server negotiate a cipher suite and authentication mechanism. If they decide to use TLS-KDH, then they negotiate either KDH-only mode or KDH-enhanced mode.

```
      CLIENT                                          SERVER

      ClientHello               -------->
                                                  ServerHello
                                               Server Certificate*
                                                ServerKeyExchange
                                               CertificateRequest
                                <--------          ServerHelloDone
      Client Certificate
      ClientKeyExchange
      CertificateVerify
     [ChangeCipherSpec]
      Finished                  -------->
                                                [ChangeCipherSpec]
                                <--------                Finished
      Application Data          <------->        Application Data

    * Indicates that Server Certificate may be empty; it is present in
      KDH-enhanced message flows, and usually empty in KDH-only message
      flows.
    [] Indicates that ChangeCipherSpec is an independent TLS protocol
      content type; it is not actually a TLS handshake message.
```

Figure 3.2: TLS-KDH message flow.

In this section we will describe the design of TLS-KDH. First the two modes of operation, KDH-only and KDH-enhanced, are described. Then, a TLS `Hello` message extension called Ticket Request Flags is described. It is used to exchange extra information between a client and a server during the handshake. Finally, we describe how Kerberos Authenticators are used as substitutes for digital signatures, and how TLS-KDH encodes Kerberos Tickets in X.509 certificates.

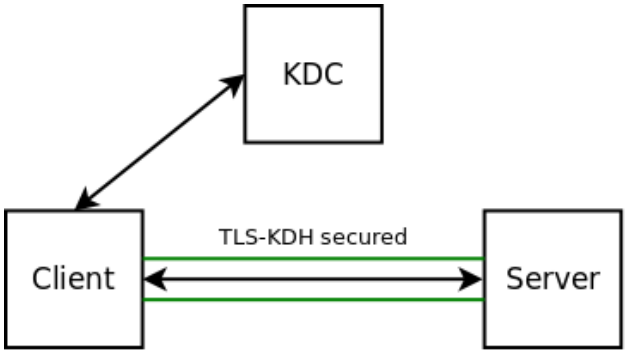


Figure 3.1: TLS-KDH secured channel.

3.1.1 KDH-only

KDH-only is one of the two modes of operation defined by TLS-KDH. It only relies on Kerberos to provide mutual authentication between a client and a server. KDH-only introduces new cipher suites for TLS. A cipher suite defines a key exchange algorithm, a bulk encryption algorithm, a MAC algorithm and a pseudorandom function. During the handshake, a client and a server agree on a cipher suite to use during the TLS session. The set of cipher suites in TLS therefore defines what combinations of cryptographic algorithms are available. Because Perfect Forward Secrecy is a desired property for a session to have, the cipher suites for KDH-only are restricted to have key exchange algorithms of the ephemeral Diffie-Hellman type (DHE). Because furthermore efficiency is important, the key exchange algorithms are further restricted to the elliptic-curve variants (EC). This results in allowed key exchange algorithms of the type elliptic-curve ephemeral Diffie-Hellman (abbreviated ECDHE). The other cryptographic algorithms that are defined in a KDH-only cipher suite are restricted to only algorithms that are nowadays considered strong, with corresponding key lengths that are sufficient to

provide enough security[1]. The proposed new cipher suites for KDH-only are listed in Table 3.1. They are referred to as the KDH-only cipher suites. TLS cipher suites use the following naming convention. First a "TLS" prefix followed by the name of the key exchange algorithm, then the "WITH" infix followed by the bulk encryption algorithm name and finally the MAC algorithm name. For example, the cipher suite TLS_ECDHE_KDH_WITH_AES_128_GCM_SHA256 defines an elliptic-curve ephemeral Diffie-Hellman key exchange algorithm, an AES-128 block cipher in Galois Counter Mode and finally a SHA256 MAC algorithm.

Cipher suite	verify_data_length (bytes)
TLS_ECDHE_KDH_WITH_AES_128_GCM_SHA256	32
TLS_ECDHE_KDH_WITH_AES_256_GCM_SHA384	48
TLS_ECDHE_KDH_WITH_AES_128_CCM	32
TLS_ECDHE_KDH_WITH_AES_256_CCM	48
TLS_ECDHE_KDH_WITH_AES_128_CCM_8	32
TLS_ECDHE_KDH_WITH_AES_256_CCM_8	48
TLS_ECDHE_KDH_WITH_ARIA_128_GCM_SHA256	32
TLS_ECDHE_KDH_WITH_ARIA_256_GCM_SHA384	48
TLS_ECDHE_KDH_WITH_CAMELLIA_128_GCM_SHA256	32
TLS_ECDHE_KDH_WITH_CAMELLIA_256_GCM_SHA384	48

Table 3.1: KDH-only cipher suites.

KDH-only also introduces a new authentication mechanism that is used in conjunction with these cipher suites. Traditionally authentication takes place via a public-key infrastructure. Certificates are being passed back and forth to prove the authenticity of the server and optionally of the client. Kerberos authentication however relies on symmetric key cryptography. Therefore, no public-key certificates are exchanged. The exchange of a client Ticket and Authenticator suffices to establish mutual authentication between a client and a server (see Section 2.2.2). Both the Kerberos infrastructure and the X.509 infrastructure accomplish authentication, but they do it in a different way. The KDH-only mechanism introduces a new key exchange algorithm for TLS that facilitates Kerberos authentication. It works as follows.

During a TLS handshake, a client and a server express their intent to do KDH-only by agreeing on a KDH-only cipher suite in the `Client Hello` and `Server Hello` messages. Furthermore, both parties must list the `kerberos` signature algorithm (see Section 3.1.4) in the list with **supported_signature_algorithms**. The latter expresses the ability to process a Kerberos Authenticator as a digital signature equivalent (see Section 3.1.4). After the server has sent its `Server Hello` it starts a normal elliptic-curve Diffie-Hellman (ECDHE) key exchange by sending a `Server Key Exchange` message to the client. This message contains no digital signature, as it would with X.509 authentication, because there is no corresponding server certificate to verify it with. The server also sends a `Certificate Request` to signal the client that it requests a Kerberos Ticket. This message must only specify `kerberos_sign`[2] in the **certificate_types** field and must specify `kerberos` based signature & hash pairs in the **supported_signature_algorithms** field. Finally, the server sends its `Server Hello Done`.

On receipt, the client continues the handshake by responding with a `Client Certificate` message. This message conveys the client's Kerberos Ticket. Secondly, the client completes the Diffie-Hellman key exchange by sending a `Client Key Exchange` message to the server. The mutually established DH-key serves as one of the inputs for the premaster secret computation. Finally, the client sends a `Certificate Verify` message. This message normally conveys a signed

[1] At least for now and the near future. Security is an ongoing arms race and new developments can lead to new security requirements. Therefore current algorithms and key lengths may be considered weak in the future.

[2] This name is a bit confusing as it suggests a signature type instead of a certificate type. The name is however conforming to IANA naming conventions.

hash that the server is able to verify with the public key of the client that was exchanged via the client's certificate. With Kerberos this is not possible and instead an Authenticator is wrapped inside this message. This Authenticator is important because it also proves the authenticity of the server to the client. If the server is able to decrypt the Authenticator, then apparently it was able to obtain the *client* $\leftrightarrow$ *server* session key that was securely wrapped inside the Ticket that the client previously sent (see Section 2.3.2). This Ticket is only decryptable by an authentic server since it was generated and encrypted by a mutually (i.e. by client and server) trusted KDC. The server now proves its authenticity by generating its **verify_data** and sending it via the `Finished` message to the client. This verification data depends on the decrypted contents of the Authenticator and therefore enables the client to see whether the server was able to decrypt it.

After performing a KDH-only handshake the client and the server have established the following:

1. Mutual authentication,
2. An ephemeral Diffie-Hellman session key,
3. A set of cryptographic algorithms to be used during this session.

Premaster secret computation

During a normal Diffie-Hellman key exchange in TLS, the resulting Diffie-Hellman key acts as the premaster secret. KDH-only prescribes a different premaster secret computation for the following reason. For the KDH-only mechanism a way is needed for the client to tell whether the server was able to decrypt the Authenticator. Otherwise authenticity of the server cannot be guaranteed. TLS-KDH solves this problem by embedding a fresh key in the `subkey` field of an Authenticator (see Appendix A). This subkey is then used in the construction of the premaster secret. This premaster secret is used to construct the master secret, and the master secret is used to compute the **verify_data** in the `Finished` message. This `Finished` message thus indirectly depends on knowledge of the subkey in the Authenticator. Being able to decrypt the Authenticator is thus reflected in the resulting `Finished` message. The adapted premaster secret (pms) is constructed as follows:

$$pms = |K_{DH}| \mathbin{+\!\!+} K_{DH} \mathbin{+\!\!+} |K_t| \mathbin{+\!\!+} K_t \mathbin{+\!\!+} |K_a| \mathbin{+\!\!+} K_a$$

Here K_{DH} is the established DH-key, K_t is the session key contained in the Ticket, K_a is the subkey contained in the Authenticator, $|\cdot|$ is the length function[3] that outputs a 16-bit length, and $+\!\!+$ is the concatenation operator.

Extended verify_data length

Because mutual authentication in KDH-only depends on the `Finished` messages, it is necessary to increase the length of the verification data beyond the default length. The **verify_data** in this message must have enough entropy and, since it is based on a keyed hash function [21], be sufficiently large to minimize the chance of a collision. TLS 1.2 specifies a property for this called the verify_data_length. If desired, a cipher suite can define its own verify_data_length value for the PRF it defines[4]. TLS-KDH specifies this for the KDH-only cipher suites. Their respective **verify_data** lengths are listed in the second column of Table 3.1.

[3] The length of an object equals the number of bytes the object is long.
[4] This custom length however must be at least as long as the default of 12 bytes.

User-to-User authentication

Kerberos supports user-to-user authentication. This can be useful when a service does not have access to its long-term service key, or when running peer-to-peer applications. It falls beyond the scope of this thesis to completely describe this mode of operation for Kerberos. Interested readers are directed to [65] for further details. It is however important to note that the TLS-KDH design has been accommodated to support user-to-user authentication. For user-to-user authentication, a client must send an extra Ticket Granting Ticket (TGT) in its Ticket Granting Server exchange (i.e. `TGS_REQ` message). This extra TGT must be received from the the other user (i.e. the server-side). TLS-KDH can exchange such a TGT via its (optional) server `Certificate` message. This message will only be used in KDH-only mode when user-to-user authentication is desired.

3.1.2 KDH-enhanced

KDH-enhanced is the second of the two modes of operation defined by TLS-KDH. It defines a mechanism to enhance standard certificate-based server authentication with Kerberos based client authentication. During the handshake the client and server negotiate one of the regular[5] ECDHE certificate-based TLS cipher suites. Contrary to KDH-only, the server now sends a regular public-key certificate to the client via a server `Certificate` message. Also, a corresponding key exchange will be initiated via the regular `Sever Key Exchange`. In KDH-enhanced mode a `Certificate Request` must be sent however. In order for the client to respond with a Kerberos Ticket, this message must list at least the `kerberos_sign` ClientCertificateType in the **certificate_types** field. Additionally, it must specify one or more `kerberos`-based signature & hash algorithm pairs in the **supported_signature_algorithms** field. Depending on the client's capabilities and priorities it may decide to respond with a Kerberos Ticket in its `Client Certificate` message. If it does so, it must also embed an Authenticator in its `Certificate Verify` message. Otherwise the server would not be able to verify the client's authenticity. Instead of verifying the client's digital signature (as would be the case for PKI authentication), the server now verifies the Authenticator in the `Certificate Verify` message. If this succeeds, then the client is successfully authenticated. The client however, may also decide to divert from the KDH-enhanced mode by not sending a Ticket. The rest of the handshake follows normal TLS behaviour (see Section 2.3.2).

3.1.3 Ticket Request Flags extension

TLS-KDH defines a new TLS `Hello` message extension (see Section 2.3.2) that enables the client and the server to negotiate Kerberos Ticket request flags during a handshake. Ticket Request Flags (TRF) are flags with which the client can specify what Ticket facilities it can offer to the server. The client sends a list with flags to the server in the `Client Hello`. The server responds in the `Server Hello` with a list of flags that it wants the client to fulfil. This must be a subset of the flags that the client offered. Since a Ticket is used for identification, it is important to note here that the client is in control of the facilities that it offers. The server can then choose from this offer. In this design the server is not able to demand more than the client is willing to fulfil. This has been a privacy consideration. Kerberos already defines a set of Ticket Flags [65]. Additionally, TLS-KDH introduces the following new flags:

[5] With regular we mean the already available cipher suites in TLS. That means no KDH-only cipher suites that are introduced by the TLS-KDH specification.

LocalRealmService Indicates that the client should look for a service Ticket in its own local realm, and should not perform realm crossover.

VisibleClientRealm Requests that the client's realm name is revealed in the service Ticket.

UniqueClientIdentity Requests that the client presents a unique identity.

When a client and a server agree on using TLS-KDH, they must include the Ticket Request Flags extension.

3.1.4 Authenticator as signature

In public-key systems, private keys can be used to generate digital signatures while corresponding public keys can be used to verify them (see Section 2.1.2). Traditional certificate-based authentication mechanisms in TLS use this property to sign and verify messages between the client and the server during a handshake. This is how authentication can be established. Kerberos however, is based on symmetric cryptography which does not know digital signatures. In such systems authenticity of a peer can be established by proving knowledge of a shared secret, for example by being able to decrypt a message and use its contents accordingly. In Kerberos this shared secret is the session key that is issued by the KDC (see Section 2.2.2). Authentication of a peer is then accomplished by proving the ability to decrypt an Authenticator (hence the name). A Kerberos Authenticator can thus be seen as the symmetric equivalent of a digital signature, and will be used as such within the context of TLS-KDH. To be able to properly distinguish between different signature algorithms, a new signature algorithm type called `kerberos` is proposed to be registered with IANA's TLS parameters registry [43]. An Authenticator is then wrapped into a TLS DigitallySigned struct (Listing 3.1) as follows:

```
struct {
  SignatureAndHashAlgorithm algorithm;
  opaque signature<0..2^16-1>;
} DigitallySigned;
```

Listing 3.1: TLS DigitallySigned struct.

algorithm Contains a hash & signature algorithms identifier pair. In this case that must be `kerberos` for the signature algorithm. The hash algorithm identifier must match the hash algorithm with which the checksum in the Authenticator is generated.

signature Contains the DER representation of the Authenticator.

3.1.5 Ticket as (X.509) certificate

Traditional authentication in TLS is based on public-key certificates. First only of the X.509 type, later also OpenPGP. This certificate type can be negotiated during the handshake via an extension called `cert_type` [55]. This certificate type is then applied to both the client and the server. Client certificates are optional in regular TLS but the KDH authentication mechanism makes them obligatory. That means that a client must always supply a Ticket in both KDH-only as well as KDH-enhanced mode. A new Kerberos Ticket certificate type would be desirable in order to transmit Tickets. This would however conflict with the symmetric nature of the certificate type negotiation during the handshake for KDH-enhanced mode. KDH-enhanced uses traditional certificate authentication for server to client authentication, enhanced with a Kerberos Ticket for client to server authentication. That would imply two different certificate types, which is not possible in TLS. TLS-KDH solves this problem by embedding a Kerberos Ticket into an X.509

certificate. That way, a client Ticket still has an X.509 certificate type, and so no type discrepancy will occur. Since X.509 also requires that certificates are digitally signed, an Authenticator is also embedded in the certificate and serves as digital signature surrogate. An X.509 certificate containing a Kerberos Ticket is constructed as follows:

First a TBSCertificate [16] is built:

- The certificate is self-signed. Its issuer and subject are set to the distinguishedName CN=Kerberos.
- The certificate must not claim validity before its contained Ticket is valid, and must not claim validity after the Ticket expires. The certificate may be short lived, lasting for example from 2 minutes before the client's current time to 3 minutes after the current time.
- The signature field is set to an OID that signifies an Authenticator. The Authenticator's checksum type must be based on a TLS-accepted hash algorithm. The Authenticator must not contain a subkey field.
- The subjectPublicKeyInfo is set to an algorithm OID that signifies a Kerberos Ticket.
- The subjectPublicKey is filled with a Kerberos Ticket that the client wants to use to access a service.

The Certificate built from the TBSCertificate adds the following fields:

- The signatureAlgorithm contains the same value as the signature field in the TBSCertificate.
- The signatureValue field is filled with an Authenticator whose ckecksum is computed over the TBSCertficate.

A proof of concept implementation for generating such certificates has been developed and can be found on `https://github.com/arpa2/kerberos2pkix/`.

3.2 Design analysis & proposed modifications

The preliminary TLS-KDH design specification has been written as an Internet-Draft [70] as part of the ARPA2 project. We analyzed the, at that time, most recent version of this specification (i.e. version draft-vanrein-tls-kdh-01). It defines a novel way of integrating Kerberos into TLS. The specification can however be improved at some points to get a more elegant and flexible design. In this section we make a case for some modifications to the original design. Our recommendations have resulted in an updated Internet-Draft [71].

3.2.1 Asymmetric certificate type negotiation

The analyzed TLS-KDH specification (i.e. version draft-vanrein-tls-kdh-01) defines a way to embed a Kerberos Ticket and Authenticator in an X.509 certificate. The reason for this is that TLS enforces the same certificate type for a client and a server during authentication. Introducing a new certificate type for a Kerberos Ticket would break the KDH-enhanced mechanism. Choosing a Kerberos Ticket type for the client would imply the same type for the server. The server would therefore be unable to perform normal X.509-based authentication. Embedding a Kerberos Ticket in an X.509 certificate is one way of solving this problem. We think however that there is a more elegant solution.

Looking at the solution that is proposed in the analyzed specification, we can identify four drawbacks of embedding a Kerberos Ticket inside an X.509 certificate structure:

- It yields a large amount of superfluous data. X.509 prescribes fields that are mandatory but are in fact not used by the TLS-KDH mechanism. It is therefore an inefficient way of transporting a Ticket.

- It requires modifications to the X.509 parser in order to recognize this special certificate variant. Since this is already a complex system, it is not recommended to modify it if that is not necessary.

- It restricts the KDH-enhanced mechanism to X.509 authentication for the server because the negotiated certificate type applies to both peers.

- It is not really semantically correct to fold non public-key material inside a public-key structure. The fact that it fits there does not necessarily mean it belongs there.

By looking at the goal that the KDH-enhanced mechanism is trying to achieve, we can clearly identify the need for asymmetric certificate types. That means the possibility to negotiate a different certificate type for the client than for the server. In particular, we need a mechanism to:

1. Transmit Kerberos Tickets and Authenticators with as little baggage as possible,

2. Negotiate asymmetric certificate types for a TLS session.

Both of these issues are addressed in a TLS extension called "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)" [90]. The extension defines two mechanisms. The first one describes a way of communicating only raw public keys. Raw public keys consist of only a public key without the surrounding certificate data. In terms of X.509 this boils down to transmitting only the SubjectPublicKeyInfo [16]. The second mechanism describes a way of independently negotiating a certificate type for the server and for the client. This preserves the possibility to negotiate the same certificate types, but also introduces the possibility to negotiate different certificate types. This mechanism is actually more flexible than the `cert_type` extension defined in [55], and should therefore be used instead.

The Raw Public Keys extension defines a new certificate type called "Raw Public Key". It is registered in IANA's Transport Layer Security (TLS) Extensions register [42] next to the types "X.509" and "OpenPGP". Since this Raw Public Key certificate type is used to, as the name implies, indicate raw public keys, we are not allowed to use it for transmission of a Kerberos Ticket. There would be no way how one could distinguish a Kerberos Ticket from a raw public key. On the other hand, we can use the IANA register to introduce a new certificate type for a Kerberos Ticket. This new certificate type can then be negotiated for a client and server independently with the asymmetric certificate type negotiation mechanism defined in the Raw Public Keys extension. We therefore propose to use:

1. A new Kerberos certificate type called "KerberosTicket".

2. The asymmetric certificate type negotiation mechanism as defined in the Raw Public Keys extension [90].

The new Kerberos certificate type that represents a Kerberos Ticket is embedded in a `Certificate` message according to the following definition. This is an extension of the original `Certificate` message definition (see Section 2.3.2).

```
struct {
  select(certificate_type){
    // Kerberos certificate type defined here.
    case KerberosTicket: opaque Ticket<0..2^24-1>;
    // Other certificate types based on the
    // "TLS Certificate Types" subregistry
  };
} Certificate;
```

Listing 3.2: Embedding of a Kerberos Ticket into a Certificate message.

Ticket Contains the DER-encoded form of a Kerberos Ticket.

The Raw Public Keys extension introduces two `Hello` message extensions called "Client Certificate Type" and "Server Certificate Type". The working of these extensions is further explained in Section 6.1.3. Using this extension greatly extends the possibilities of TLS' certificate authentication without breaking existing mechanisms. The advantages of using a dedicated certificate type for Kerberos Tickets is threefold:

- It is more efficient. No extra X.509 formatting overhead is present, including, no extra Authenticator needs to be encapsulated in the certificate.
- It enables server authentication via other mechanisms than only X.509 certificates.
- It introduces a semantically sound data type and identifier for Kerberos certificates that can be used to properly distinguish between certificate types.

3.2.2 Ticket Request Flags encoding

The Ticket Request Flags (TRF) extension makes it possible to negotiate extra constraints on the Ticket that a client supplies. The analyzed specification however, dictates that this extension must always be sent. It serves, next to potentially choosing a KDH-only cipher suite, as a signal that both parties are able to do TLS-KDH. With the new asymmetric certificate type negotiation mechanism this is no longer necessary. The ability and willingness to do TLS-KDH can now accurately be expressed via this new mechanism. We therefore propose to make the TRF extension optional, and to only use it when Ticket Request Flags need to be negotiated. Secondly, we think that the encoding of the TRF can be improved. The analyzed specification dictates the following:

> The wire format representing TicketRequestFlags is a sequence of 32-bit integers. The lower 5 bits of a flag number provide the number of the bit within a 32-bit integer, and the higher bits indicate the integer's index in the sequence of 32-bit integers. Senders MUST set unknown flags to value 0 and recipients MUST NOT act on flags they cannot interpret.

Looking at this specification, two things stand out. First, it deviates from Kerberos' way of specifying flags. Kerberos uses an ASN.1 bit string that represents a boolean vector, to encode flag values. Currently, a minimum length of 32 bits is required according to the specification [65]. Because there are currently no more than 32 flags defined, KerberosFlags are being encoded as 32-bit bit strings. Secondly, this encoding looks rather complicated for the number of flags it currently defines. The TRF extension enables transmission of Kerberos' TicketFlags plus three extra flags defined by the TLS-KDH specification. Since Kerberos reserves 32 bits for its own flags[6],

[6] Not all reserved space is currently used, but this might be the case in the future.

additional space needs to be reserved for the extra flags that TLS-KDH defines. In order to leave room for extra flag definitions in the future, and to conform to Kerberos' way of encoding flags, we propose to use an extra 32-bit bitmap. This yields a simple and easy to implement encoding. The proposed TRF encoding would then become 32 bits Kerberos TicketFlags concatenated with 32 bits TLS-KDH Ticket Request Flags. It is depicted graphically in Figure 3.3.

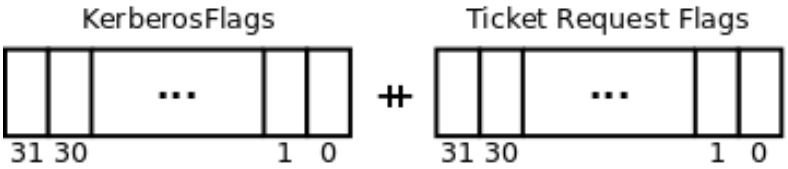


Figure 3.3: Proposed Ticket Request Flags encoding.

3.2.3 Premaster secret computation

As is described in Section 3.1.1, the premaster secret computation for KDH-only is composed of:

- The negotiated Diffie-Hellman key,
- The session key that is embedded in the Ticket (KDC-controlled),
- The subkey that is embedded in the Authenticator (client-controlled).

From a security point of view we propose to remove the session key from this premaster secret computation. First of all, leaving the session key in the premaster secret computation adds no extra security. The premaster secret should depend on the decrypted Authenticator in order for the client to be able to verify the server's authenticity. Since this Authenticator can only be decrypted with the session key that is embedded in the Ticket, the server already proves its knowledge of this (session) key by decrypting the Authenticator.

Secondly, a Kerberos session key is a shared secret between a Client and a Ticket Granting Server, or a Client and an Application Server. It is generally a good practice to keep it that way. Using this key in the premaster secret computation would require the server to decrypt the Ticket and hand over the embedded session key to the TLS layer. The TLS layer is then able to potentially decrypt Kerberos data (i.e. data exchanged between a Client and an Application Server). Handing over both the session key and the Authenticator / subkey could be dangerous. A vulnerable TLS library could compromise Kerberos sessions by exposing these keys. In general it is good security practice not to hand over sensitive information, like encryption keys, if that is not necessary. We therefore suggest to compute the premaster secret (pms) as follows:

$$pms = |K_{DH}| + K_{DH} + |A| + A$$

Here K_{DH} is the established DH-key, A is the decrypted Authenticator, $|\cdot|$ is the length function[7] that outputs a 16-bit length and $+$ is the concatenation operator. We use the entire Authenticator, which must contain a fresh **subkey** (see Appendix A), to add more entropy. The amount of entropy provided by the Authenticator must at least equal the number bits in the key of the symmetric cipher in the negotiated cipher suite. Because most fields of an Authenticator can be guessed by an attacker, a client must supply this entropy by means of a sufficiently large and random **subkey**. We therefore require that that the amount of entropy provided by the **subkey** must at least equal the number bits in the key of the symmetric cipher in the negotiated cipher suite.

[7] The length of an object equals the number of bytes the object is long.

3.2.4 Cipher suite updates

Looking at the KDH-only cipher suites, we propose to remove the ARIA cipher suites because the ARIA cipher [51] is not implemented in any of the well known TLS libraries [88]. Furthermore, we propose to add new variants of the remaining KDH-only cihper suites to also include the SHA512 hash algorithm. The SHA512 hash algorithm is becoming more mainstream and should therefore be included for the cipher suites to be more up-to-date. The new list with proposed KDH-only cipher suites is given in Table 3.2. The original list can be found in Table 3.1.

Cipher suite	verify_data_length	PRF
TLS_ECDHE_KDH_WITH_AES_128_GCM_SHA256	32	SHA256
TLS_ECDHE_KDH_WITH_AES_256_GCM_SHA384	48	SHA384
TLS_ECDHE_KDH_WITH_AES_256_GCM_SHA512	64	SHA512
TLS_ECDHE_KDH_WITH_AES_128_CCM	32	SHA256
TLS_ECDHE_KDH_WITH_AES_256_CCM	48	SHA384
TLS_ECDHE_KDH_WITH_AES_128_CCM_8	32	SHA256
TLS_ECDHE_KDH_WITH_AES_256_CCM_8	48	SHA384
TLS_ECDHE_KDH_WITH_CAMELLIA_128_GCM_SHA256	32	SHA256
TLS_ECDHE_KDH_WITH_CAMELLIA_256_GCM_SHA384	48	SHA384
TLS_ECDHE_KDH_WITH_CAMELLIA_256_GCM_SHA512	64	SHA512

Table 3.2: Proposed new KDH-only cipher suites with their properties.

3.2.5 Explicit PRF definitions

TLS defines a pseudorandom function (PRF) that is used to expand random material (see Appendix B). As of TLS 1.2, a cipher suite can define its own PRF and corresponding **verify_data** length (as was described in Section 3.1.1). The analyzed TLS-KDH specification does specify custom **verify_data** lengths for the KDH-only cipher suites, but does not specify a PRF to be used. A different PRF however, would yield different **verify_data** and therefore different `Finished` messages. Implementations that would use different PRFs for the same cipher suite would become incompatible. We therefore propose to explicitly define a PRF for every new cipher suite. The proposed PRFs are listed in Table 3.2. The PRFs are chosen such that their output lengths correspond to the verify_data_length values.

3.2.6 Authenticator checksum value

The analyzed specification dictates that an Authenticator should contain a hash in its checksum field. It is however not directly clear from this specification over what data this hash has to be computed. According to the specification [70]:

> Kerberos has a symmetric analogon to a signature, namely an Authenticator [Section 5.5.1 of [RFC4120]] that MUST have a secure hash embedded in the cksum field. The checksum type used in the context of TLS MUST be taken to match one of the entries in IANA's TLS HashAlgorithm Registry.

To create a proper cryptographic binding between Kerberos and TLS, this hash should be computed over the **handshake_messages** as is the case for a regular `Certificate Verify` message (see Section 2.3.2). We therefore propose to make this design decision explicit in the TLS-KDH specification. The rest of the Authenticator still conforms to the specification described in Section 3.1.4.

Chapter 4

Security analysis of TLS-KDH

TLS-KDH provides a cryptographic binding between Kerberos and (ECDH) TLS. By combining these two mechanisms, their strong points can be leveraged whereas their shortcomings can be mitigated. In this chapter we will give a security analysis of the TLS-KDH mechanism.

4.1 Kerberos security

Kerberos is a mature authentication protocol that has become an industry standard. It has therefore been subjected to many security analyses [19, 12, 6, 91, 22, 83]. Kerberos is based on the symmetric Needham-Schroeder protocol. This protocol establishes a session key between two communicating parties that is used to establish (mutual) authentication. The original protocol however, is known to be vulnerable to replay attacks. Kerberos solves this problem by incorporating a timestamp (i.e., an extra nonce) in the message exchanges [62, 20]. Although the protocol itself is secure, several key security considerations can be identified and solved. A formal analysis or extensive security review of Kerberos falls beyond the scope of this thesis. We will however, give an overview of the most important security remarks.

Kerberos' security depends on the security of the KDC. The KDC contains all the long-term secret keys of all principals in its security realm. They are used to encrypt tickets and session keys. If the KDC gets compromised, then all trust that a client and server derive from this KDC is gone. An attacker can use the compromised keys to decrypt all traffic (even previously recorded traffic) and to impersonate users and services. The KDC is a single point of failure and it is therefore of utmost importance to properly protect it. Additionally, principals should keep their secret keys secret as well. A principal with compromised keys can be impersonated by an attacker, without this being easily detected.

Traditional Kerberos employs user passwords to derive long-term secret keys. By compromising such passwords an attacker would be able to impersonate the respective user. This cannot be easily detected, other than by the user itself. Kerberos does not protect against password-guessing or brute-force attacks. These attacks can be performed offline by attempting to decrypt messages that were encrypted with this password-derived key. It is therefore important that users choose strong passwords and protect them well. Additionally, these passwords should be entered via a trusted path. If the initial registration can be monitored, or if a client program is compromised, then an attacker could learn enough to impersonate the user. Extensions such as FAST [36] and PKINIT [94] have been developed to overcome the issues related to password logins.

Application services that use Kerberos for authentication must implement a mechanism to detect replay attacks. A common solution for this is a replay cache that buffers Authenticators for the period of time that they are considered valid. Because Authenticators contain a timestamp

representing the moment they were generated, a replayed Authenticator can be detected. In order for this mechanism to work, some form of clock synchronization between principals is necessary. It is important that this synchronization mechanism is also protected from tampering.

Kerberos does not provide authorization. It does however provide placeholders for authorization data in Tickets and can therefore be used as a base for building separate (distributed) authorization services [63]. It is important to note that applications or services that use Kerberos should not assume that an authenticated user is automatically authorized for a service. These services should perform additional (authorization) checks based on the asserted identity. The only statement that can be derived from a client's Ticket, is that its (encrypted) contents are authentic. A Ticket might include (external) authorization data, but this cannot be generally assumed.

Kerberos is vulnerable to Denial-of-Service (DOS) attacks. If an attacker is able to block communications between a principal and the KDC, or overload a KDC, then authentication cannot take place. Additionally, any service that depends on Kerberos is therefore also vulnerable to DOS attacks. When designing systems where availability is important, one should deploy redundancy solutions for the KDC.

Kerberos does not provide Perfect Forward Secrecy. When a user password or long-term secret gets compromised, all subsequent Tickets, Authenticators and session data can be unravelled, as well as all data that has been previously recorded by an attacker. This can easily be seen by the fact that every session key is wrapped inside a structure that is encrypted with another session key, up to the point of the key that is derived from the user's password (see Section 2.2.2).

4.2 TLS security

TLS is a mature and widely used cryptographic protocol. Like Kerberos, it has been subjected to many security analyses [30, 49, 60, 69, 22]. Additionally, a summary of all currently known attacks on TLS has been officially published [76]. It falls beyond the scope of this thesis to discuss all security issues of TLS. That would be worth a thesis on its own. Many attacks and security problems of TLS have been addressed in newer versions of the protocol, or in specific extensions. Also, several security problems arise from faulty implementations of the protocol. Therefore, a thorough list of implementation pitfalls is given in [21] and should be taken into account when implementing a TLS library. It is therefore recommended to use an existing TLS library that has been thoroughly tested and evaluated. Extra care should be taken when implementing your own library, or extending an existing one like we did for our proof of concept (see Chapter 6). It should be noted that all known TLS security issues may also apply to TLS-KDH unless explicitly refuted in this chapter.

Man-in-the-Middle (MitM) attacks are a common threat to communication protocols and are complex to handle. They can be mounted on a variety of attack surfaces, ranging from the protocol itself to the applications and machines using it. Man-in-the-Middle attacks can be prevented by using authenticated encryption. That means that the encryption keys are authenticated. Standard TLS realizes this through a public-key infrastructure like X.509. Public keys are authenticated via the accompanying certificate, while the private key is kept private to the owner of the certificate. This system works fine until there are certificate authorities that get compromised. False certificates can be issued to mislead the client. Additionally, if endpoints themselves get compromised, then again no trust can be derived from the protocol. If an attacker gets hold of a server's private key, it can impersonate that server. If, for example, a client's web browser gets compromised via a script exploit, then the client can be tricked into trusting a rogue server. Extensive research into MitM attacks has been done for TLS [47, 7, 18, 40]. Depending on where and how TLS is used, risks for MitM attacks may still apply. We are not going to perform an in-depth analysis of MitM attacks for general TLS in this thesis. We do, however, address it in the context of TLS-KDH in Section 4.3.

4.3 TLS-KDH security

To argue the security of TLS-KDH we have analyzed those places in the TLS protocol flow where the KDH mechanism diverts from the regular flow or processing. If not explicitly stated otherwise in this section, existing Kerberos and TLS security issues (see Sections 4.1 and 4.2 respectively) may still apply to TLS-KDH. The identified places of interest for the security analysis of TLS-KDH are in particular the certificate verify procedures and the construction of the `Finished` messages.

4.3.1 General analysis

By integrating Kerberos into TLS, the strengths of both mechanisms can be leveraged. First of all, TLS now has the capability of performing authentication through Kerberos. That means that it now supports a widely used strong authentication mechanism that does not rely on a public-key infrastructure (PKI). Although really useful and widely used, PKI's have shown to have some weaknesses [67]. If certificate authorities get compromised, then the trust of the PKI and its certificates can be undermined. It is therefore desired to have strong alternatives with respect to authentication mechanisms. Kerberos offers such facilities. The disadvantage of Kerberos with respect to a PKI is that there is no Ticket revocation mechanism. Instead, it relies on short-lived, often-released credentials. A Ticket that has been issued remains valid until it is expired. Certificates on the other hand can be revoked on-the-fly, but require a bulky CRL or an online mechanism such as OCSP.

TLS implements a large set of cryptographic algorithms, many more than Kerberos implements. By combining Kerberos and TLS, TLS' algorithms are now also available to Kerberos. That means that Kerberos can now use state-of-the-art encryption to secure its sessions, including elliptic-curve cryptography (ECC). Furthermore, TLS supports PFS through ephemeral Diffie-Hellman key exchange algorithms. By enforcing both TLS-KDH modes to use only the ECDHE key exchange algorithm, all TLS-KDH Kerberos sessions now satisfy the property of PFS.

As explained in Section 4.2, Man in the Middle (MitM) attacks pose a serious threat against communication protocols. Both TLS and Kerberos are secure against such attacks under certain assumptions. The most important one is that the mechanism that provides trust is secure. For X.509, this means that the public-key infrastructure (PKI) is trustworthy (which has been proven to be a dangerous assumption [67]). For Kerberos, this means that the KDC is secure. Additionally, it is mandatory that both endpoints (i.e. client and server) are secure as well. It would be silly to derive trust from a communication protocol if one of the endpoints is compromised. For the analysis of TLS-KDH, we therefore assume that both endpoints as well as the KDC are secure. We can then analyze the behaviour of the TLS-KDH mechanism itself.

In KDH-enhanced mode, only client authentication depends on Kerberos. That means that MitM attacks are still possible if the mechanism with which the server authenticates itself gets compromised. This however, introduces no new issues related to KDH-enhanced mode because it involves a different (already existing) authentication mechanism (e.g. X.509 PKI). For KDH-only mode, the authentication of both the client and the server depend on Kerberos. Since we assume a secure KDC, the KDH-only mechanism is secure if the Kerberos-derived authentication can be bound to a TLS session. Because in KDH-only mode, TLS and Kerberos are cryptographically bound, this is indeed the case: The TLS session's master secret depends on a fresh key that is embedded inside a Kerberos Authenticator (see Section 3.1). This key is not used as a key, but serves as entropy for the Authenticator and it is concealed from anyone but the intended recipient. The decrypted Authenticator is, together with the session's Diffie-Hellman key, used as the inputs for the premaster secret (see Section 3.1.1). The master secret is, among others, constructed from the premaster secret and the client and server exchanged random values (see Section 2.3.4). This ensures that a TLS session is bound to a Kerberors-authenticated principal. As long as the KDC is secure, then KDH-only sessions are protected from MitM attacks.

Normal Kerberos application servers must implement a replay cache to detect replayed Authenticators. A replayed Ticket/Authenticator pair can be useful to an attacker within the validity time of an Authenticator, to impersonate a client. TLS-KDH overcomes the need for a replay cache because each TLS session is seeded with random material. This random material comes from the client and server generated random values in the `Hello` messages. Additionally, in case of KDH-only, random ECDHE key exchange material is added. Since an Authenticator contains a checksum over the session's preceding handshake messages, a replayed Authenticator will not pass this checksum validation. In order to replay an Authenticator, a new session to the server must be initiated, and therefore different handshake parameters are used (i.e. fresh random values). It will therefore not be necessary for TLS-KDH servers to cache used Authenticators. Not needing to implement an Authenticator cache reduces complexity and optimizes performance, especially in the case of distributed systems that would need to share the replay cache.

4.3.2 Out-of-Band Replay Attack

Although TLS-KDH protects itself from replay attacks, there is still a broader setup in which a replay attack can be mounted, if not properly mitigated. We call it the "Out-of-Band Replay Attack", and it works as follows. A graphical representation is given in Figure 4.1.

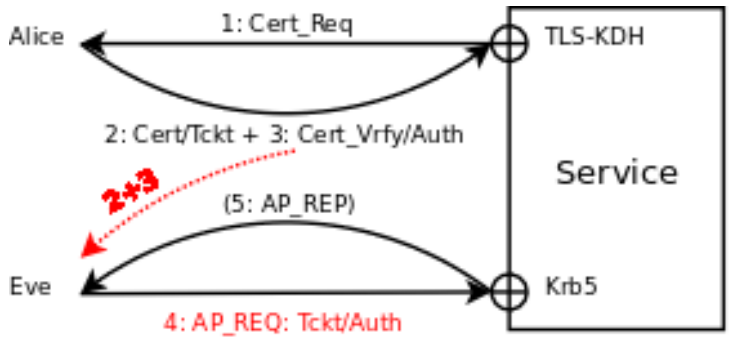


Figure 4.1: Out-of-Band Replay Attack. Eve steals a Ticket/Authenticator pair from Alice and replays it via a different channel to the same service. The TLS-KDH channel is assumed not to support a replay cache because it does not look beyond TLS-KDH. The Krb5 channel may assume, and rely on, a replay cache. Message number 5 is optional.

Assume we have a server that is identified by the principal name "print/services.arpa2.net". This application service is capable of doing TLS-KDH, but also accepts a regular kerberized protocol based on `AP_REQ` messages. In other words, this service has two different channels on which it can authenticate. Now assume that the TLS-KDH implementation does not provide support for a replay cache. Lets further assume that there is an attacker, Eve, that is able to monitor and capture all network traffic. A legitimate user, say Alice, now wants to print something and therefore starts a TLS-KDH handshake with print/services.arpa2.net. Eve is listening on the network and sees a Ticket and corresponding Authenticator passing by. Eve records these credentials and replays them, within the validity window of the Authenticator, to the same service. However, he replays them in an `AP_REQ`-based channel instead of the TLS-KDH channel. Since the server has no replay cache for TLS-KDH, the replayed Authenticator will not be detected. Additionally, since a checksum is optional for an Authenticator, a regular Kerberos service may not notice its presence. The checksum that is added by TLS-KDH is only relevant for that mechanism. Therefore, Eve has successfully mounted a replay attack and has impersonated Alice for this service.

It must be noted that this attack can only succeed if and only if the following conditions are met:

- The attack is mounted on a service with the same principal name as the principal for which Alice got the Ticket and Authenticator. If not, then the key under which the Ticket is encrypted by the KDC is different.
- The attack must be mounted within the validity time of the Authenticator as observed by the `AP_REQ`-based service. Otherwise the Authenticator is detected as being expired.
- The service must be accessible via two channels that do not share a replay cache. This is of key importance for this attack to succeed! If the service implements one replay cache for both TLS-KDH accesses and regular Kerberos accesses, than a replayed Authenticator would be detected. If separate replay caches are used (i.e. one for each mechanism), or if one of the mechanisms does not have a replay cache, then this attack can be successful.
- No other replay detection mechanisms are deployed by the service.

Countermeasures

The Out-of-Band replay attack demonstrates that the necessity for a replay cache depends on the deployment of a service. If one only uses TLS-KDH to provide access to a service, then a replay cache may be omitted. If other means of access are also present, extra care should be taken. As was already stressed by the designers of Kerberos [65], the need for a replay cache (or other detection mechanism) has to be carefully considered by the designers of the application service. The Out-of-Band replay attack can be easily prevented by implementing a replay cache that is shared for both access channels. The fact that TLS-KDH does not require a replay cache does not mean that implementers are forbidden to add one. Another solution would be to limit the number of access mechanisms to the service to just one. This reduces system complexity and makes the necessity for a replay cache more evident and easier to implement.

Applicability

As an example, we sketch a typical scenario in which the Out-of-Band attack might be applied. Assume a service provider that offers IMAP services with server authentication via TLS (e.g. X.509). Originally, client authentication is provided through SASL [57] and Kerberos via the GSS-API [54]. Assume that, at some point, the provider wants to migrate to the TLS-KDH mechanism (in combination with the SASL-EXTERNAL mechanism). In order to provide a migration period for its users, both the old and the new authentication mechanisms must be active. This results in two channels to the same service that employ kerberos authentication. This is a typical scenario where the Out-of-Band replay attack can be mounted if no proper replay detection mechanism is put in place.

Chapter 5

Performance analysis of TLS-KDH

In this chapter we analyze the performance of the TLS-KDH mechanism compared to TLS with X.509. TLS with the X.509 public-key infrastructure for authentication is currently the most widely used configuration for TLS and resembles the most similarities with the KDH mechanism. The performance will be analyzed based on the designs of both mechanisms and will therefore be theoretical in nature. A practical analysis, based on real runs, falls beyond the scope of this thesis, among others because of the complexity and the absence of the setup required to do a proper analysis. The analysis performed in this chapter compares both mechanisms based on the number of computations needed to set up and execute comparable TLS sessions.

When analysing the performance of both mechanisms, we only need to look at those points where the two mechanisms differ. Looking at the TLS protocol flow we can identify three phases:

1. Obtaining credentials.
2. Authentication and parameter negotiation (i.e. the TLS handshake).
3. Exchange of data over the established secure channel.

Since TLS-KDH neither introduces any new cipher, hash or MAC algorithms nor requires specific compression algorithms [71], Phase 3 will be the same for both TLS with X.509 and TLS-KDH. After a successful handshake, the same cryptographic algorithms can be negotiated for both mechanisms and therefore the actual number and type of computations performed in Phase 3 is equal[1].

The actual differences between the two mechanisms occur in Phases 1 and 2. During Phase 1, peers need to obtain credentials to use for authentication. For X.509, this means obtaining client and/or server certificates. For Kerberos, this means obtaining Tickets. TLS allows for anonymous authentication where only cryptographic keys are being exchanged for data encryption, and no identities are being verified [21]. This anonymous use-case therefore does not require any credentials. TLS-KDH however, always requires authentication and therewith the presence of credentials. In order to do a fair comparison, we will therefore only compare TLS use-cases that perform authentication during the handshake. The details of each comparison are given in their respective sections. Since obtaining credentials is not really part of the TLS protocol itself, we will mainly focus our analysis on Phase 2. However, since credentials are necessary in order to be able to authenticate, a generic comparison between Tickets and certificates will be made in Section 5.2.

[1] When performed on the same input data.

In Phase 2, authentication takes place and the session's security parameters are negotiated. TLS-KDH places some restrictions on the cryptographic algorithms that are allowed in order to enforce algorithms that are considered strong at the time of writing. In order to do a fair comparison between the two mechanisms, we will therefore limit our analysis to only those algorithms. TLS-KDH introduces two modes of operation: KDH-enhanced and KDH-only (see Section 3.1 for their description). We will analyze the computational differences between these two modes and the X.509 mechanism within the scope of TLS.

5.1 Comparing symmetric and asymmetric encryption

In order to be able to discuss the performance of cryptographic protocols, one must first understand the fundamental difference between symmetric and asymmetric encryption. Symmetric encryption algorithms are based on the principles of "confusion" and "diffusion" [75]. Without delving into the mathematical details, this comes down to mixing the input data in such a way that the output looks completely random. Additionally, it should be (practically) impossible to find a relation between the plaintext, the ciphertext and the key. The basic operations necessary for achieving confusion and diffusion are typically xor, addition, bit rotations and (small) table lookups. These are all operations that a CPU can do extremely fast. Additionally, they are easy to invert. Recall from Section 2.1.2 that symmetric encryption uses the same key for encryption as for decryption. In order to reverse an encryption (i.e. decrypt a ciphertext), the steps in the encryption algorithm must be performed in reverse order and the operations must be inverted. Encryption is therefore equally complex as decryption.

Asymmetric encryption algorithms, on the other hand, are constructed differently. They realize a more difficult feature of being able to publish a public key for encryption, while not revealing anything about the private key that is used for decryption. In order for this to be possible, asymmetric encryption algorithms are based on mathematical functions that are easy to compute but hard to reverse. Most of these functions are currently based on the integer factorization problem [68], the discrete logarithm problem [68] (DLP) and the elliptic-curve discrete logarithm problem [68] (ECDLP). The operations involved to compute these functions consist primarily of modular exponentiation. This is an expensive operation that costs significantly more CPU cycles than the basic operations needed for symmetric encryption. Additionally, in order to achieve a comparable level of security, asymmetric encryption algorithms require much greater key lengths [52]. That means that the operands in the calculations are much larger than with symmetric algorithms. Arithmetic with numbers much larger than the CPU's natural word length is slow.

Because symmetric algorithms lend themselves well for bulk encryption, hardware support for such algorithms has been developed to speed things up even more. This makes the performance difference between symmetric and asymmetric systems even greater in some cases. Although several studies have been performed on the performance of encryption algorithms [61, 23, 8, 13, 53], they rarely compare symmetric with asymmetric algorithms. Despite lacking a formal proof or thorough testing, a general estimate is that asymmetric algorithms are guessed to be a factor 100 to 5000 [82, 81, 27] slower than symmetric algorithms. The actual performance of an encryption algorithm is in the end based on the specific algorithm itself, its parameters (e.g. key size) and the size of the input. Also, the underlying hardware on which the algorithm is executed plays a major role. Without being able to fully quantify the computational difference between symmetric and asymmetric encryption algorithms, we can conclude that, in general, symmetric algorithms are more efficient than asymmetric ones. For the sake of comparison, we use the aforementioned difference factors in the remainder of this thesis.

5.2 Obtaining credentials

Obtaining credentials for authentication purposes is not really part of the TLS protocol. Credentials are however necessary for an authenticated handshake. Even if the handshake itself would be very efficient for a specific mechanism, if obtaining credentials would be very tedious the mechanism could still be unfavourable. We therefore compare how credentials are obtained for TLS-KDH and X.509.

5.2.1 X.509 credentials

X.509 certificates are used within the context of a public-key infrastructure. Typically, Certificate Authorities (CA) issue certificates to anyone who requests one. Certificates come in various "validation levels". That means that the amount of validation performed on the public information in these certificates varies. Standard certificates can be issued fully automatic, whereas Extended Validation certificates require a more elaborate procedure of, for example, verifying personal or corporate information. For every new certificate, a fresh public/private key pair should be generated. The public key is then embedded into the certificate together with other public information of the holding identity. Finally, the certificate is signed by the issuing CA. Certificates have a typical validity of one to three years, but they can be revoked if the corresponding private keys get compromised. Key pair generation for asymmetric cryptographic algorithms is not straightforward. Depending on the algorithm, different requirements apply. Below, we give an overview of the key generation procedures for RSA [68] and DSA [68]. RSA is based on the integer factorization problem, whereas DSA is based on the discrete logarithm problem. Both algorithms are supported in TLS.

RSA key generation

RSA is a widely used public-key algorithm that can be used for encryption and digital signatures. It is based on the integer factorization problem. Because RSA is relatively slow, it is not commonly used in practice to encrypt large amounts of data. Instead, it is often used to encrypt and distribute a symmetric key between to parties. This is referred to as hybrid encryption. A summary of the RSA key generation steps is given to provide some insight into the complexity of generating an RSA public/private key pair [79].

1. Choose two distinct (large) prime numbers p and q. Suitable numbers can be found by using a primality test.
2. Compute $n = pq$. Typically n has a length of 2048 bits or higher[2].
3. Compute $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1) = n - (p+q-1)$, where ϕ is Euler's totient function. This value is kept private.
4. Choose an integer e such that $1 < e < \phi(n)$, and e and $\phi(n)$ are coprime.
5. Determine d as $d \equiv e^{-1} \bmod \phi(n)$.

The public key will be $\langle n, e \rangle$. The private key will be d. It can be seen from the above that the generation of a key pair is quite involved and requires non-trivial math. It is in any case more complex than generating a random bit-string that can be used as a symmetric encryption key (see Section 5.2.2).

[2] Depending on the required level of security.

DSA key generation

The Digital Signature Algorithm (DSA) is a signature algorithm that is part of the NIST[3] Digital Signature Standard (DSS). It is based on the discrete logarithm problem and is a variant of the ElGamal Signature Scheme. A summary of the DSA key generation steps is given to provide some insight into the complexity of generating a DSA public/private key pair [79].

1. Choose an approved cryptographic hash function H (e.g. from the SHA2 family). The hash output may be truncated to the size of a key pair.
2. Decide on a key length L and N. FIPS 186-3 specifies L and N length pairs of (1,024, 160), (2,048, 224), (2,048, 256), and (3,072, 256). N must be less than or equal to the output length of the hash H.
3. Choose an N-bit prime q.
4. Choose an L-bit prime modulus p such that $p - 1$ is a multiple of q.
5. Choose g, a number whose multiplicative order modulo p is q.
6. Choose a secret key x by some random method, where $0 < x < q$.
7. Calculate the public key $y = g^x \bmod p$.

Again it can be seen that the generation of a key pair is quite involved and requires non-trivial math. It is in any case more complex than generating a random bit-string that can be used as a symmetric encryption key (see Section 5.2.2).

5.2.2 TLS-KDH credentials

Kerberos credentials consist of a Ticket. This can be a Ticket Granting Ticket that is used by a Client to authenticate to a Ticket Granting Server, or a service Ticket that is used to authenticate to an Application Server. In both cases the KDC issues these Tickets. Tickets contain, among others, information about the Client, the service and a session key. Part of the data inside a Ticket is symmetrically encrypted using a randomly generated long-term secret key (see Section 2.2.2). Generating symmetric keys (i.e. a random bit-string) and applying symmetric encryption is significantly more efficient than its asymmetric counterparts. We therefore conclude that TLS-KDH credentials can be more efficiently (with respect to the computational complexity) obtained than X.509 credentials (see Section 5.2.1). Tickets however, are short-lived and have a typical validity of 8 to 24 hours. Additionally, a Ticket needs to be obtained for every service separately. Tickets thus have to be generated more often than X.509 certificates.

5.3 TLS handshake

When X.509 is negotiated as authentication mechanism for TLS, then client authentication is optional. The server decides whether or not to send a `Certificate Request` to the client. However, even when the server decides to send such a request, the client is free to choose whether or not to reply with a `Certificate`. TLS-KDH however, always requires a client certificate. Both KDH-enhanced and KDH-only modes establish mutual authentication. In order to do a proper comparison, we therefore compare these two modes with X.509's mutual authentication mode (i.e. X.509 with `Certificate Request` and client `Certificate` messages). It should be

[3] National Institute of Standards and Technology. NIST is a measurement standards laboratory in the USA.

noted however, that a client is allowed to divert from the KDH-enhanced mode by not sending the requested Ticket. For example, when it cannot fulfil the Ticket Request Flags. In that case one does not speak of KDH-enhanced any more, but rather of "regular" TLS (with X.509 server authentication).

5.3.1 Comparing KDH-enhanced with X.509

When looking at a TLS handshake, different messages are passed back and forth depending on the authentication mechanism that has been negotiated. For KDH-enhanced and X.509, the message flow of the handshake is the same. Only the contents, and the corresponding generation and verification processes, of some of these messages will differ. The messages that contain differences between KDH-enhanced and X.509 are depicted graphically in Figure 5.1 and are marked in red.

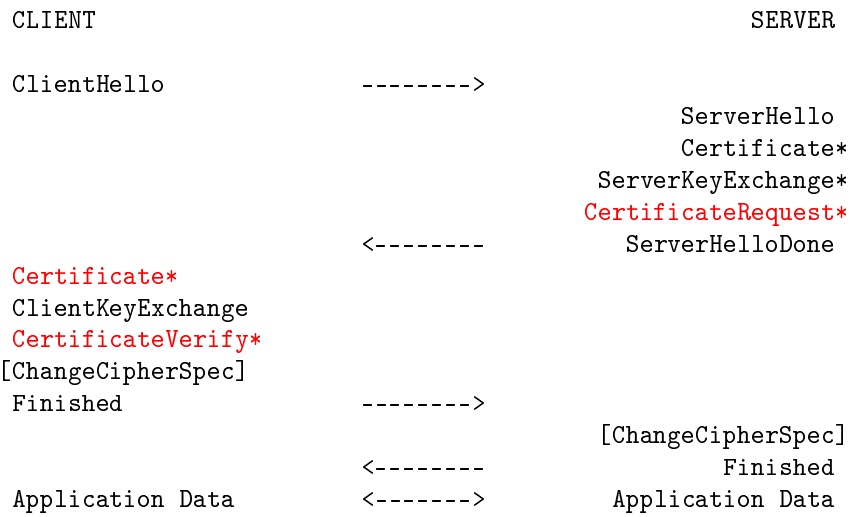


Figure 5.1: TLS handshake message flow. Messages in red mark the differences between KDH-enhanced and X.509.

First of all, it should be noted that the key exchange messages are the same for both mechanisms. They depend only on the negotiated key exchange algorithm, but not on the negotiated certificate type. Choosing either Kerberos or X.509 for client authentication does not change the contents or behaviour of these messages.

The `Certificate Request` message contains a list of allowed certificate types and allowed signature & hash algorithm pairs for the client (see Section 2.3.2). When operating in KDH-enhanced mode, the **certificate_types** list must contain the `kerberos_sign` value, and the **supported_signature_algorithms** list must contain at least one signature & hash algorithm pair that is based on a `kerberos` signature type. Adding those items to these lists is a constant time operation that does not add significant computational complexity. The difference resulting from this (extra) operation can therefore be neglected.

The client `Certificate` message contains either an X.509 certificate or a Kerberos Ticket. Both are credentials that have to be procured in Phase 1, and they do not computationally influence the protocol behaviour. A description of the difference between obtaining different credential types is given in Section 5.2.

The `Certificate Verify` message contains either a digitally signed hash (see Section 2.3.2) or an Authenticator containing that same hash (see Section 3.2.6). This is where a difference between the two mechanisms arises. First of all, the hash over the **handshake_messages** needs to be computed for both mechanism, on both sides. We can therefore neglect that computation in our

comparison. In the case of X.509, this hash is then digitally signed with a RSA, DSA or ECDSA signature algorithm. In the case of KDH-enhanced, this hash is wrapped inside an encrypted Authenticator. Kerberos uses symmetric encryption and is by default configured to use AES256-CTS. Depending on the negotiated hash type[4], the length of this hash varies between 20 and 64 bytes. The length of an Authenticator therefore varies between 80 and 124 bytes (The contents of an Authenticator and their lengths are specified in Appendix A). Since symmetric encryption is more efficient than asymmetric encryption, this slightly larger input size does not result in a significant performance penalty, if at all.

The verification process of the `certificate verify` message is different however. Verification of an Authenticator equals decrypting the Authenticator. This equals one symmetric decryption operation. Verifying a digital signature requires decrypting the signature itself, plus verifying the rest of the certificate chain up to a trust anchor. A typical certificate chain has a length of 3. That means that in such case 2 additional digital signatures have to be verified (The signature under the root certificate need not be verified.). Additionally, the validity of the peer's certificate needs to be verified. This can be done via either a certificate revocation list (CRL) or an OSCP request. A CRL is downloaded only periodically, from the CA that issued the certificate. It is a bulk list that contains all revoked certificates for that CA. The Online Certificate Status Protocol (OCSP) is used to query the status of a certificate live. Both mechanisms require an additional signature verification. Either for the CRL itself, or for the OCSP response message.

From this analysis we observe that the KDH-enhanced mechanism requires 2 symmetric cryptographic operations (i.e. encryption and decryption of an Authenticator). Using X.509 requires at least 3 asymmetric cryptographic operations, namely creating the signature on the client, verifying the signature on the server, and then verifying the signature of the certificate containing the public key of the client. This is the best-case scenario where the certificate chain has length 1. In the average case where a certificate chain has length 3, the number of signatures that have to be verified is 3 (i.e. 2 for the certificates in the chain except the root, plus the signature generated on the client). A schematic representation of a trust chain with length 3 is depicted in Figure 2.2. Additionally, one extra signature for each OSCP request has to be verified. This brings the average total to 4 signatures (when OCSP is used, which is quite common nowadays). This yields 5 asymmetric cryptographic operations on average (i.e. generating one signature on the client, and verifying it on the server). We therefore conclude that the KDH-enhanced mechanism is more efficient than using X.509 for client authentication. The results of this comparison are summarized in Table 5.1.

	symmetric ops	asymmetric ops	workload (est.)
KDH-enhanced	2	0	2
X.509 (best)	0	3	[300,15000]
X.509 (avg)	0	5	[500,25000]

Table 5.1: Computational difference between X.509 and KDH-enhanced. The estimated workload equals the number of operations times the computational difference factor (see Section 5.1).

5.3.2 Comparing KDH-only with X.509

In this section the differences between KDH-only and X.509 are analyzed. Again, the message flow of the handshake is the same for both mechanisms. Only the contents, and the corresponding generation and verification processes, of some of these messages will differ. The messages that contain differences between KDH-only and X.509 are depicted graphically in Figure 5.1 and are marked in red.

[4]TLS currently supports MD5, SHA1, SHA256, SHA384 and SHA512 [43]. Since Kerberos does not support MD5 [41] and because this hash is considered insecure for TLS, we do not support it in the TLS-KDH specification.

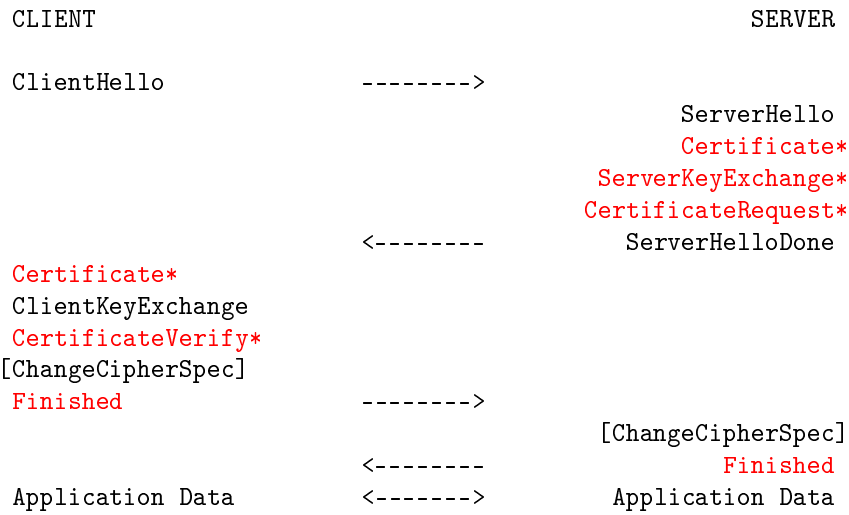


Figure 5.2: TLS handshake message flow. Messages in red mark the differences between KDH-only and X.509.

The server `Certificate` message is usually[5] omitted in KDH-only mode. That means that a server does not have to obtain certificate credentials, and that the server `Certificate` is not sent during the handshake. This yields an optimization for Phase 1 and Phase 2. The absence of a server certificate means that the client does not have to perform certificate validation. As was already explained in the previous section, certificate validation costs at least 1 asymmetric cryptographic operation, and on average 2 asymmetric cryptographic operations. Additionally, a CRL or OCSP response has to be verified which yields an additional asymmetric operation.

The key exchange method is limited to the elliptic-curve ephemeral Diffie-Hellman type (ECDHE) for all KDH-only cipher suites (see Section 3.1). In order to do a fair comparison, we only compare handshakes with ECDHE key exchange algorithms. The contents of the key exchange messages are therefore of the same type for both mechanisms. In X.509 however, the `Server Key Exchange` is digitally signed with the private key of the server. For KDH-only, this message contains no signature because there is no public/private key pair when performing Kerberos authentication. That means that the X.509 mechanism requires 1 asymmetric cryptographic operation for the generation of the digital signature versus no cryptographic operation in KDH-only mode.

For KDH-only, the same arguments apply for the `Certificate Request`, client `Certificate` and `Certificate Verify` messages as for KDH-enhanced (see Section 5.3.1). To summarize, the KDH mechanism requires only 2 symmetric cryptographical operations, whereas X.509 requires 3 asymmetric operations (best-case) or 5 asymmetric operations (average-case). The only difference here is that an Authenticator now also contains a **subkey**. The size of an Authenticator for KDH-only therefore varies between 109 and 170 bytes on average (see Appendix A). Since symmetric encryption is more efficient than asymmetric encryption, this slightly larger input size (i.e. [80,124] bytes for KDH-enhanced vs. [109,170] bytes for KDH-only) does not result in a significant performance penalty, if at all.

The final difference between the two mechanisms can be found in the `Finished` messages. KDH-only requires a longer **verify_data** field (see Section 3.1.1) that is furthermore based on a different premaster secret computation (see Section 3.2.3). The adapted premaster secret computation is based on an extra concatenation operation. Since an Authenticator is small, this concatenation can be done in constant time, and therefore adds no computational complexity. The **verify_data** is constructed from the output of the Pseudo Random Function (PRF) that is defined by the cipher

[5] Except when user-to-user authentication is desired (see Section 3.1.1).

	symmetric ops	asymmetric ops	PRF rounds	workload (est.)
KDH-only	2	0	1	3
X.509 (best)	0	6	1	[601,30001]
X.509 (avg)	0	11	1	[1101,55001]

Table 5.2: Computational difference between X.509 and KDH-only. The estimated workload equals the number of operations times the computational difference factor (see Section 5.1).

suite (see Section 3.2.5). The PRF of TLS is based on a keyed hash function, and is able to produce an output of arbitrary length. Its specification is given in Appendix B. Each KDH-only cipher suite specifies this **verify_data** length, and it varies between 32 and 64 bytes. The cipher suites for X.509 do not explicitly specify a **verify_data** length and therefore default to a length of 12 bytes. TLS' default PRF hash is SHA256. This hash has an output length of 32 bytes. For cipher suites other than the KDH-only ones (i.e. in this comparison thus the X.509 suites), this results in the execution of one PRF round of which the output is truncated to 12 bytes. The PRF for KDH-only cipher suites is defined to be equal to the hash algorithms used for MACs. The output lengths of these hash algorithms correspond to the respective **verify_data** lengths. That means that it also requires one PRF round to compute the **verify_data** for the KDH-only cipher suites. The length of the input for the PRF will be around 100 bytes and depends on the chosen hash algorithm [21]. The current X.509 cipher suites will always default to the SHA256 hash, whereas KDH-only cipher suites may use the SHA384[6] and SHA512 hashes. As can be learnt from hashes comparisons [8], the performance difference between the SHA256, SHA384 and SHA512 hashes can be neglected. We therefore conclude that there will be no significant computational difference for computing the `Finished` messages between the two mechanisms.

From this analysis we conclude that KDH-only is significantly more efficient than X.509, while achieving the same goal of mutual authentication. The results of the comparison are summarized in Table 5.2.

[6] SHA384 is actually SHA512 with the output truncated to 48 bytes.

Chapter 6

Proof of concept

To prove that the proposed TLS-KDH mechanism actually works, a proof of concept has been built. Since TLS-KDH extends the TLS protocol with Kerberos authentication capabilities, an existing TLS library has been chosen which was extended to incorporate the TLS-KDH functionality. This gives rise to the question which TLS library should we choose? Besides building a nice proof of concept we want the TLS-KDH mechanism to be actually used. TLS-KDH is part of the much larger ARPA2 project which incorporates it into their TLS Pool project [2]. In order for TLS-KDH to be easily adopted we should find a TLS library that is mature and widely used. If we look at the currently available TLS libraries that adhere to these requirements three candidates emerge:

- OpenSSL
- GnuTLS
- mbed TLS

We have chosen to implement the TLS-KDH mechanism into GnuTLS [33]. GnuTLS is a mature, community-driven, open-source TLS library that is widely used around the world. It was originally developed for applications of the GNU project but is now extensively used in other projects as well. The advantages of GnuTLS over OpenSSL and mbed TLS are its well-structured code base, its architecture that makes it easy to extend and the vast set of supported cryptographic algorithms and extensions [88].

6.1 Modifications to GnuTLS

At the time that the implementation phase of this project started the most recent version of GnuTLS was version 3.4.7. This is the version that we extended with the TLD-KDH mechanism. In order to implement this mechanism into GnuTLS, we needed to realize the following things. They are further addressed in this section.

- Define new cryptographic algorithms,
- Implement a new authentication mechanism,
- Implement two TLS Hello-message extensions,
- Extend existing TLS v1.2 behaviour.

GnuTLS is designed such that it can be easily extended. It has a modular architecture where new algorithms, authentication methods and extensions can easily be added. Code has been loosely coupled by defining abstract structures that link generic interfaces to concrete functions that implement specific behaviour for each of the available algorithms and extensions. This makes it fairly easy to add a new authentication mechanism. This is done by simply registering a new authentication method structure that points to the specific routines that implement the new behaviour. It is in principle not necessary to change existing code. The latter of course also depends on the functionality that is to be added. In our case for example, we had to adapt a lot of existing code when implementing the Raw Public Keys extension (see Section 6.1.3) in order to maintain compatibility with library versions that do not support this extension. The following sections describe the modifications that we made to GnuTLS.

6.1.1 New cryptographic algorithms

According to the specification [71] several new cryptographic algorithms must be defined. A new set of KDH-only cipher suites has been defined, as well as a new key exchange algorithm, public-key algorithm[1], signature types and certificate type (to accommodate Tickets). GnuTLS has been adapted to accommodate these new algorithms and artefacts.

6.1.2 New authentication mechanism

The key element of the TLS-KDH mechanism is the addition of Kerberos authentication capabilities to TLS. In order to do so we have implemented a new authentication method into GnuTLS and modified the existing certificate-based authentication methods to support client Kerberos Tickets. As discussed in Chapter 3, TLS-KDH introduces two distinct authentication modes: KDH-only and KDH-enhanced. These two additions to GnuTLS will now be discussed separately.

KDH-only

With KDH-only the server performs no explicit authentication step. Only the client is required to supply the server with a Kerberos Ticket. Because Kerberos accomplishes mutual authentication with just a single client Ticket (see Section 2.2), both parties are still mutually authenticated. In order to implement the KDH-only mechanism into GnuTLS, new cipher suites have been defined and a new authentication method has been added.

To keep the implementation as simple as possible we have tried to reuse as many of the existing certificate processing and handling capabilities as possible. Because certificates are implemented as binary strings it does not really matter whether such an object represents an X.509 certificate, OpenPGP certificate or Kerberos Ticket. Since a Kerberos Ticket or Authenticator does not have to be parsed by TLS but only has to be passed along, this even simplifies matters. Existing certificate code has therefore been adapted to accommodate the new certificate type that represents a Kerberos Ticket. Where necessary new routines have been written to generate and process such Kerberos Tickets and Authenticators.

GnuTLS defines authentication methods as abstract structures with a unified interface. This structure links the interface with concrete routines that implement specific handshake behaviour. An authentication structure is defined according to Listing 6.1. It contains a human readable name and a list of pointers to functions that fulfil specific parts of the handshake. If certain actions do not apply for the method, e.g. preshared key authentication does not require sending

[1] The name of this identifier is a bit misleading. It in fact defines the type of digital signature that is going to be used during the authentication and key exchange phases. Because TLS-KDH uses Kerberos Authenticators as a signature equivalent we need a means to distinguish between different signature mechanisms. That is way we need this new identifier.

any certificates, the respective fields can be set to `NULL`. The handshake procedure will detect this and will skip these steps in the handshake accordingly.

```
typedef struct
{
  const char *name;
  int (*gnutls_generate_server_certificate) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_certificate) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_server_kx) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_kx) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_cert_vrfy) (gnutls_session_t, gnutls_buffer_st *);
  int (*gnutls_generate_server_certificate_request) (gnutls_session_t,
                                                     gnutls_buffer_st *);

  int (*gnutls_process_server_certificate) (gnutls_session_t, opaque *,
                                            size_t);
  int (*gnutls_process_client_certificate) (gnutls_session_t, opaque *,
                                            size_t);
  int (*gnutls_process_server_kx) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_client_kx) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_client_cert_vrfy) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_server_certificate_request) (gnutls_session_t,
                                                    opaque *, size_t);
} mod_auth_st;
```

Listing 6.1: Authentication method definition structure.

name Contains a human readable name for the authentication method.

gnutls_generate_server_certificate Contains a pointer to a function that is responsible for generating a server `Certificate` message. This routine has been adapted to incorporate the new `KerberosTicket` certificate type. According to the Kerberos specification [65], a server-to-client Ticket is possible in order to facilitate user-to-user authentication. This feature is however not (yet) used in practice. Our TLS-KDH implementation has been made such that it is already capable of transmitting such Tickets. See Section 3.1.1 for more information about this feature.

gnutls_generate_client_certificate Contains a pointer to a function that is responsible for generating a client `Certificate` message. This routine has been adapted to incorporate the new `KerberosTicket` certificate type.

gnutls_generate_server_kx Contains a pointer to a function that generates a `Server Key Exchange` message. In case of KDH-only this is always is an elliptic-curve ephemeral Diffie-Hellman key exchange.

gnutls_generate_client_kx Contains a pointer to a function that generates a `Client Key Exchange` message. In case of KDH-only this is always is an elliptic-curve ephemeral Diffie-Hellman key exchange.

gnutls_generate_client_cert_vrfy Contains a pointer to a function that generates a `Certificate Verify` message. This routine has been adapted to transmit a Kerberos Authenticator instead of a regular digitally signed structure.

gnutls_generate_server_certificate_request Contains a pointer to a function that generates a `Certificate Request` message. This routine has been adapted to incorporate the new `kerberos_sign` ClientCertificateType [43].

gnutls_process_server_certificate Contains a pointer to a function that processes a server `Certificate` message. This routine has been adapted to incorporate the new `KerberosTicket` certificate type. Here the same comments holds as for the **gnutls_generate_server_certificate** field.

gnutls_process_client_certificate Contains a pointer to a function that processes a client `Certificate` message. This routine has been adapted to incorporate the new `KerberosTicket` certificate type.

gnutls_process_server_kx Contains a pointer to a function that processes a `Server Key Exchange` message. In case of KDH-only this is always is an elliptic-curve ephemeral Diffie-Hellman key exchange.

gnutls_process_client_kx Contains a pointer to a function that processes a `Client Key Exchange` message. In case of KDH-only this is always is an elliptic-curve ephemeral Diffie-Hellman key exchange.

gnutls_process_client_cert_vrfy Contains a pointer to a function that processes a `Certificate Verify` message. It has been adapted to be able to process a Kerberos Authenticator instead of a regular digitally signed structure.

gnutls_process_server_certificate_request Contains a pointer to a function that processes a `Certificate Request` message. This routine has been adapted to incorporate the new `kerberos_sign` ClientCertificateType.

KDH-enhanced

KDH-enhanced means that client authentication will be accomplished by means of a Kerberos Ticket. During a normal handshake (see Section 2.3.2) the server can send a `Certificate Request` to the client. Normally, if the client answers, the client answers with a certificate that matches the server's certificate type. This is the symmetric case that is described in Section 6.1.3. In KDH-enhanced mode, the client must supply a Kerberos Ticket whereas the server uses another certificate-based authentication method. This is described as the asymmetric case in Section 6.1.3. In order to implement this feature the code that is handling certificate authentication in GnuTLS has been adapted to incorporate the new certificate type (i.e. Kerberos Ticket). Additionally, a new mechanism was needed to negotiate different certificate types for the client and the server. Such an extension was already designed [90] but not yet implemented. It is further explained in Section 6.1.3. By implementing this new extension in GnuTLS, it is now possible for two communication peers to negotiate a `KerberosTicket` certificate type for the client and a X.509, OpenPGP or Raw Public Key certificate type for the server. This is exactly what the KDH-enhanced mechanism is designed to do: traditional server to client certificate authentication enhanced with Kerberos client to server authentication.

6.1.3 New extensions

In order to be able to communicate Kerberos Ticket constraints and to be able to negotiate the desired certificate types during the handshake, extra extensions had to be implemented. The first extension, Ticket Request Flags, is defined in the TLS-KDH specification (see Section 3.1.3). The second extension, Raw Public Keys, follows implicitly from the KDH-enhanced mechanism (see Section 3.2.1).

With TLS extensions[2] extra functionality can be added to the protocol. They rely on two generic extension mechanisms that aid in the negotiation of to-be-used extensions between client and server [11]. They define a generic and extensible field in the `Client Hello` and `Server Hello` messages that can be used to transmit extra extension-specific data (see Section 2.3.2). They are

[2]TLS extensions are a feature of the TLS protocol and should not be confused with extensions that extend the protocol itself (such as TLS-KDH).

at the same time used to determine which extensions should be enabled during the handshake and the subsequent session.

In GnuTLS, extensions can be added by defining a new extension structure and registering it in the `_gnutls_ext_init` routine. Secondly, the appropriate routines must be written and added to a file that contains the code for the extension. Finally, all new interface symbols must be properly exported via the `libgnutls.map` file. An example extension definition and its description is given in Appendix C.

The following two sections describe the purpose and functioning of the Ticket Request Flags and Raw Public Keys extensions.

Ticket Request Flags extension

TLS-KDH requires a mechanism to express Ticket constraints between a client and a server (see Section 3.1.3). Both might have different capabilities with respect to Kerberos Tickets and need to reach consensus about the to be exchanged Ticket(s). A client will therefore use the Ticket Request Flags (TRF) extension in its `Client Hello` to communicate to the server which flags it understands and is able to fulfil. In response, the server will pick a subset of these flags that it wishes to be fulfilled by the client and put these flags in its `Server Hello`. TLS itself has nothing to do with these flags and merely acts as a messenger. The flags originate from a Kerberos-aware client at the Application Layer and will at the other end of the connection be directly passed on to a Kerberos-aware client at the Application Layer. It is these applications that interpret the flags and supply TLS with the appropriate Tickets during the authentication process.

Raw Public Keys extension

TLS has been extended to support OpenPGP certificates for authentication [55]. This requires that both parties agree on the type of certificate that is going the be used. The `cert_type` extension [55] implements this negotiation during the handshake by extending the `Client Hello` and `Server Hello` messages with the appropriate data. The client sends a list of certificate types that it supports in order of preference to the server. The server then chooses the client's most preferred certificate type that it supports. Both parties then agree on the same certificate type to use for authentication (i.e. either X.509 or OpenPGP). This mechanism is what we call symmetric (see Section 3.2.1). TLS-KDH however introduces the desire for an asymmetric mechanism. In KDH-enhanced mode, server to client authentication is done in a traditional manner[3], while client to server authentication is accomplished via Kerberos. This means that a server must be able to send, for example, an X.509 server certificate to the client, while the client must be able to send a Kerberos Ticket to the server. These certificate types are not the same and hence we need an asymmetric certificate type negotiation mechanism. Luckily such a mechanism has been defined in [90]. This RFC defines the notion of a raw public key as opposed to full fledged certificates, and defines two extensions that enable certificate type negotiation. The Client Certificate Type extension defines a mechanism to negotiate a certificate type for the client and the Server Certificate Type extension defines a mechanism to negotiate a certificate type for the server.

Both extensions extend the `Client Hello` and `Server Hello` messages. For the Client Certificate Type extension, the client sends a list of supported certificate types to the server that it is able to supply in response to a `Certificate Request` message. This list is sorted in order of preference by the client. Upon receipt, the server chooses from this list a certificate type that it is able to process. The server then sends this choice to the client via its `Server Hello`. Both parties have now agreed on a client certificate type. For the Server Certificate Type extension,

[3] Traditional here means by using the already supported certificate mechanisms X.509 or OpenPGP.

the client sends a list of supported certificate types to the server that it is able to process. This list is also sorted in order of preference by the client. Upon receipt, the server chooses from this list a certificate type of which it has credentials to supply. The server then sends this choice to the client via its `Server Hello`. Both parties have now agreed on a server certificate type. The server will now send a certificate of this type to the client in a subsequent `Certificate` message.

The `cert_type` extension and the Client and Server Certificate Type extensions have overlapping but also contradictory functionality. Both negotiate certificate types but they do so in a different manner. Since one can negotiate both symmetric and asymmetric certificate types with the Client and Server Certificate Type extensions these extensions obsolete, in our opinion, the more limited `cert_type` extension.

GnuTLS however, has built-in support for OpenPGP certificates and corresponding negotiation mechanisms[4] [55], but not for the Raw Public Keys extension. Because we need this functionality for the TLS-KDH mechanism we have implemented the Raw Public Keys TLS extension in GnuTLS according to its specification [90]. In order to maintain compatibility with the `cert_type` extension [55] a lot of work needed to be done. The two extensions cannot be used at the same time by GnuTLS, but fully disabling the cert_type extension would break compatibility with older clients that do not know the new extensions. The implementation is therefore crafted such that both extensions can co-exist. This is currently realized by letting the library user explicitly choose to use the new Client and Server Certificate Type extensions by setting an option in the Priority strings[5] [34]. The rationale behind this choice is that when we force a user to explicitly enable some functionality, that we assume that she also knows how to use this functionality properly. Because the new extensions need to be enabled explicitly they cannot be used accidentally and thereby breaking existing systems without knowing. These same Priority strings can then be used to specify what certificate types are enabled for the client and the server, and what their respective priorities are. By enabling the new extensions the `cert_type` extension is automatically ignored.

6.1.4 TLS v1.2: verify_data length

According to the TLS v1.2 specification [21] a cipher suite may define a custom **verify_data** length. The **verify_data** length is the length of the output of the pseudorandom function (i.e. the verification data) that is computed over the **master_secret**, the **finished_label** and the **Hash(handshake_messages)** in the `Finished` messages (see Section 2.3.2). Prior to TLS v1.2 this length was fixed to 12 octets, which is currently still the default. As of v1.2 cipher suites may define their own length as long as it has a minimum length of 12 octets. This feature was not yet implemented in GnuTLS. All currently implemented cipher suites used the default of 12 and there was no mechanism to deviate from this. Because the new TLS-KDH cipher suites define custom **verify_data** lengths we have implemented this feature and applied it as prescribed by the TLS-KDH specification.

6.2 Open-source contributions

The implementation of TLS-KDH in GnuTLS yielded several open-source contributions. Since GnuTLS is released under the LGPL all modifications are also released under the LGPL and are therefore free. All source code has been made available in our GitHub repository at `https://github.com/arpa2/gnutls-kdh`. At the time of writing we have released an Alpha release that needs further testing. Eventually we are going to provide a patch that will be offered to GnuTLS

[4] According to [88] GnuTLS is the only TLS library that supports OpenPGP certificates. Additionally, none of these TLS libraries currently supports Raw Public Keys.

[5] Priority strings are used by GnuTLS to customize library behaviour and for instance to allow or disallow certain certificate types.

to be incorporated into their main branch so that the TLS-KDH mechanism will be automatically available to all GnuTLS users.

During development, several TLS extensions have been implemented that were necessary for a properly functioning TLS-KDH mechanism. These are readily donated to the open-source community and include:

- verify_data length functionality from the TLS 1.2 specification [21]
- Client Certificate Type extension [90]
- Server Certificate Type extension [90]
- Raw Public Keys extension [90]

Chapter 7

Related work

In this chapter, some related work with respect to Kerberos and TLS is discussed.

7.1 Kerberos alternatives

As was touched upon in the Introduction, several network-based authentication systems exist. The most common alternatives for Kerberos are TACACS, RADIUS and Diameter. They are so-called Authentication, Authorization, and Accounting (AAA) systems. To give a comparison between these mechanisms they are briefly discussed here.

TACACS (Terminal Access Controller Access-Control System) is a set of protocols handling remote authentication and related services for networked access control. It runs on top of TCP or UDP, and only supports username/password-based authentication. Additionally, it does not support any form of encryption. TACACS has generally been replaced by TACACS+ and RADIUS. TACACS+ is an extension of TACACS designed by CISCO that encrypts the full contents of each packet. It is however a new protocol that is not compatible with other TACACS versions. TACACS+ is not an official IETF standard (yet), although it has been submitted as an Internet-Draft [17]. TACACS+ is designed for device administration AAA, to authenticate and authorize users into mainframe and Unix terminals, and other terminals or consoles.

RADIUS (Remote Access Dial-In User Service) is a network protocol designed for network access AAA. It is often used by ISPs and enterprises to manage access to the Internet, internal networks and wireless networks. RADIUS is however lacking proper cryptographic mechanisms. First of all it encrypts only the users' password as it travels from the RADIUS client to the RADIUS server. All other information such as the username, authorization and accounting are transmitted in clear text. Secondly, this password encryption is based on the MD5 algorithm, which has been proven to be insecure. RADIUS was originally designed to use UDP as the underlying Transport Layer protocol. It has however been extended with the RadSec extension [89] that enables RADIUS over TCP, and provides better security through TLS. As a side note, the eduroam network is based on RADIUS and RadSec.

Diameter was originally designed as a successor of RADIUS because RADIUS has issues with reliability, scalability, security and flexibility. Diameter is a AAA framework designed to overcome these issues. It is built to operate on top of the Transport Layer protocols TCP and SCTP. Additionally, it is able to use TLS for improved security. Diameter serves as a base protocol for derived protocols called Diameter Applications. These applications can define new command codes and/or new mandatory Attribute-Value Pairs (AVPs). Examples of such applications are MobileIP [14], Diameter Session Initiation Protocol [31] (SIP) and Diameter Extensible Authentication Protocol [24] (EAP). Although Diameter was designed to replace RADIUS, this is generally

not yet the case. One of the main reasons for that is that switches and access points typically implement RADIUS, but not Diameter. Currently, Diameter is largely used in the wireless mobile space whereas RADIUS is used elsewhere.

TACACS, RADIUS and Diameter primarily focus on regulating network access (i.e. *device-to-network* authentication or *client-to-network* authentication). Kerberos however, focusses on regulating service access (i.e. client-to-service authentication). It also supports client-to-client authentication [65]. This is useful in situations where the server is running as an unprivileged user and might not have access to system keys, or in the context of peer-to-peer (P2P) applications. Kerberos therefore offers more fine-grained control and flexibility. Additionally, Kerberos gives the client a credential (i.e. a Ticket) with which the client can authenticate itself to other services in the realm. This is the basis of Kerberos' single sign-on functionality (see Section 2.2.2). The other mechanisms only give a "yes" or "no" answer, with respect to the authentication question.

7.2 Other combinations of Kerberos and TLS

The combination of Kerberos and TLS is not new. A few mechanisms have been proposed but they all have their own limitations or shortcomings. They will be briefly discussed in the next sections.

7.2.1 Kerberos cipher suites in TLS

RFC2712 [56] defines a new Kerberos authentication mechanism and corresponding cipher suites for TLS. The mechanism integrates with the existing TLS message flow and is therefore easy to implement. Because no regular certificate authentication needs to take place, the `Certificate`, `Server Key Exchange`, `Certificate Request` and `Certificate Verify` messages can be omitted. This is allowed according to the TLS protocol specification. Only the `Client Key Exchange` message will be extended and used to communicate an encrypted pre-master secret, a Kerberos Ticket and optionally an Authenticator from the client to the server. To that extend, a Kerberos data wrapper has been defined to hold all the aforementioned data (see Listing 7.1). If the client and server negotiate a Kerberos key exchange algorithm, this wrapper is used to transmit the Kerberos data from the client to the server.

```
struct
{
  select (KeyExchangeAlgorithm)
  {
    case krb5:             KerberosWrapper;       /* new addition */
    case rsa:              EncryptedPreMasterSecret;
    case diffie_hellman:   ClientDiffieHellmanPublic;
  } Exchange_keys;

} ClientKeyExchange;

struct
{
  opaque Ticket;
  opaque Authenticator;            /* optional */
  opaque EncryptedPreMasterSecret; /* encrypted with the session key
                                      which is sealed in the ticket */
} KerberosWrapper;                 /* new addition */
```

Listing 7.1: Kerberos data wrapper definition.

The premaster secret will be encrypted with the (client ↔ server) session key that is embedded in the Ticket. The server can prove its authenticity to the client by showing that it is able to decrypt the Ticket, thereby retrieving the premaster secret and finally computing the same master secret as the client. Vice versa, the client proves its authenticity to the server by showing that it encrypted the premaster secret with the same key that was wrapped inside the Ticket by the KDC. When both parties end up with the same master secret, mutual authentication (based on the `Finished` messages) has been established.

Pros & cons

The advantage of this method is that it is simple and therefore easy to implement. Only a set of new cipher suites and a new `Client Kex Exchange` message need to be defined. The disadvantage is that is supports no Perfect Forward Secrecy (PFS) because the premaster secret is computed comparable to the RSA case in standard certificate-based authentication (see Section 2.3.2). Compromise of the long-term secret between a user and the KDC potentially reveals previous sessions (see Section 4.1).

The difference with TLS-KDH is that the latter requires an elliptic-curve ephemeral Diffie-Hellman key exchange and therefore enforces PFS on all sessions. Additionally, an extra mode of operation is available that combines Kerberos client authentication with regular certificate based server authentication. The mechanism described in this section only supports a "Kerberos only" mode.

7.2.2 Krb5 STARTTLS

RFC6251 [46] defines a mechanism to use Kerberos version 5 over the TLS protocol. It realizes this by upgrading a normal plaintext TCP connection to an encrypted TCP connection by defining an opportunistic TLS extension (often called STARTTLS) called Krb5 STARTTLS. The extension only upgrades connections between clients and KDCs but this is enough to accomplish the desired security goals. Using Kerberos over TLS ensures data confidentiality for all the contents of Kerberos messages. Parts of Kerberos messages are unencrypted and contain information such as principal names, the encryption types supported by the client and the lifetime of Tickets. Leaking this data might form a privacy risk. Additionally, TLS protects against downgrade attacks affecting the encryption types and pre-authentication data negotiation because these fields are sent without integrity or privacy protection.

Kerberos version 5 defines a mechanism for TCP transport extension [45]. TCP specific Kerberos extensions can be enabled by setting the (reserved) high bit of the request length field. When this bit is set, the 31 remaining bits can be used as a bitmask that codes for the activation of certain Kerberos TCP extensions. A client that supports the Krb5 STARTTLS extension can then request a KDC to negotiate a TLS-secured channel. For this to work the respective KDC must also support this extension. If both parties are willing to employ TLS then first a normal TLS handshake will commence. After a successful handshake is completed, the regular Kerberos protocol will be executed over the secured TLS channel. The extended protocol has been graphically depicted in Figure 7.1.

Pros & cons

The advantage of this method is that it requires no modifications to TLS. Only Krb5 clients and KDCs need to support the STARTTLS extension. This removes dependencies for a specific TLS library. Additionally, Perfect Forward Secrecy is possible when a TLS cipher suite with ephemeral Diffie-Hellman is chosen. The disadvantage is that this method supports no actual Kerberos authentication within TLS. This results in extra overhead because two separate authentication

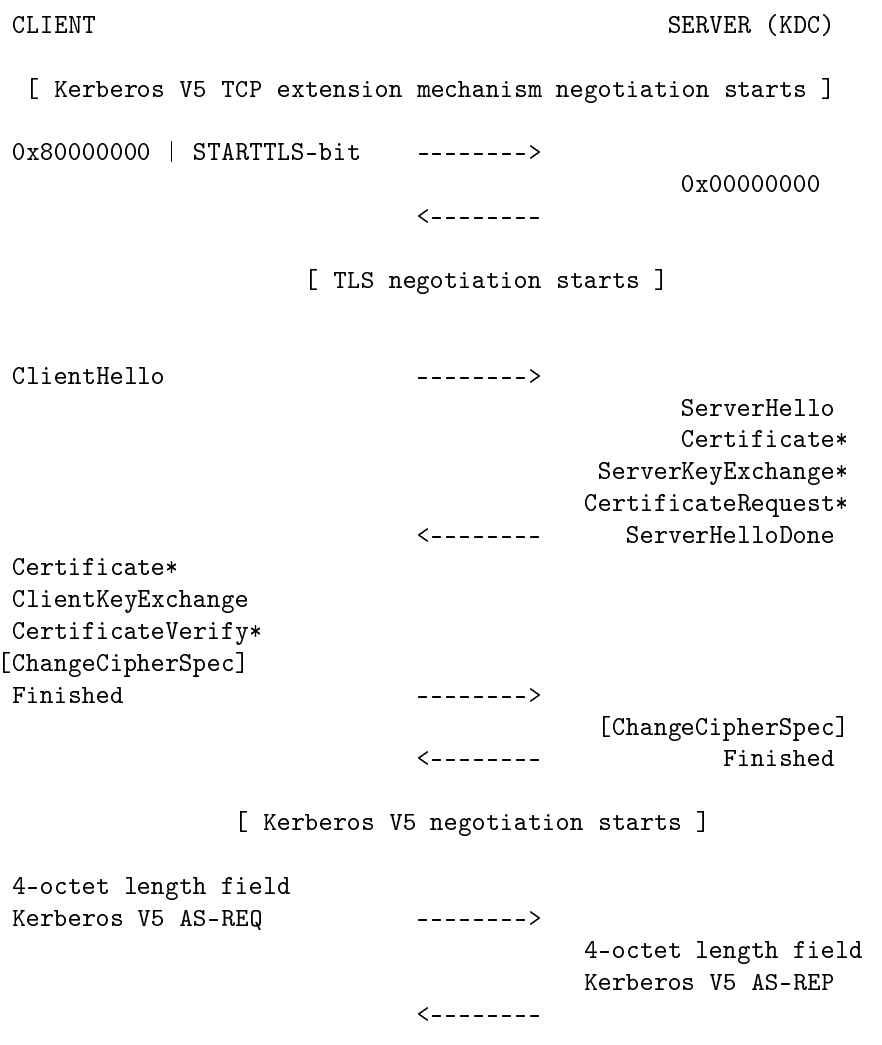


Figure 7.1: Message exchange for the Krb5 STARTTLS extension.

mechanisms have to be executed: 1) a full TLS handshake and authentication, 2) Kerberos authentication. This extra overhead can be overcome by choosing an anonymous cipher suite and skipping the authentication steps for TLS. This yields an extra layer of encryption (with potentially PFS) that protects against passive attackers. Unauthenticated key exchanges are however vulnerable for active attackers that want to perform a Man-in-the-Middle attack. Finally, this method is not designed to work with datagram TLS (DTLS).

The difference with TLS-KDH is that TLS-KDH protects the channel between a client and a server instead of the channel between a client and the KDC. That means that with the method described here, a rogue KDC is still able to learn client-to-server session data because this channel is not protected with PFS[1]. TLS-KDH is not vulnerable for this attack. Furthermore, TLS-KDH is integrated into TLS as an authentication method thereby overcoming the overhead that results from an extra handshake. Additionally, TLS-KDH enforces PFS on all sessions whereas this method let it depend on the negotiated cipher suite. Finally, TLS-KDH also works with DTLS.

[1] It should be noted however, that when there is a rogue KDC that there is a big problem anyway.

7.3 Kerberos Realm Crossover

Within Kerberos, a KDC is responsible for one or more realms. It has knowledge of all the principals within that realm and is able to issue Tickets for them. The KDC is the trusted third party that enables authentication between all principals in the realm. As an example, a typical realm could be a university or a company. It is however possible in practice that a client from realm A wants to authenticate itself to a service in realm B. Both may be under control of a different KDC. In such a case cross-realm authentication is necessary. Kerberos has been designed such that it is possible to perform realm crossover [64]. Cooperating KDCs must therefore "register" with each other by agreeing on a cross-realm key. A principal from realm A that wants to authenticate to an application server in a remote realm B then asks its own KDC (realm A) for a Ticket Granting Ticket of the remote realm (realm B). The principal then uses this TGT in a ticket granting exchange (see Section 2.2.2) with the remote KDC to get a Ticket for a service in realm B. During the ticket granting exchange, the KDC of realm B sees that the TGT was issued by a foreign KDC. It then tries to validate the foreign TGT by looking up the appropriate cross-realm key and by trying to decrypt the Ticket. If all succeeds, the KDC of realm B issues a session key and a Ticket to the requesting client. The client is then able to finalize its cross-realm authentication. It is also possible that a client requests a service in its local realm, but that it is not aware that the requested service actually resides in a remote realm. The local KDC might know this and send the client a TGT for the remote realm. The client recognises this TGT and uses it to request a Ticket at the remote KDC as before.

The exchange of cross-realm keys between KDCs is a manual task. System administrators must explicitly exchange keys with administrators from other realms and configure their KDCs such that they know specific other KDCs. This explicit binding is neither scalable nor administrator-friendly. Automatic retrieval or exchange of cross-realm keys is therefore desirable. A solution for this has been proposed in [5] and is called the KXOVER protocol. It uses DNS secured with DNSSEC to reliably lookup remote KDCs. Additionally, DANE [39] is used to exchange public-key material between foreign KDCs which can be used to verify each other's digital signatures. A proof of concept has been built, but further work is needed to process lessons learnt about the protocol and to create a mature and robust standard.

Realm crossover is an important feature that greatly extends the authentication capabilities of Kerberos by connecting separate realms. It is also useful in the context of TLS-KDH because now TLS-KDH can be used to secure a connection between a client and a server that belong to different realms.

Chapter 8

Conclusions and Future work

In this thesis we have addressed the following research questions. This chapter summarizes the results and suggests future work that can be done.

1.1 *Is it possible to design a system that combines the strong authentication properties of Kerberos with strong cryptographic properties of TLS?*

We have analyzed the preliminary specification of TLS-KDH: a mechanism that combines Kerberos with TLS. Based on this analysis we proposed some improvements to the original design. These recommendations can be viewed in Section 3.2. They resulted in an updated Internet-Draft for the TLS-KDH specification that can be found at: `https://tools.ietf.org/html/draft-vanrein-tls-kdh-04`. By building a proof of concept implementation we have shown that the design actually works and that it is possible to integrate Kerberos into TLS. We have extended the widely used GnuTLS library with TLS-KDH functionality and donated all code to the open-source community. We hope that our work can be easily adopted this way. We are proud that our work will be used for ARPA2's TLS Pool project [2] and that it finds actual practical applicability.

1.2 *Security analysis of TLS-KDH. What are its weak and strong points?*

Additionally, a security analysis of the improved design has been conducted. We may conclude that TLS-KDH improves Kerberos security by always providing encryption on all sessions. Furthermore, perfect forward secrecy is enforced. TLS-KDH does not introduce any new security issues for TLS. Existing security issues may still apply however (see Section 4.2). For Kerberos application servers that implement TLS-KDH we found the possibility for a replay attack, which we named the "Out-of-Band Replay Attack". This attack can only occur under specific conditions (see Section 4.3.2), but nevertheless it is important to consider when designing a TLS-KDH based system. TLS-KDH is based on the centrally managed Kerberos infrastructure and therefore forms a strong alternative to PKI-based authentication.

1.3 *How does TLS-KDH perform with respect to X.509?*

From the performance analysis that we have conducted, we may conclude that the TLS-KDH mechanism is significantly more efficient, with respect to the computational complexity, than TLS with X.509. This difference can be attributed to the fact that TLS-KDH uses symmetric cryptography whereas X.509 is based on asymmetric cryptography. The conducted performance analysis was theoretical in nature and only analyzed the designs of both mechanisms. Depending on the used computational difference factor of symmetric and asymmetric cryptographic algorithms[1],

[1] We found several factors in literature that vary significantly depending on the hardware used for testing.

the workload differences between TLS-KDH and TLS with X.509 range from a factor 150 to 18000, in favour of TLS-KDH. The results of the analysis can be found in Tables 5.1 and 5.2. The actual "real-life" performance depends on many more factors such as the underlying hardware and network latencies.

Although a new draft specification has been written and a proof-of-concept has been built, future work is needed to achieve a mature and accepted standard. In order to get there, further review of the design and corresponding specification is necessary, by experts in the field. Additionally, further testing of the implementation needs to be done to make sure that all code is stable and can be submitted for inclusion in the main GnuTLS branch. Finally, real benchmarks of the TLS-KDH mechanism may be performed to come up with real performance numbers on a given platform.

Bibliography

[1] ARPA2. *ARPA2.net homepage*. 2016. URL: https://arpa2.net/ (visited on 07/20/2016).

[2] ARPA2. *ARPA2.net - TLS Pool project page*. 2016. URL: http://tlspool.arpa2.net/ (visited on 07/28/2016).

[3] ARPA2. *ARPA2.net - TLS-KDH project page*. 2016. URL: https://tls-kdh.arpa2.net/ (visited on 07/20/2016).

[4] M. Badra and I. Hajjeh. *ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)*. RFC 5489. RFC Editor, Mar. 2009.

[5] Orial Caño Bellatriu. "Kerberos Realm Crossover." Eindhoven University of Technology, 2016. URL: http://research.arpa2.org/library/bellatriu-2016-kerberos-realm-crossover.pdf.

[6] S. M. Bellovin and M. Merritt. "Limitations of the Kerberos Authentication System." In: *SIGCOMM Comput. Commun. Rev.* 20.5 (Oct. 1990), pp. 119–132. ISSN: 0146-4833. DOI: 10.1145/381906.381946. URL: http://doi.acm.org/10.1145/381906.381946.

[7] Kevin Benton and Ty Bross. "Timing Analysis of SSL/TLS Man in the Middle Attacks." In: *CoRR* abs/1308.3559 (2013). URL: http://arxiv.org/abs/1308.3559.

[8] Daniel J. Bernstein and Tanja Lange (editors). *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. 2016. URL: https://bench.cr.yp.to (visited on 08/07/2016).

[9] K. Bhargavan et al. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. RFC 7627. RFC Editor, Sept. 2015.

[10] S. Blake-Wilson et al. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492. http://www.rfc-editor.org/rfc/rfc4492.txt. RFC Editor, May 2006. URL: http://www.rfc-editor.org/rfc/rfc4492.txt.

[11] S. Blake-Wilson et al. *Transport Layer Security (TLS) Extensions*. RFC 3546. RFC Editor, June 2003.

[12] Alexandra Boldyreva and Virendra Kumar. "Provable-security analysis of authenticated encryption in Kerberos." In: *IET Information Security* 5.4 (2011), pp. 207–219. DOI: 10.1049/iet-ifs.2011.0041. URL: http://dx.doi.org/10.1049/iet-ifs.2011.0041.

[13] I. Branovic, R. Giorgi, and E. Martinelli. "Memory Performance of Public-Key cryptography Methods in Mobile Environments." In: *ACM SIGARCH Workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA-03)*. New Orleans, LA, USA, Sept. 2003, pp. 24–31. URL: http://www.dii.unisi.it/~giorgi/papers/Branovic03a.pdf.

[14] P. Calhoun et al. *Diameter Mobile IPv4 Application*. RFC 4004. RFC Editor, Aug. 2005.

[15] J. Callas et al. *OpenPGP Message Format*. RFC 4880. http://www.rfc-editor.org/rfc/rfc4880.txt. RFC Editor, Nov. 2007. URL: http://www.rfc-editor.org/rfc/rfc4880.txt.

[16] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. http://www.rfc-editor.org/rfc/rfc5280.txt. RFC Editor, May 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[17] Lol Grant D. Carrel. *The TACACS+ Protocol Version 1.78*. Internet-Draft draft-grant-tacacs-02. http://www.ietf.org/internet-drafts/draft-grant-tacacs-02.txt. IETF Secretariat, Jan. 1997. URL: http://www.ietf.org/internet-drafts/draft-grant-tacacs-02.txt.

[18] Italo Dacosta, Mustaque Ahamad, and Patrick Traynor. "Trust No One Else: Detecting MITM Attacks against SSL/TLS without Third-Parties." In: *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 199–216. ISBN: 978-3-642-33167-1. DOI: 10.1007/978-3-642-33167-1_12. URL: http://dx.doi.org/10.1007/978-3-642-33167-1_12.

[19] P. Degano et al. "Automated Reasoning for Security Protocol Analysis Formal analysis of Kerberos 5." In: *Theoretical Computer Science* 367.1 (2006), pp. 57–87. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2006.08.040. URL: http://www.sciencedirect.com/science/article/pii/S0304397506005743.

[20] Dorothy E. Denning and Giovanni Maria Sacco. "Timestamps in Key Distribution Protocols." In: *Commun. ACM* 24.8 (Aug. 1981), pp. 533–536. ISSN: 0001-0782. DOI: 10.1145/358722.358740. URL: http://doi.acm.org/10.1145/358722.358740.

[21] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. http://www.rfc-editor.org/rfc/rfc5246.txt. RFC Editor, Aug. 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[22] Ling Dong and Kefei Chen. "Security Analysis of Real World Protocols." In: *Cryptographic Protocol: Security Analysis Based on Trusted Freshness*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 153–213. ISBN: 978-3-642-24073-7. DOI: 10.1007/978-3-642-24073-7_5. URL: http://dx.doi.org/10.1007/978-3-642-24073-7_5.

[23] Diaa Salama Abdul. Elminaam, Hatem Mohamed Abdul Kader, and Mohie Mohamed Hadhoud. "Performance Evaluation of Symmetric Encryption Algorithms." In: *IJCSNS International Journal of Computer Science and Network Security*. Vol. 8. 12. Dec. 2008, pp. 280–286. URL: http://paper.ijcsns.org/07_book/200812/20081240.pdf (visited on 08/08/2016).

[24] P. Eronen, T. Hiller, and G. Zorn. *Diameter Extensible Authentication Protocol (EAP) Application*. RFC 4072. RFC Editor, Aug. 2005.

[25] P. Eronen and H. Tschofenig. *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. RFC 4279. http://www.rfc-editor.org/rfc/rfc4279.txt. RFC Editor, Dec. 2005. URL: http://www.rfc-editor.org/rfc/rfc4279.txt.

[26] V. Fajardo et al. *Diameter Base Protocol*. RFC 6733. http://www.rfc-editor.org/rfc/rfc6733.txt. RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6733.txt.

[27] fgrieu and Thomas. *Why is asymmetric cryptography bad for huge data?* June 16, 2014. URL: https://crypto.stackexchange.com/questions/5782/why-is-asymmetric-cryptography-bad-for-huge-data (visited on 08/10/2016).

[28] C. Finseth. *An Access Control Protocol, Sometimes Called TACACS*. RFC 1492. RFC Editor, July 1993.

[29] P. Ford-Hutchinson. *Securing FTP with TLS*. RFC 4217. http://www.rfc-editor.org/rfc/rfc4217.txt. RFC Editor, Oct. 2005. URL: http://www.rfc-editor.org/rfc/rfc4217.txt.

[30] Sebastian Gajek et al. "Universally Composable Security Analysis of TLS." In: *Provable Security: Second International Conference, ProvSec 2008, Shanghai, China, October 30 - November 1, 2008. Proceedings.* Ed. by Joonsang Baek et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 313–327. ISBN: 978-3-540-88733-1. DOI: 10.1007/978-3-540-88733-1_22. URL: http://dx.doi.org/10.1007/978-3-540-88733-1_22.

[31] M. Garcia-Martin et al. *Diameter Session Initiation Protocol (SIP) Application.* RFC 4740. RFC Editor, Nov. 2006.

[32] GNU. *GNU Shishi.* 2016. URL: https://www.gnu.org/software/shishi/ (visited on 07/21/2016).

[33] GnuTLS. *GnuTLS homepage.* 2016. URL: http://gnutls.org/ (visited on 07/28/2016).

[34] GnuTLS. *GnuTLS manual - Priority strings.* 2016. URL: https://gnutls.org/manual/html_node/Priority-Strings.html (visited on 07/28/2016).

[35] GnuTLS. *GnuTLS manual - TLS layers.* 2016. URL: http://www.gnutls.org/manual/html_node/TLS-layers.html#TLS-layers (visited on 07/28/2016).

[36] S. Hartman and L. Zhu. *A Generalized Framework for Kerberos Pre-Authentication.* RFC 6113. RFC Editor, Apr. 2011.

[37] Heimdal. *Heimdal Kerberos.* 2016. URL: https://www.h5l.org/index.html (visited on 07/21/2016).

[38] P. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security.* RFC 3207. http://www.rfc-editor.org/rfc/rfc3207.txt. RFC Editor, Feb. 2002. URL: http://www.rfc-editor.org/rfc/rfc3207.txt.

[39] P. Hoffman and J. Schlyter. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA.* RFC 6698. http://www.rfc-editor.org/rfc/rfc6698.txt. RFC Editor, Aug. 2012. URL: http://www.rfc-editor.org/rfc/rfc6698.txt.

[40] Ralph Holz et al. "X.509 Forensics: Detecting and Localising the SSL/TLS Men-in-the-Middle." In: *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings.* Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 217–234. ISBN: 978-3-642-33167-1. DOI: 10.1007/978-3-642-33167-1_13. URL: http://dx.doi.org/10.1007/978-3-642-33167-1_13.

[41] IANA. *Kerberos Parameters.* 2016. URL: https://www.iana.org/assignments/kerberos-parameters/kerberos-parameters.xhtml (visited on 08/10/2016).

[42] IANA. *Transport Layer Security (TLS) Extensions.* 2016. URL: https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml (visited on 08/06/2016).

[43] IANA. *Transport Layer Security (TLS) Parameters.* 2016. URL: https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml (visited on 08/05/2016).

[44] ITU. *ITU-T X.680A. bstract Syntax Notation One (ASN.1): Specification of basic notation.* Internet Standard. ITU, 2015. URL: http://handle.itu.int/11.1002/1000/12479 (visited on 08/11/2016).

[45] S. Josefsson. *Extended Kerberos Version 5 Key Distribution Center (KDC) Exchanges over TCP.* RFC 5021. RFC Editor, Aug. 2007.

[46] S. Josefsson. *Using Kerberos Version 5 over the Transport Layer Security (TLS) Protocol.* RFC 6251. RFC Editor, May 2011.

[47] Nikolaos Karapanos and Srdjan Capkun. "On the Effective Prevention of TLS Man-in-the-Middle Attacks in Web Applications." In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 671–686. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/karapanos.

[48] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. http://www.rfc-editor.org/rfc/rfc4301.txt. RFC Editor, Dec. 2005. URL: http://www.rfc-editor.org/rfc/rfc4301.txt.

[49] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis." In: *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448. ISBN: 978-3-642-40041-4. DOI: 10.1007/978-3-642-40041-4_24. URL: http://dx.doi.org/10.1007/978-3-642-40041-4_24.

[50] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. 5th. Addison-Wesley Publishing Company, 2010. ISBN: 0131365487, 9780131365483.

[51] Daesung Kwon et al. "New Block Cipher: ARIA." In: *Information Security and Cryptology - ICISC 2003: 6th International Conference, Seoul, Korea, November 27-28, 2003. Revised Papers*. Ed. by Jong-In Lim and Dong-Hoon Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 432–445. ISBN: 978-3-540-24691-6. DOI: 10.1007/978-3-540-24691-6_32. URL: http://dx.doi.org/10.1007/978-3-540-24691-6_32.

[52] Arjen K. Lenstra and Eric R. Verheul. "Selecting Cryptographic Key Sizes." In: *Journal of Cryptology* 14 (1999), pp. 255–293.

[53] Albert Levi and Erkay Savas. "Performance Evaluation of Public-Key Cryptosystem Operations in WTLS Protocol." In: *Proceedings of the Eighth IEEE Symposium on Computers and Communications (ISCC 2003), 30 June - 3 July 2003, Kiris-Kemer, Turkey*. 2003, pp. 1245–1250. DOI: 10.1109/ISCC.2003.1214285. URL: http://dx.doi.org/10.1109/ISCC.2003.1214285.

[54] J. Linn. *Generic Security Service Application Program Interface Version 2, Update 1*. RFC 2743. RFC Editor, Jan. 2000.

[55] N. Mavrogiannopoulos and D. Gillmor. *Using OpenPGP Keys for Transport Layer Security (TLS) Authentication*. RFC 6091. RFC Editor, Feb. 2011.

[56] A. Medvinsky and M. Hur. *Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)*. RFC 2712. RFC Editor, Oct. 1999.

[57] A. Melnikov and K. Zeilenga. *Simple Authentication and Security Layer (SASL)*. RFC 4422. RFC Editor, June 2006.

[58] MIT. *MIT Kerberos Distribution Page*. 2016. URL: http://web.mit.edu/kerberos/dist/ (visited on 07/21/2016).

[59] MIT. *MIT Kerberos website*. 2016. URL: http://web.mit.edu/kerberos/ (visited on 07/20/2016).

[60] P. Morrissey, N. P. Smart, and B. Warinschi. "A Modular Security Analysis of the TLS Handshake Protocol." In: *Advances in Cryptology - ASIACRYPT 2008: 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*. Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 55–73. ISBN: 978-3-540-89255-7. DOI: 10.1007/978-3-540-89255-7_5. URL: http://dx.doi.org/10.1007/978-3-540-89255-7_5.

[61] A. Nadeem and M.Y. Javed. "A Performance Comparison of Data Encryption Algorithms." In: *Mobile Networks and Applications*. 2005. URL: https://www.researchgate.net/profile/Muhammad_Javed56/publication/4224931_A_Performance_Comparison_of_Data_Encryption_Algorithms/links/553f1de10cf20184050fa739.pdf (visited on 08/08/2016).

[62] Monica Nesi and Giuseppina Rucci. "Proceedings of the Second Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA 2005) Formalizing and Analyzing the Needham-Schroeder Symmetric-Key Protocol by Rewriting." In: *Electronic Notes in Theoretical Computer Science* 135.1 (2005), pp. 95–114. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2005.06.002. URL: http://www.sciencedirect.com/science/article/pii/S1571066105050541.

[63] B. C. Neuman. "Proxy-based authorization and accounting for distributed systems." In: *Proceedings of the 13th International Conference on Distributed Computing Systems*. IEEE, May 1993, pp. 283–291. ISBN: 0-8186-3770-6. DOI: 10.1109/ICDCS.1993.287698.

[64] B. Clifford Neuman and Theodore Ts'o. "Kerberos: An Authentication Service for Computer Networks." In: *IEEE Communications*. Vol. 32. 9. IEEE, 1994, pp. 33–38.

[65] C. Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120. http://www.rfc-editor.org/rfc/rfc4120.txt. RFC Editor, July 2005. URL: http://www.rfc-editor.org/rfc/rfc4120.txt.

[66] Chris Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. http://www.rfc-editor.org/rfc/rfc2595.txt. RFC Editor, June 1999. URL: http://www.rfc-editor.org/rfc/rfc2595.txt.

[67] R. Oppliger. "Certification Authorities Under Attack: A Plea for Certificate Legitimation." In: *IEEE Internet Computing* 18.1 (Jan. 2014), pp. 40–47. ISSN: 1089-7801. DOI: 10.1109/MIC.2013.5.

[68] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642041000, 9783642041006.

[69] Lawrence C. Paulson. "Inductive Analysis of the Internet Protocol TLS." In: *ACM Trans. Inf. Syst. Secur.* 2.3 (Aug. 1999), pp. 332–351. ISSN: 1094-9224. DOI: 10.1145/322510.322530. URL: http://doi.acm.org/10.1145/322510.322530.

[70] Rick van Rein. *TLS-KDH: Kerberos + Diffie-Hellman in TLS*. Internet-Draft draft-vanrein-tls-kdh-01. http://www.ietf.org/internet-drafts/draft-vanrein-tls-kdh-01.txt. IETF Secretariat, Dec. 2015. URL: http://www.ietf.org/internet-drafts/draft-vanrein-tls-kdh-01.txt.

[71] Rick van Rein. *TLS-KDH: Kerberos + Diffie-Hellman in TLS*. Internet-Draft draft-vanrein-tls-kdh-04. http://www.ietf.org/internet-drafts/draft-vanrein-tls-kdh-04.txt. IETF Secretariat, June 2016. URL: http://www.ietf.org/internet-drafts/draft-vanrein-tls-kdh-04.txt.

[72] E. Rescorla. *HTTP Over TLS*. RFC 2818. http://www.rfc-editor.org/rfc/rfc2818.txt. RFC Editor, May 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[73] C. Rigney et al. *Remote Authentication Dial In User Service (RADIUS)*. RFC 2865. http://www.rfc-editor.org/rfc/rfc2865.txt. RFC Editor, June 2000. URL: http://www.rfc-editor.org/rfc/rfc2865.txt.

[74] Berry Schoenmakers. *Lecture Notes Cryptographic Protocols*. University Lecture. Version 1.2. Feb. 2016. URL: https://www.win.tue.nl/~berry/2DMI00/LectureNotes.pdf (visited on 07/21/2016).

[75] Claude E. Shannon. "Communication Theory of Secrecy Systems." In: *The Bell System Technical Journal* 28.4 (Oct. 1949), pp. 656–715. ISSN: 0005-8580. URL: http://bstj.bell-labs.com/BSTJ/images/Vol28/bstj28-4-656.pdf;%20http://en.wikipedia.org/wiki/Communication_Theory_of_Secrecy_Systems;%20http://www.cs.ucla.edu/~jkong/research/security/shannon1949.pdf.

[76] Y. Sheffer, R. Holz, and P. Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. RFC 7457. http://www.rfc-editor.org/rfc/rfc7457.txt. RFC Editor, Feb. 2015. URL: http://www.rfc-editor.org/rfc/rfc7457.txt.

[77] D. Simon, B. Aboba, and R. Hurst. *The EAP-TLS Authentication Protocol*. RFC 5216. http://www.rfc-editor.org/rfc/rfc5216.txt. RFC Editor, Mar. 2008. URL: http://www.rfc-editor.org/rfc/rfc5216.txt.

[78] Ben Smyth and Alfredo Pironti. *Truncating TLS Connections to Violate Beliefs in Web Applications*. Research Report. This document extends https://hal.inria.fr/hal-00863371v1. INRIA Paris, Oct. 2014. URL: https://hal.inria.fr/hal-01102013.

[79] William Stallings. *Network Security Essentials: Applications and Standards*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 0137067925, 9780137067923.

[80] D. Taylor et al. *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*. RFC 5054. RFC Editor, Nov. 2007.

[81] Hannes Tschofenig and Manuel Pegourie-Gonnard. *Performance Investigations*. IETF proceeding 92. ARM. Mar. 15, 2015. URL: https://www.ietf.org/proceedings/92/slides/slides-92-lwig-3.pdf (visited on 08/14/2016). Presentation.

[82] Hannes Tschofenig and Manuel Pegourie-Gonnard. *Performance of State-of-the-Art Cryptography on ARM-based Microprocessors*. NIST Lightweight Cryptography Workshop 2015 Session VII: Implementations & Performance. ARM. July 21, 2015. URL: http://csrc.nist.gov/groups/ST/lwc-workshop2015/presentations/session7-vincent.pdf (visited on 08/14/2016). Presentation.

[83] Chundong Wang and Chaoran Feng. "Security Analysis and Improvement for Kerberos Based on Dynamic Password and Diffie-Hellman Algorithm." In: *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies* (2013), pp. 256–260. DOI: http://doi.ieeecomputersociety.org/10.1109/EIDWT.2013.49.

[84] WikiPedia. *Communications protocol — Wikipedia, The Free Encyclopedia*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Communications_protocol&oldid=729634850 (visited on 07/21/2016).

[85] WikiPedia. *Confidentiality — Wikipedia, The Free Encyclopedia*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Confidentiality&oldid=718451771 (visited on 07/21/2016).

[86] WikiPedia. *Cryptographic protocol — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Cryptographic_protocol&oldid=692292315 (visited on 07/21/2016).

[87] Wikipedia. *Active Directory — Wikipedia, The Free Encyclopedia*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Active_Directory&oldid=728918293 (visited on 07/21/2016).

[88] Wikipedia. *Comparison of TLS implementations — Wikipedia, The Free Encyclopedia*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations&oldid=731466926 (visited on 07/28/2016).

[89] S. Winter et al. *Transport Layer Security (TLS) Encryption for RADIUS*. RFC 6614. RFC Editor, May 2012.

[90] P. Wouters et al. *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7250. RFC Editor, June 2014.

[91] Thomas Wu. *A Real-World Analysis of Kerberos Password Security*. NDSS Symposium 1999. 1999.

[92] Yanpas. *Chain of trust ssl certificate*. Image. CC BY-SA 4.0. Jan. 14, 2016. URL: https://commons.wikimedia.org/w/index.php?curid=46369922.

[93] P. Yee. *Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 6818. `http://www.rfc-editor.org/rfc/rfc6818.txt`. RFC Editor, Jan. 2013. URL: `http://www.rfc-editor.org/rfc/rfc6818.txt`.

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

Appendix A

Authenticator contents

Listing A.1 describes the contents of an Authenticator. The lengths of the fields are given below. Since some fields are optional or have a variable length, typical lengths have been chosen to come up with an average Authenticator length for TLS-KDH.

```
Checksum         ::= SEQUENCE {
  cksumtype        [0] Int32,
  checksum         [1] OCTET STRING
}

EncryptionKey    ::= SEQUENCE {
  keytype          [0] Int32 -- actually encryption type --,
  keyvalue         [1] OCTET STRING
}

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
}
```

Listing A.1: ASN.1 specification of a Kerberos Authenticator.

Every field consists of an explicit tag ([·]) and a data type identifier (e.g. `Int32`). A tag consists of a tag ID and a length. This length specifies the number of bytes of the data represented by the data type identifier. An explicit tag is encoded as 1 byte tag ID and 1 byte length. A data type identifier field (e.g. `Int32`) also contains a tag and a length prefix. That means that, for example, an `Int32` field is encoded as: tag (1 byte) $+\!\!+$ length (1 byte) $+\!\!+$ data (1 .. 3 bytes). As an example, the chsumtype field from Listing A.1 is then encoded as: tag ID (1 byte) $+\!\!+$ length (1 byte) $+\!\!+$ tag ID (1 byte) $+\!\!+$ length (1 byte) $+\!\!+$ integer (1 byte). Furthermore, composite structures such as a `SEQUENCE` also have a tag and a length prefix of one 1 byte each. Using this encoding we can calculate the total length of an Authenticator. We use TLS-KDH typical field lengths for this calculation. A full description of the ASN.1 notation and encoding can be found in [44]. We only use here what is relevant for the Authenticator length. The lengths are computed as follows:

Checksum: `cksumtype` (2+2+1) + `checksum` (2+2+20**) + sequence prefix (2) = 31 bytes.

EncryptionKey: `keytype` (2+2+1) + `keyvalue` (2+2+16***) + sequence prefix (2) = 27 bytes.

Authenticator (KDH-enhanced): `authenticator-vno` (2+2+1) + `crealm` (2+2+8*) + `cname` (2+2+13*) + `cksum` (2+31) + `cusec` (2+2+3) + `ctime` (2+2+4) + sequence prefix (2) = 80 bytes.

Authenticator (KDH-only): `authenticator-vno` (2+2+1) + `crealm` (2+2+8*) + `cname` (2+2+13*) + `cksum` (2+31) + `cusec` (2+2+3) + `ctime` (2+2+4) + `subkey` (2+27) + sequence prefix (2) = 109 bytes.

* *These values are estimated and may vary.*
** *We take a SHA1 checksum here as an example.*
*** *We take a 128-bit key here as an example.*

In TLS-KDH, the checksum type is negotiable and its length varies between 20 (SHA1) and 64 bytes (SHA512). Additionally, the subkey length depends on the chosen symmetric cipher and varies between minima of 16 and 32 bytes. The average length of a KDH-enhanced Authenticator thus lies between 80 and 124 bytes. The average minimum length of a KDH-only Authenticator thus lies between 109 and 170 bytes.

Appendix B

TLS PRF definition

The Pseudo Random Function that is defined for TLS is based on a keyed hash function [21]. The PRF is used for secret expansion into arbitrary-length output, for the purposes of key generation and validation. It is defined as follows:

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

Here + denotes concatenation, and A() is defined as:

      A(0) = seed
      A(i) = HMAC_hash(secret, A(i-1))

PRF(secret, label, seed) = P_hash(secret, label + seed)
```

Listing B.1: Pseudo code for PRF definition.

Appendix C

GnuTLS extension definition

Here an example definition for a TLS handshake extension is given.

```
extension_entry_st ext_mod_kdh_trf = {
  .name = "KDH Ticket Request Flags",
  .type = GNUTLS_EXTENSION_KDH_TRF,
  .parse_type = GNUTLS_EXT_APPLICATION,
  .recv_func = _gnutls_kdh_trf_recv_params,
  .send_func = _gnutls_kdh_trf_send_params,
  .pack_func = _gnutls_kdh_trf_pack,
  .unpack_func = _gnutls_kdh_trf_unpack,
  .deinit_func = _gnutls_kdh_trf_deinit
};
```

Listing C.1: Extension definition structure.

.name Contains the human readable name of the extension.

.type Contains a system identifier for the extension. This is often a constant defined in the main GnuTLS library header file `gnutls.h`.

.parse_type Contains a parse type that dictates the order in which extensions will be processed. Some extensions need their data to be available at the TLS level while others only need to be available at application level. This might require a different processing order.

.recv_func Contains a pointer to a function that is executed when extension data needs to be decoded en written to internal memory just after being received over the wire.

.send_func Contains a pointer to a function that is executed when extension data needs to be encoded en written to the output buffer prior to being transmitted over the wire.

.pack_func Contains a pointer to a function that will be executed when extension data needs to be encoded in order to be stored for session resumption.

.unpack_func Contains a pointer to a function that will be executed when extension data needs to be decoded in order to be restored during session resumption.

.deinit_func Contains a pointer to the deinit function that is used to de-initialize private extension data.

Index