TA4NGI – Evaluation & Design

Version 1.0

Milestone 2

Author:

David Hübner

Co-Authors:

Peter Gietz

Martin Haase

Ali Haider

File name: Evaluation&Design_v1.0.odt

Created on: 15.04.2021

Last change: 15.04.2021

Version control:

Version	Date	Author	Changes
1.0	15.04.21	David Hübner Peter Gietz Martin Haase Ali Haider	Version 1.0 including technology evaluation and implementation concept for PoC 1,2 & 3

Final acceptance:

Version	Date	Accepted by	Role / position
1.00			

Table of Contents

1 Introduction....1
1.1 Purpose Of This Document....1
1.2 Introduction And Problem Statement....1
1.3 NGI-Pointer....2
1.4 TA4NGI....2
2 Technology....3
2.1 Related Technology....3
2.1.1 Kerberos....3
2.1.2 TLS....4
2.1.3 Diffie-Hellman....6
2.2 TLS-KDH....6
2.2.1 Motivation....6
2.2.2 KRB5-KDH....7
2.2.3 TLS-KDH....8
2.2.4 Related Work....9
2.3 Satosa....10
2.3.1 Architecture....10
2.4 Corteza & Crust....11
3 Implementation Concept for the PoC....12
3.1 Satosa Module for Authentication with TLS....12
3.1.1 Goal Description....12
3.1.2 Concept....12
3.1.3 KPI....15
3.2 Satosa Module for Authentication with Kerberos....15
3.2.1 Goal Description....15
3.2.2 Concept....16
3.2.3 KPI....19
3.3 Satosa Module For Authentication With TLS-KDH....19
3.3.1 Goal Description....19
3.3.2 Concept....20
3.3.2.1 Goal 1 – Client-Server Hello World Application....20
3.3.2.2 Goal 2 – TLS-KDH Proxy....21
3.3.3 KPI....24
3.4 Evaluation of TLS-KDH for Corteza....24
4 Summary....24

1 Introduction

1.1 Purpose Of This Document

The purpose of this document is to give an introduction to the TA4NGI project funded by NGI-Pointer, its underlying technology and the scope and goals of the technical proof of concepts deliverered by the end of the project.

It addresses the project team at DAASI International who is going to work on the project, stakeholders in the NGI-Pointer project and interested third parties, that want to follow the progress of TA4NGI. While this document contains some technical details of the underlying protocols and planned implementations, the goal is not to provide a comprehensive technical documentation but rather a high level overview.

1.2 Introduction And Problem Statement

TA4NGI aims to evaluate future internet technology for authentication purposes. In that capacity it aims to integrate new authentication mechanisms in established single sign-on (SSO) protocols and existing and widely adopted open source software. One of these promising new specification is TLS-KDH, which combines the strengths of established protocols, namely Kerberos, Diffie-Hellmann (DH), and TLS to create a future proof authentication and encryption mechanism. Future proof particularly refers to sufficient capacities for encryption scheme secure enough to even withstand the challenges presented by much higher computational capacities as represented in the perspective of quantum computing. Furthermore, the specification also offers perfect forward secrecy, i. e. it mitigates risks to the confidentiality of the encrypted data even if the security of the encryption keys are compromised in the future.

Today, SSO in web scenarios is predominantly achieved relying on open standards, such as OpenID Connect (OIDC) and SAML 2.0. The open source software Satosa offers support for these protocols based on various Python libraries and is widely used especially in research and education. Its modular approach allows to connect a wide variety of applications and to support different authentication mechanisms. While these authentication mechanisms are by default mostly SSO protocols, again (turning Satosa into a SSO proxy solution), Satosa is not limited to those and can also authenticate based on e.g. TLS, Kerberos or – which will be the main focus of our work – TLS-KDH.

With TA4NGI we plan to implement proof of concepts for Satosa authentication modules for the following technologies:

1. TLS client certificates
2. Kerberos tickets
3. TLS-KDH

The focus of these modules will be twofold. For the established protocols the solution should support major user centric scenarios of today. For example, TLS client certifi-

cates authentication must support authentication via OpenID Connect by using certificates installed in modern web browsers. For the new specification, TLS-KDH, the solution should evaluate feasible integration scenarios with given constraints in mind. For example, it is unlikely that web browsers support TLS-KDH in the foreseeable future. Therefore the proof of concept in this regard will be limited to server-to-server scenarios.

Beyond just authentication, TLS-KDH can also be used for more secure transport layer encryption. This will be evaluated further on a theoretical level for usage in the open source collaboration and low-code development platform Corteza, e.g. for messaging or CMS use cases.

1.3 NGI-Pointer

NGI Pointer is an initiative of the European Commission aiming at „Funding The Next Generation Ecosystem of Internet Architects“[1]. Basically an attempt to create new technologies within the European Union, while defining architects as „people with an ambition of changing the Internet and Web with European Values at the core”[2]. NGI Pointer is one of several such funding schemes under the umbrella of NGI (Next Generation Internet), which aims at „an Internet that responds to people’s fundamental needs, including trust, security, and inclusion, while reflecting the values and the norms all citizens cherish in Europe.“[3] NGI had an initial dedicated funding and is now part of the Horizon Europe Programme (2021-2027). NGI Pointer is a rather focused activity that provides funding for various small projects with a total spending of 5.6 Million, intended to build practical applications of state-of-the-art technologies. This was a perfect fit for our ideas on creating a proof of concept for a very ambitious new technology based on a combination of well established protocols.

1.4 TA4NGI

TA4NGI stands for Trust and Authentication for Next Generation Internet; here trust, among other things, stands for authenticity and safety from interception; and authentication stands for securely proving the identity of a user and their rightful ownership of respective credentials. TA4NGI wants to find innovative solutions based on already existing technologies as an attempt at a rather pragmatic approach.

This kind of approach can be seen in several IETF Drafts of Rick van Rein, especially „Quantum Relief with TLS and Kerberos“[4] connected to the Arpa2 Project[5]. Here such a higher level of security in TLS encrypted communication is aimed at, that it will still hold true in the age of quantum computing, which is not so far away from today. In this draft a solution is proposed that complements the X.509 based encryption by adding Kerberos, as it builds a symmetric-key infrastructure including cross-realm connectivity options and also integrating (Elliptic-Curve) Diffie-Hellman for perfect forward secrecy. This approach is called KDH (Kerberised Diffie-Hellman) and added to the transport security of TLS, thus

1 See https://pointer.ngi.eu/
2 dito
3 See https://www.ngi.eu/about/
4 Current Version see https://tools.ietf.org/html/draft-vanrein-tls-kdh-06
5 See http://arpa2.net/

the name TLS-KDH.

TA4NGI implements TLS-KDH as a proof of concept into an open source SSO-Proxy called Satosa[6]. Satosa has a very structured architecture which allows for creating so-called microservices as plugins. TA4NGI will implement such microservices for:

- Backend module to support authentication with TLS client authentication
- Backend module to support authentication with Kerberos
- Implementation of TLS-KDH in Satosa based on the two backends

The fourth deliverable will be an evaluation of transport layer security with TLS-KDH in Corteza, a new Open Source business application framework with a CRM module and a low-code development platform; this way introducing an actual use case for this new technology.

2 Technology

2.1 Related Technology

2.1.1 Kerberos

Kerberos, as an authentication/ single sign-on protocol, dates back to the 80s and was developed at the MIT, version 5 is latest one. Kerberos5 uses two separate components besides the client and the target service:

- An authentication service (AS) which allows for single sign-on, granting so-called ticket-granting tickets (TGT), that are short-lived (approximate lifespan of one day)
- A ticket-granting service (TGS) which will similarly create short-lived session keys for the communication of the client to the target host.

The following table visualises the steps the Kerberos protocol is composed of:

Step	AS/TGS	Client	Target Service
AS-REQ	Authenticate e.g. with password hash to AS		
AS-REP	AS sends client-key-encrypted TGT to Client, plus TGT-encrypted part for TGS		
TGS-REQ	Client sends TGT-encrypted part to TGS, including target service ID		
TGS-REP	TGS sends session key encrypted by both the TGT, and the target service's key		
AP-REQ		Client sends target-service-encrypted session key to target service, plus a session-key-encrypted authenticator that conveys the client's ID	
AP-REP		Service replies with session-key-encrypted authenticator that conveys the services' ID	

6 See https://github.com/IdentityPython/SATOSA

Note that the AS-REQ/REP is usually done once per day, and TGS-REQ/REP and AP-REQ/REP for each service accessed. The result of all steps is a symmetric short-lived session key which can be used for the encryption of data between client and service.

Kerberos uses symmetric cryptography only. The following keys are used and known by the key distribution center (KDC) which is usually bundled with the AS and TGS:

- Long-lived key for user
- Long-lived key for target service
- Short-lived key for SSO: the TGT
- Short-lived session key for data encryption

The symmetric key ciphers currently regarded as secure, and thus recommended, are AES-256 and Camellia-256, even though currently also 128bit ciphers are considered to be sufficient in this case, as the keys are are only temporary.

Kerberos does not allow for perfect forward secrecy, although knowledge of the long-lived keys (for user, or for host) will not allow decryption of past data traffic, since all messages are encrypted by short-lived session keys. However, this does not hold true for the initial session key exchange messages: the session key that is used for communication between client and target host can be uncovered in several circumstances (see table above for the steps):

- A) By uncovering the client’s long-lived key
    - 1) If the attacker uncovered the client’s long-lived key, and recorded the AS-REP, he can decrypt the TGT
    - 2) If the attacker uncovered the TGT, and recorded the TGS-REP, he could uncover the session key
- B) If the attacker uncovered the service’s long-lived key, and recorded the AP-REQ, he could extract the session key

With the session key compromised by either A) or B), all further communication between the client and the service can be uncovered. Of course a rogue KDC admin is also able to intercept such communication rendering it insecure. Thus, PFS is not part of standard Kerberos5, although DH is supported in Pkinit.

2.1.2 TLS

TLS is an acronym for Transport Layer Security. TLS is a protocol situated above the transport layer in the OSI network model. TLS enables two parties to authenticate themselves, and communicate with each other while maintaining privacy and data integrity. The party which initiates the communication becomes the client and the party to which the communication request is sent becomes the server. TLS can be used to enable secure communication for any higher level application protocol, for example, HTTP, FTP, SMTP, etc. TLS enhances the HTTP (Hypertext Transfer Protocol) to become HTTPS where the added ‘S’ stands for security.

TLS enables the authentication of the client and server by exchanging X509 certificates. An X509 certificate includes a public key of the holder of the certificate and is digitally signed by a certification authority (CA). The CA is an agreed-upon entity which issues digital certificates. Each, client and server are in possession of a list of all trusted CAs so they can verify the authenticity of a digital certificate. Usually on the open-web, only server authentication is required, which only requires a server to transmit its certificate to a client. This is due to it not being scalable nor convenient for all the clients, mostly browsers, to configure client certificates. However, in these cases where security and authentication are paramount, client authentication, more commonly referred to as mutual TLS authentication, can be used to enforce authentication on both, the server and the client.

TLS uses the symmetric encryption technique to encrypt and decrypt data on the communication channel. This stands in contrast to the asymmetric encryption/decryption technique where data is encrypted or decrypted with two distinct keys. Whereas asymmetric encryption/decryption uses a private key and a public key, the symmetric encryption/decryption uses one key which must be private. Due to the enormous computational capacities necessary for asymmetric algorithms, using the symmetric encryption technique in TLS instead considerably increases the performance. Nonetheless, the same private key needs to be exchanged securely between the client and the server before initiating secure communication. The initiation of every communication session begins with the TLS handshake in which a shared secret session is exchanged by means of the asymmetric encryption technique

The following section briefly lists the steps of the TLS handshake:

1. The client sends a “client hello” message which includes the cipher-suits the client can support. A cipher-suite includes the following:
   - TLS protocol version, for example, TLS 1.2, 1.3.
   - Key exchange method, for example, DH, RSA, etc.
   - Authentication scheme, for example, RSA, ECDSA, etc.
   - Cipher: the symmetric cipher which uses the agreed shared session key to encrypt/decrypt the network data, for example, AES, GCM/CBC, etc.
   - MAC (Message Authentication Code): this hashing algorithm is used by the client and server for authentication and data integrity purposes, for example, SHA, MD5, etc.
2. The server selects a cipher-suit and sends a “server hello” message back to the client. This also includes the server certificate. If the server also requires the client to authenticate themselves, the message also includes “client certificate request” which includes a list of supported certificate types and the distinguished names of acceptable CAs.
3. The client checks the validity of this certificate.
4. The client generates a random byte string, encrypts it with the server public key

within the server certificate, and sends it back to the server. The random string, which is also known as the pre-master secret, is used by the client and server to generate the master secret, or shared session key. If the server included the “client certificate request” with the original message, the client encrypts the random string with the client's private key instead. It is then sent back together with the client's digital certificate, or a no digital certificate warning; the handshake would then break if the client authentication is mandatory.

5. The server verifies the client certificate if “client certificate request” was included.
6. The client sends a “finished” message to the server, which is encrypted with the shared secret session key, indicating that the client’s part in the handshake is complete.
7. The server sends a “finished” message to the client, which is encrypted with the shared secret session key, indicating that the server also completed its part in the handshake.
8. For the duration of the TLS session, the server and client can now exchange messages which are symmetrically encrypted with the shared secret key.

2.1.3 Diffie-Hellman

Diffie-Hellman is a key exchange method. The essence of the Diffie-Hellman key exchange method is that the client and the server need to exchange a shared secret key for symmetric encryption afterward. The mathematics involved in the Diffie-Hellman algorithm enable perfect forward secrecy. Perfect forward secrecy is a feature of key exchange methods which means that even if the private key of secure communication is leaked in the future, the past session data encrypted with this private key would not be compromised.

In the TLS handshake flow listed above, if an attacker comes into possession of the server's private key, the attacker can then use the key to decrypt the pre-master secret generated by the client in step 4 of the TLS handshake. And then further, can generate the shared session key from this pre-master secret and can eventually decrypt all the session data. Diffie-Hellman avoids this concern entirely by using some mathematical properties involving a prime number and a module which can be exposed publicly. The key exchange algorithm does not rely on either the public key nor the private key of the client/server to exchange the pre-master secret, and as a result enables perfect forward secrecy.

2.2 TLS-KDH

2.2.1 Motivation

TLS-KDH is an evolving standard protocol for high security authentication and transport encryption, which aims at security in a future world of quantum computing. It combines three established technologies to achieve this:

- Kerberos, which is an established and secure authentication protocol

- TLS, the standardised form of secure socket layers, which provides encryption of the body load sent via an Internet protocol like HTTP and LDAP, and is based on X509 asymmetric encryption. It is also used for bidirectional authentication between server and client, and thus a method to provide secure authentication
- DH, Diffie-Hellman key agreement protocol for key establishment featuring perfect forward secrecy in conjunction with a key-secured hash message authentication code for achieving mutual authentication and message integrity of the key management messages exchanged

Since DH and TLS can be used together since TLS v 1.2, as recommended in RFC 7525, the real innovation of TLD-KDH is the integration of the Kerberos protocol in order for a client to be able to use a Kerberos ticket for authentication instead of an X509 client certificate. This efficiently facilitates using highly secure technology, because the user does not have to manage their own x509 private key and certificate. Furthermore, using Kerberos tickets over X509 certificates can be faster and more elegant, since it does not require validating long CA chains. Also on a fundamental level, using short-lived Kerberos tickets poses various advantages as opposed to usually long-lived certificates. For example, the period in which a compromised ticket can be used, is much smaller.

The introduction of Kerberos as the authentication mechanism in TLS may also allow to use synergies with established Kerberos infrastructures. Sometimes an existing Kerberos infrastructure might be easier to use than setting up a new CA.

A parallel specification, KRB5-KDH, introduces strong encryption with DH into Kerberos and builds the foundation for TLS-KDH.

Ultimately, using TLS-KDH enhances the security of authentication and encryption achieved through these protocols by leveraging the strong authentication mechanisms of Kerberos and key exchange with DH to enhance these properties in the standard TLS communication.

2.2.2 KRB5-KDH

KRB5-KDH is a specification which combines Kerberos as an authentication mechanism and DH as the key exchange mechanism for encryption within Kerberos. While this specification can be used on its own, it has strong ties with TLS-KDH and eventually will be used as one of the components of the latter.

Kerberos offers established and strong authentication. Tickets used to transmit the authentication in Kerberos are usually encrypted. These tickets can contain sensitive information, as they might at the very least contain the identity of the authenticated user. The encryption properties can be enhanced by using DH. Thus, DH compliments the Kerberos authentication with its stronger encryption properties and enables perfect forward secrecy.

Using KRB5-KDH, the Kerberos protocol is extended to support the DH key exchange method to agree on the shared secret for ticket encryption. For that purpose, the following changes are made to Kerberos [7]:

7 http://tls-kdh.arpa2.net/krb5-kdh.html

- A new encryption type for a Diffie-Hellman key exchange message;
- A new Kerberos5 ticket flag to indicate support for the Diffie-Hellman encryption type.

Integration on that level leads to minimal changes in the actual Kerberos protocol flow. This allows to continuously use mechanisms such as GSSAPI and also minimise the changes required to the existing Kerberos implementations.

The abstract KRB5-KDH protocol flow is demonstrated in [8]. This illustrates the usage of DH key exchange to acquire a secure shared secret with the perfect forward secrecy property:

KDH Client	**KDH Server**
Obtain ticket for server	
Construct local DH key	
Client -> server: AP_REQ with DH Key exchange in the subkey field	
	Construct local DH key
Client <- server: AP_REP with DH Key Exchange in the subkey field	
Compute DH Shared Secret	Compute DH Shared Secret
Exchange wrapped data, encrypted with the DH shared secret	

2.2.3 TLS-KDH

While KRB5-KDH focuses on the combination of Kerberos and DH to achieve a more secure authentication through Kerberos (by enhancing its encryption), TLS-KDH integrates these properties into the TLS protocol. There are a variety of major advantages in doing so:

- DH is used for encryption key exchange in TLS, enabling stronger encryption and perfect forward secrecy
- Authentication (both server and client) in the TLS handshake can be done based on Kerberos tickets, enabling both usability (Kerberos tickets might be preferred over X509 certificates) and security (better encryption of identity information) advantages

Resistance against quantum computer attacks and perfect forward secrecy in such scenarios is achieved through a combination of DH key exchange and the added entropy generated during the TLS handshake. For this purpose a quantum_relief TLS extension is introduced.

These advantages are achieved with minimal changes to TLS, while maintaining the general protocol flow, which should allow integration of TLS-KDH into existing TLS implemen-

8 http://tls-kdh.arpa2.net/conceptual.html

tations with manageable effort. In particular, the following changes are made to TLS [9]:

- New cipher suites TLS_DHE_KRB5_*and TLS_ECDHE_KRB5_*;
- A new TLS extension for realm names;
- New cases for ServerKeyExchange and ClientKeyExchange types;
- Incorporating a new calculation method for Diffie-Hellman shared secrets.

The additions to Kerberos to support KRB5-KDH explained above are also required in the context of TLS-KDH.

Using TLS with the TLS-KDH extensions allows for two different protocol flows, the client-to-server and peer-to-peer flows. For the purpose of our work only the client-to-server variant will be considered further. This is in line with today's scenarios for TLS for authentication use cases. Here, TLS-KDH offers three features [10]:

- (1) additional secret entropy for encryption
- (2) client authentication through Kerberos Tickets and
- (3) server authentication through Kerberos Tickets

The first property enhances the encryption strength of TLS and does not immediately affect the authentication properties, but provides quantum relief for TLS encrypted data. During the TLS handshake Kerberos tickets are supplied by both the server and the client [11]. Combined, they can be used to securely derive a shared secret, which is then distributed using the normal TLS key schedule. This is achieved by using a new quantum_relief extension in the TLS protocol [12].

The second property is the main focus of our work, since it allows to use the enhanced authentication scenarios in combination with the TLS protocol flow.

The third property is not within scope of our proof of concept implementations.

2.2.4 Related Work

The idea to use Kerberos as authentication for TLS is not new. Simo Sorce (Redhat) proposed a similar idea in RFC2712 [13], which can be considered as a predecessor of the TLS-KDH work. The main drawback of this RFC is the lack of perfect forward secrecy. It only covers authentication but not more secure encryption.

There also is a proposal from Josh Howlett (Janet) to integrate GSSAPI into TLS communication [14], which has been rejected by the respective IETF working group. The changes to the TLS protocol flow are more severe and it also does not directly touch the integration of DH.

---

9 http://tls-kdh.arpa2.net/tls-kdh.html
10 https://tools.ietf.org/html/draft-vanrein-tls-kdh-06 Chapter 3.2
11 https://tools.ietf.org/html/draft-vanrein-tls-kdh-06 Chapter 2
12 https://tools.ietf.org/html/draft-vanrein-tls-kdh-06 Chapter 4.1
13 http://tools.ietf.org/html/rfc2712
14 http://tools.ietf.org/html/draft-williams-tls-app-sasl-opt

A related topic is Realm Crossover [15] which aims to allow using decentral identity providers, very much like federation use cases in the web SSO protocol SAML2. In Kerberos, KXOVER [16] can be used to allow secure exchange of key material between different Kerberos domains, so that clients from one domain can access services in the other domain. The project still is a work-in-progress and could be combined with i.e. TLS-KDH to further improve the security. While we do not explicitly use Realm Crossover in our work, this might be an interesting addition in the future.

2.3 Satosa

Satosa is an authentication and authorisation proxy – It translates one authorisation protocol into another. At the same time, the modular architecture of Satosa also enables the augmentation and customisation of request and response data.

Satosa is often used to enable communication between a service and an identity provider. For example, an educational institution may run a service which initially only supports SAML IdPs for the authentication and authorisation of users. However, in the future a new protocol may be introduced into the market and will be adapted by many new IdPs, i.e. the introduction of OAuth2, Social IDs for authentication, etc. In this case, Satosa can help to translate the SAML-based authorisation to OAuth2.0 or Social-ID-based authorisation.

2.3.1 Architecture

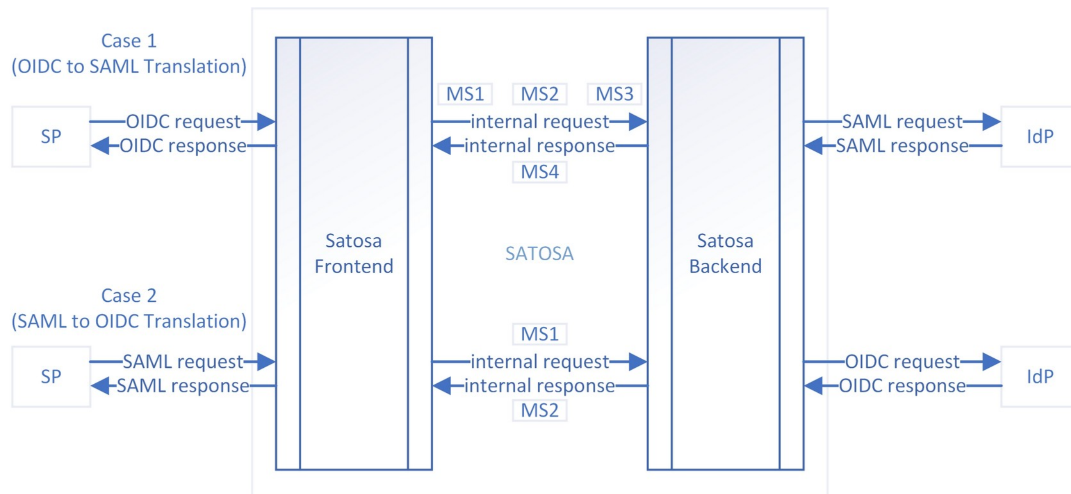
Satosa has a modular structure and consists of the following three layers:

- Frontend: the frontend component receives an authorisation request from a service provider (SP). The component then converts the authorisation protocol-specific request to its SATOSA's Internal authorisation request format. Once a request is converted to the internal authorization request, a series of (request) microservices (MS) can be called upon the internal request. The order of the calling sequence is controlled by Satosa proxy configuration files. SATOSA is shipped with some built-in frontend components: OIDC, SAML.
- Microservices (MS): microservices are located between the frontend and backend components of SATOSA and comes with a variety of functions. While one microservice may contact LDAP or an external database to fetch some additional attributes of a user, another may redirect a user to a consent screen to ask permission to release some of her attributes.
- Backend: the backend component usually connects to an IdP where a user can authenticate themselves. SATOSA includes a couple of backend component implementations out-of-the box: SAML and OIDC, as well as specific backends for social IdPs. Nonetheless, SATOSA is not restricted to only connect to an IdP as exemplified in our POC, here it is only necessary to check for the presence of an environment variable. Based on the value of this variable, the authorisation response is formulated and the flow is completed. The backend is responsible for the validation

15 http://realm-xover.arpa2.net/
16 https://k5wiki.kerberos.org/wiki/Projects/Realm_Crossover_between_KDCs

and finalisation of the authorisation protocol-specific response as well as its conversion into SATOSA's internal authorisation response format. After the response is converted to the internal authorisation response, then again, a series of (response) microservices can be called upon the internal response, and finally, the internal response reaches the frontend component where it is converted to the authorisation protocol-specific response.

The discussed architecture is illustrated below



While Satosa has a lot of functions making it a so called SP-IDP-proxy (i.e. it can translate between different frontend SSO protocols and other backend SSO protocols), it of course also provides the framework for other SSO use cases. For example, the backend can just do authentication with username and password against an LDAP server. We use this modularity to implement different backends for various authentication methods.

2.4 Corteza & Crust

Corteza[17] is a low-code development environment which makes building responsive web applications, i.e. for customer relationship management, a lot easier. It has been developed by the open source company Crust Ltd. The vision for Corteza is to build a community to sustainably provide a digital work platform that is designed from the ground up to help create a better world. It is thus very much in alignment of the overall goals of NGI and by extension NGI Pointer, and therefore a good example for evaluating, how new avangarde security technologies such as TLS-KDH can be applied to increase it's security.

Corteza includes a mode called Corteza Federation, which enables different Corteza instances to establish a federated network to exchange information. Since Corteza is used for different kinds of business processes, some of which entail processing very sensitive

17 See https://cortezaproject.org/

data, this kind of federated communication needs to be as secure as possible, even in the age of quantum computing.

For the purposes of Corteza, TA4NGI will reflect on the practical usability of TLS-KDH in the form of a text deliverable, including an evaluation of how the security can be enhanced as well as the necessary work to do so.

3 Implementation Concept for the PoC

3.1 Satosa Module for Authentication with TLS

3.1.1 Goal Description

This task is about the development of a backend module for Satosa which can authenticate using TLS client certificates.

As a first step, X509 client certificates can be used, these are sent from a web browser (installed as client certificates in the browser) and integrated in web-brower-based authentication flows (e.g. OIDC authorization code flow).

Additionally, support for requests from any other TLS client application (e. g. some server side implementation) should be supported. In these cases non-web-browser-based authentication flows must be supported (e.g. OAuth2 client credentials flow).

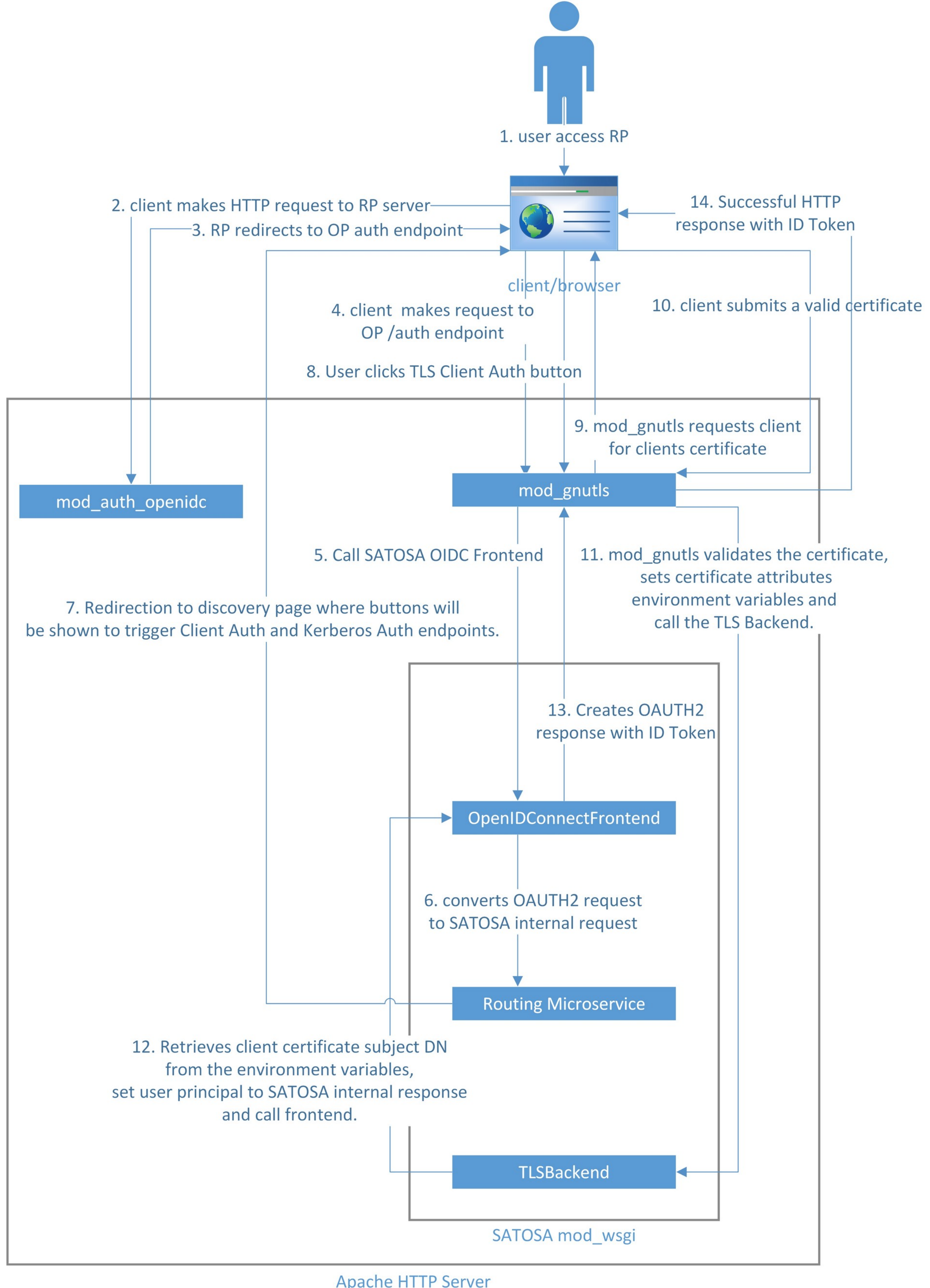
The solution will be the basis for implementation of TLS-KDH, meaning the underlying technology should be chosen with this goal in mind. Changes required to eventually support TLS-KDH should be minimal (e.g. using GluTLS instead of OpenSSL etc.).

3.1.2 Concept

As the main focus of our TLS-KDH PoC is client authentication through Kerberos Tickets, which is equivalent to client authentication through X509 certificates in the TLS protocol. Therefore the PoC would begin with the implementation of a TLSBackend in Satosa which would assert a successful TLS client authentication flow. This backend has no direct technical link to TLS-KDH but can support that ultimate goal by e.g. bootstrapping the initial development environment setup. It can also help in practically realizing the final TLS-KDH client authentication flow which is the end goal, and may also provide grounds for the quantitative performance analysis between TLS and TLS-KDH.

Just like TLS, Kerberos, or TLS-KDH is application authenticaion/ authorisation protocol agnostic; the implementation of the new Satosa TLS backend and then later the Kerberos backend as part of this PoC is independent of authentication/ authorisation protocol used in the frontend. The OIDC frontend has merely been selected as an example that how a high level application authentication/ authorization protocol can be used with a lower level transport layer authenitcation protocol.

The idea here is to use the OIDC authorisation flow to assert a successful TLS client authentication flow. The overall flow as explained below and illustrated in the diagram:



1. A user accesses RP (Relying party). An Apache module, `mod_auth_openidc` would be used, which would serve as RP. `mod_auth_openidc` is an authentication/authorisation module for the Apache 2 HTTP server that authenticates users against an OP (OpenID Connect Provider).
2. RP redirects the user to the OP which is based on Satosa to fulfil the authorisation requests. An example of such an authorisation request URL may look like this:

https://op/authorization?
response_type=id_token&scope=openid&client_id=rp&state=f8KCuG ipIYq4DLLhYeB_pfMRF9s&redirect_uri=https%3A%2F%2Frp %2Fprotected%2Fredirect_uri&nonce=SeLf2JdPuy6I_J16AafCBZUVmb-vz81JXs2vlcdQ3daA

This means, `response_type` and value `id_token` are used to issue an ID token from the authorisation endpoint and hence no token endpoint would be used.

3. The Satosa OIDC frontend component would process the request, then convert the OIDC request into an internal request, and then the request would be intercepted by a new request microservice: discovery router microservice.
4. The discovery router microservice redirects the client to a discovery page: this page contains two buttons, the first one for TLS authentication, the second for Kerberos authentication. Each button will initiate the corresponding authentication mechanism by sending out the appropriate HTTP call, the endpoints of which are registered in the Satosa backends, namely the TLS backend and the Kerberos backend respectively.
5. The user clicks on the TLS authentication button which makes the following HTTPS call:

   https://op/tls/authorization

   In the Apache `mod_gnutls` module, the above URL is configured to be able to perform mutual TLS authentication. This means the request is already intercepted by the Apache mod_gnutls module before it even reaches Satosa. The Apache mod_gnutls module would also be in charge of TLS client certificate authentication.
6. `mod_gnutls` will be configured to perform mutual TLS authentication/client authentication via client certificates. This means that `mod_gnutls` will ask the client (RP) for a client certificate. If the RP runs in a web browser, the client certificate must also be configured in the browser. A successful mutual TLS authentication indicates that the browser has presented a valid/unexpired certificate and possesses the private key of the certificate. Moreover, the submitted certificate must be issued from a valid/recognised CA. This CA will be configured in `mod_gnutls` via its httpd `GnuTLSClientCAFile` directive, which would assert the authenticity of the incoming certificate from RP. `GnuTLSClientVerify` with the value `require` will be added only to the protected endpoint of OP: `/tls/authorization` which would ask for a valid client certificate from the client. Any requests without a valid client certificate will be denied. The `SSL_CLIENT_VERIFY` environment variable will only be set to the value: `SUCCESS`.
7. After a successful mutual TLS authentication, mod_gnutls would extract the attributes of the client X509 certificate from the HTTP request and would dump them in a set of the environment variables. With `GnuTLSExportCertificates` configu-

ration of `mod_gnutls` enabled, mod_gnutls exports the same environment variables to the CGI process as `mod_ssl`. After this, `mod_gnutls` will hand over the authorisation request to Satosa.

8. TLSBackend would cater the request as the /tls/authorization URL is registered in this component.
9. The environment variable set by `mod_gnutls` which will hold the value of the certificate attribute containing the client certificate's subject DN will be read in TLS-Backend. This would be considered as the user principal and will be set to Satosa's Internal Response data structure which will then be used by the OpenIDConnect frontend to create the ID token as part of a successful OIDC authorisation response.
10. If the TLS handshake is not successfully completed, for instance due to an invalid or missing client certificate, or else, the OIDC authorisation request will not even reach OP (SATOSA), and will result in an authentication failure for the client (RP).

3.1.3 KPI

To evaluate the successful implementation of this module we plan to test the following:

1. Login with the microservice must be possible with TLS client certificates stored in web browsers. The implementation must be able to successfully authenticate a user, who presents a valid (i.e. issues by a trusted CA) client certificate, and extract the user identifier. This should be recognisable both, in the Apache web server logs and the Satosa application logs.
2. A non-valid client certificate (i.e. one that is not issues by a trusted CA) must be blocked from authentication.
3. For testing purposes, the authentication module must be compatible with Satosa's OIDC frontend. Using the Implicit Flow, the extracted user identifier from the certificate must be transmitted to the client if the authentication was successful.

3.2 Satosa Module for Authentication with Kerberos

3.2.1 Goal Description

This task is about the development of a backend module for the authentication with Kerberos tickets in Satosa.

This module must support the presentation of a valid Kerberos ticket via GSSAPI/SPNEGO in modern web browsers using web-browser-based authentication flows (e.g. OIDC authorization code flow).

Additionally, support of ticket presentation by other means (e.g. SASL) might be necessary, if they are required for the full POC implementation for TLS-KDH. In these flows a

non-web-browser-based authentication flow must be used instead (e.g. OAuth2 client credentials flow). If this is in fact necessary will be evaluated in the course of the POC.

3.2.2 Concept

This part of the concept will set up a Kerberos5 infrastructure and Kerberos5 authentication module for SATOSA. The infrastructure to be set up will be similar to the final TLS-KDH set-up, meaning that as many parts as possible can be re-used.

The components needed are similar to the TLS PoC, with an obvious difference being the Kerberos5 server (KDC) and the Kerberos SATOSA backend:

- Relying Party secured with mod_auth_openidc (same as in TLS PoC)
- Web Browser (without client certificate)
- OIDC SATOSA frontend (same as in TLS PoC)
- Kerberos5 SATOSA backend understanding SPNEGO (to be developed)
- Kerberos5 Server (i.e. KDC with AS and TGS)

The SATOSA Backend to be developed will consist of the following parts:

- SATOSA within an Apache web server using mod_wsgi
- A new backend endpoint protected by mod_auth_gssapi. It is important that this module is compiled against GnuTLS to allow for a seamless transition to TLS-KDH
- Glue Code that extracts the Kerberos User Principal from the Apache Environment provided by mod_auth_gssapi
- Additional configuration parameters to manage the predefined rules to generate a Kerberos user principal (USER123@REALM) in an account in SATOSA (user123). This might be possible, for instance, using regular expression substitution

The following steps are necessary for the flow:

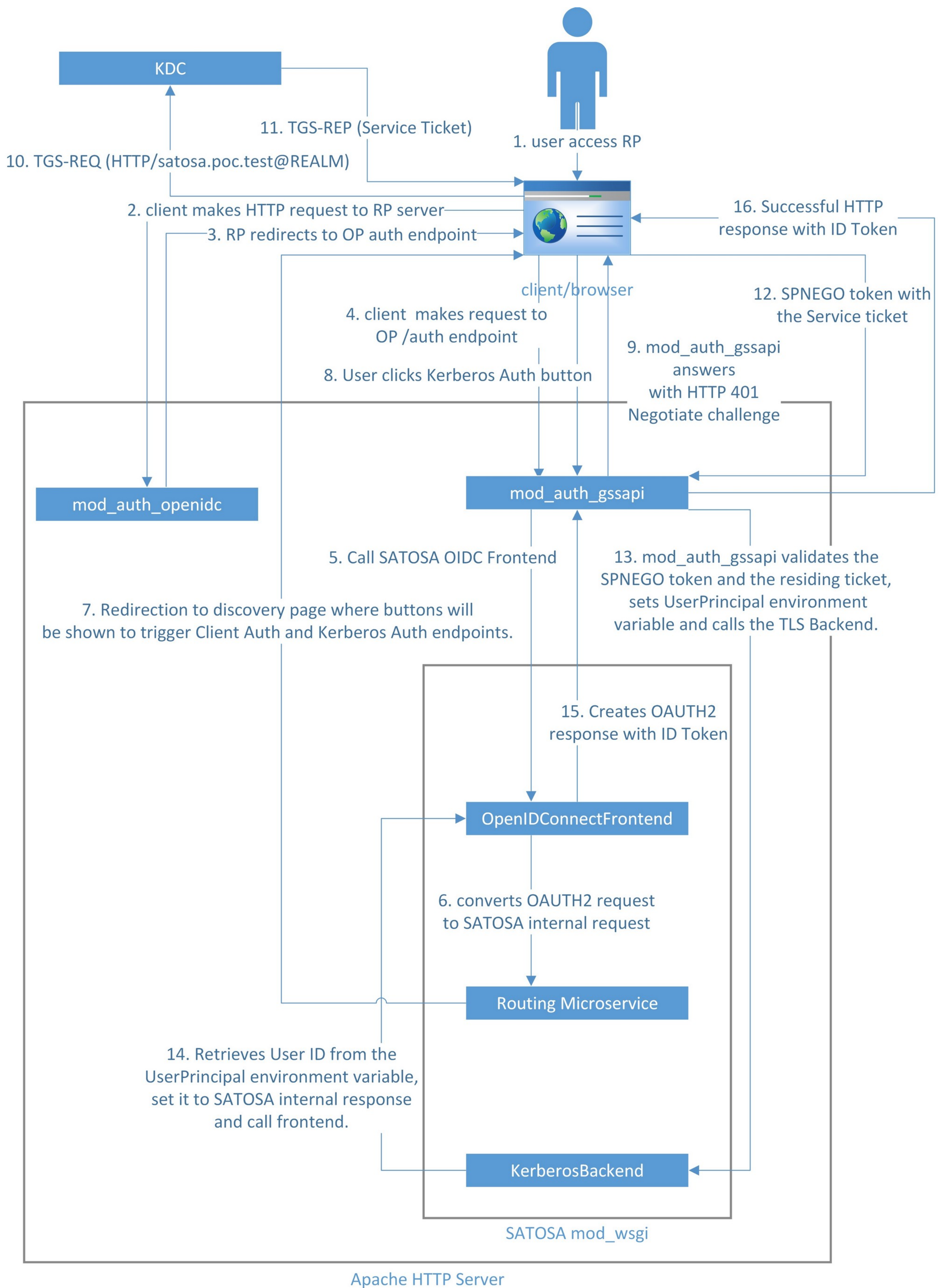
0. User authenticates on the client against the KDC and has receives their daily TGT which is stored in their PC's Kerberos5 credential cache
1. User accesses RP (as in TLS PoC)
2. RP redirects to the SATOSA OIDC Frontend (as in TLS PoC)
3. SATOSA selects the new Kerberos backend via the request microservice for routing in which the user can click on a button “Authenticate with Kerberos”. Clicking this button redirects to the Kerberos backend which is protected by mod_auth_gssapi and initiates the following authentication flow:
    1. mod_auth_gssapi will answer the client browser with an HTTP 401 challenge header that contains the authenticate: negotiate status.
    2. The client browser is configured to support SPNEGO and uses the SATOSA server hostname (satosa.poc.test) to request a service ticket for HTTP/satosa.poc.test@REALM, i.e. the browser issues an TGS-REQ for the KDC.
    3. The KDC replies to the client browser with a TGS-REP with a service ticket, including a session key encrypted using SATOSA's long-lived key.

4. The client browser sends the service ticket to mod_auth_gssapi. This SPNEGO token includes the user's identity.

5. SATOSA will pick up the UserPrincipal set by mod_auth_gssapi in the Apache environment and applies its rules to generate a User ID.

4. The generated User ID will be sent to the OIDC frontend
5. The OIDC frontend replies to the RP (as described in the TLS PoC)

This flow is also illustrated the the diagram below:


KDC
11. TGS-REP (Service Ticket)
1. user access RP
10. TGS-REQ (HTTP/satosa.poc.test@REALM)
2. client makes HTTP request to RP server
3. RP redirects to OP auth endpoint
16. Successful HTTP response with ID Token
client/browser
4. client makes request to OP /auth endpoint
12. SPNEGO token with the Service ticket
9. mod_auth_gssapi answers with HTTP 401 Negotiate challenge
8. User clicks Kerberos Auth button
mod_auth_openidc
mod_auth_gssapi
5. Call SATOSA OIDC Frontend
13. mod_auth_gssapi validates the SPNEGO token and the residing ticket, sets UserPrincipal environment variable and calls the TLS Backend.
7. Redirection to discovery page where buttons will be shown to trigger Client Auth and Kerberos Auth endpoints.
15. Creates OAUTH2 response with ID Token
OpenIDConnectFrontend
6. converts OAUTH2 request to SATOSA internal request
Routing Microservice
14. Retrieves User ID from the UserPrincipal environment variable, set it to SATOSA internal response and call frontend.
KerberosBackend
SATOSA mod_wsgi
Apache HTTP Server

3.2.3 KPI

To evaluate the successful implementation of this module we plan on testing the following:

1. Login with the microservice must be possible with Kerberos tickets via SPNEGO/GSSAPI in web browsers. The implementation must be able to successfully authenticate a user, who presents a valid Kerberos5 service ticket, and extract the user identifier. This should be recognisable both, in the Apache web server logs and the Satosa application logs.
2. A non-valid Kerberos5 service ticket (i.e. expired or issued for another service) must be blocked from authentication.
3. For testing purposes, the authentication module must be compatible with Satosa‘s OIDC frontend. Using the implicit flow, the extracted user identifier from the ticket must be transmitted to the client if the authentication was successful.

3.3 Satosa Module For Authentication With TLS-KDH

3.3.1 Goal Description

The scope of this task is a PoC implementation of a backend module in Satosa to support authentication via TLS-KDH. For this purpose, the modules for TLS and Kerberos authentication will be reused as much as possible.

It is yet to be determined to what extent this is actually possible. Ideally, the demonstration of the TLS-KDH flow is done in web-browser-based authorisation flows (e.g. OIDC id_token or authorisation code flows) with the TLS-KDH client as the browser which uses Kerberos tickets. As many browsers currently do not support this it is uncertain whether this goal can be reached.

Keeping in mind the aforementioned uncertainties, the plan is to complete the PoC pursuing the following two sub-goals instead:

1. A simple client-server hello world application using the TLS-KDH protocol.
2. A proxy application listening to HTTPS/TLS requests from a client (browser), sends out HTTPS/TLS-KDH requests to OP server, then receives HTTPS/TLS-KDH response from the OP server, and finally responds to the client on HTTPS/TLS with a HTTP status code and the HTTP headers sent by the OP.

The completion of goal 1 is the minimal requirement for this PoC, whereas accomplishing thesecond goal would be a question of time and access to the necessary technology.

We are aware that as of now the TLS-KDH specification is still undergoing changes; they are expected to be finalised within the next couple of months. The current plan is based on version 6 of the IETF draft and for now we plan our PoC with this version. Based on the exact timing we might deviate from some implementation details to better match the final specification. However, this is only possible if specification and the according changes to GnuTLS are finalised before we start implementing our PoC for TLS-KDH.

3.3.2 Concept

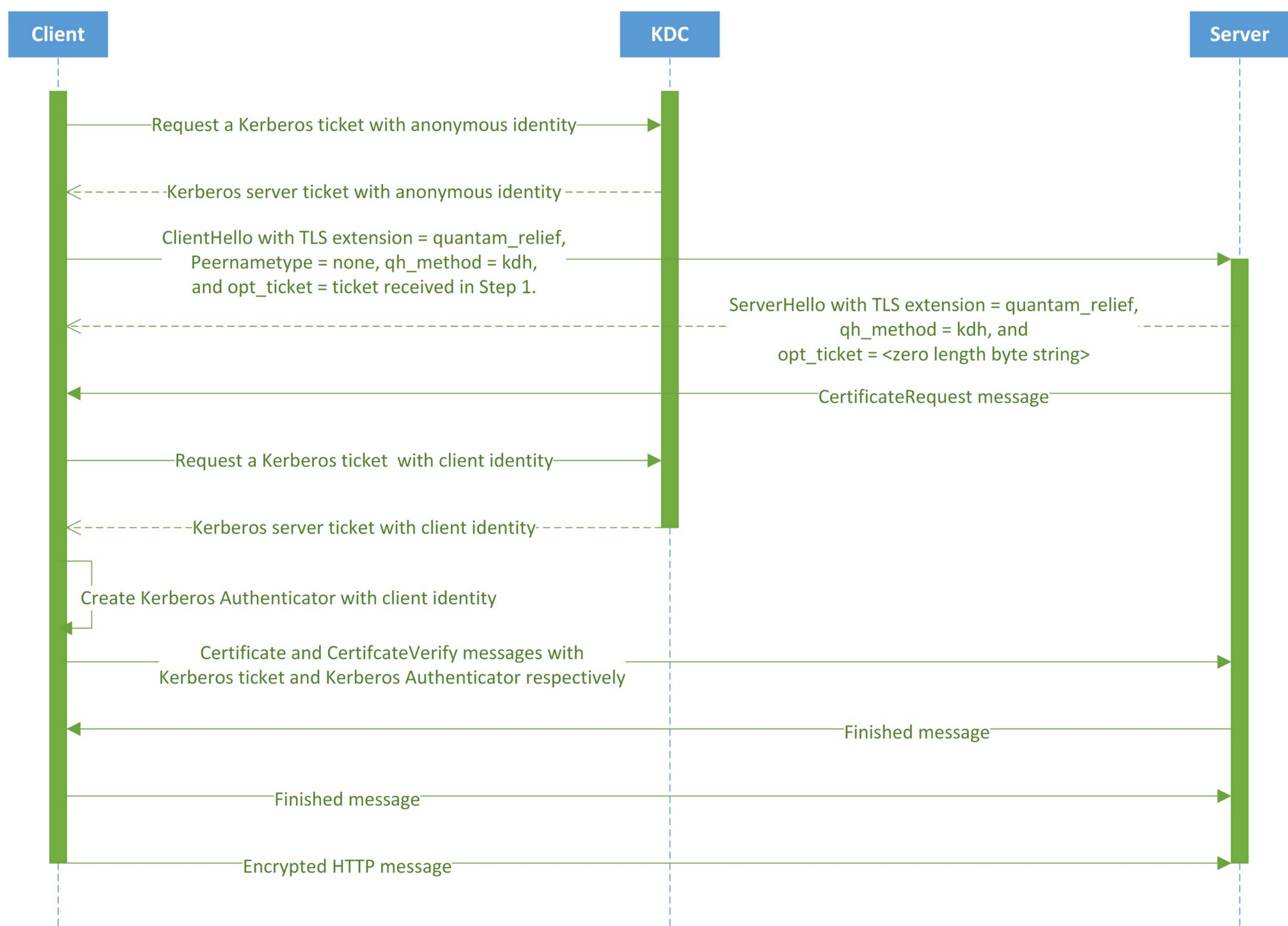
3.3.2.1 Goal 1 – Client-Server Hello World Application

A simple client-server hello world application will be created as part of this goal. The application uses the already created TLS-KDH prototype which is based on GnuTLS. This would be implemented in the programming language C in line with the GnuTLS programming interface.

The client will initiate a TLS-KDH session with the server: this would include all the TLS 1.3 handshake steps with the inclusion of the `quantum_relief` TLS extension in the client hello and server hello TLS messages. Furthermore, the server-side of the application will be configured to request a CertificateRequest TLS message from the client to verify the client's identity. The client would subsequently send the certificate and CertificateVerify messages to the server. For the `client_certificate_type`, a Kerberos ticket would be used. At the client-side, KDC would be configured to generate tickets for the initial client hello TLS message as well as the subsequent Certificate TLS message.

Once the TLS-KDH session is established, the client would send the text, "Hello" and the server would reply with the text, "World". The logging would be added to the application to assert the proper execution of the TLS-KDH handshake.

The following diagram illustrates the described flow:

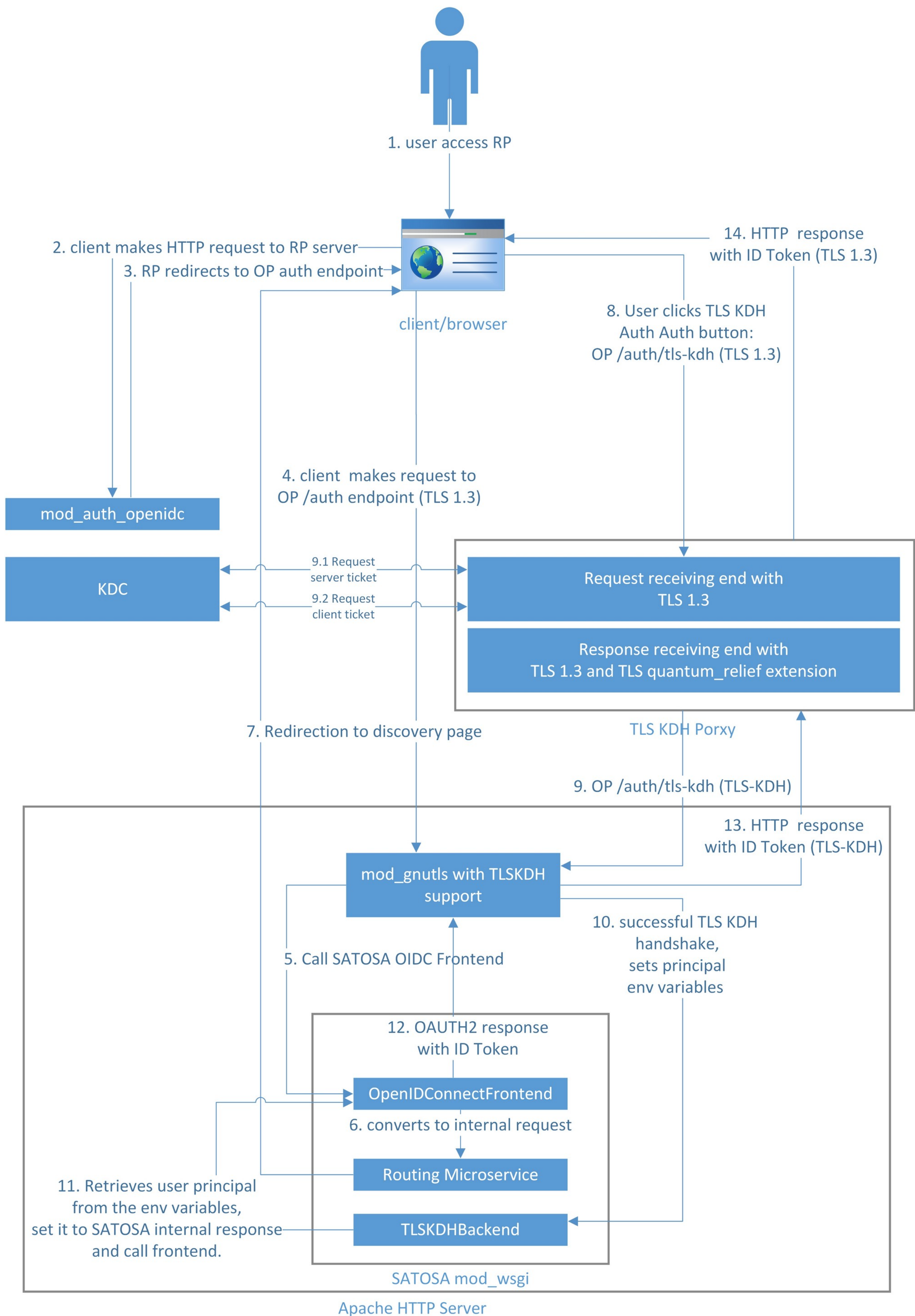


3.3.2.2 Goal 2 – TLS-KDH Proxy

The part of this goal is to test HTTPS over TLS-KDH. To test such an HTTPS flow, the idea is to add a third button to the discovery page mentioned under section 3.1 and section 3.2, clicking on this button would continue the id_token flow via TLS-KDH connection with the new TLS-KDH Satosa backend. The steps for this flow are specified below. The initial steps up until the flow reaches the discovery page would be the same as the steps 1-4 under section 3.1.2, hence the following steps pick up at the discovery page.

1. The user clicks on the TLS-KDH authentication button on the discovery page which makes the following HTTPS call: `https://op/tls-kdh/authorization`
2. The HTTPS request is intercepted by the TLS-KDH proxy which is implemented in the programming language C, using a gnutls fork with TLS-KDH support. The request receiving end of the proxy adheres to HTTPS/TLS 1.3 requests. At this point, the proxy already has an open connection to the OP server. This connection is based on TLS-KDH (with GNUTLS_ENABLE_KDH, GNUTLS_ENABLE_QUANTUM_RELIEF flags set). The request receiving end of the proxy extracts all the HTTP request headers and forwards the request to the OP using its previously established TLS-KDH connection. The TLS-KDH proxy interacts with KDC to fetch server and client tickets to establish the TLS-KDH connection.
3. OP server uses the mod_gnutls apache module with a GnuTLS fork including TLS KDH support. mod_gnutls would perform the TLS 1.3 handshake with the quantum_relief TLS extension. The mod_gnutls is also configured in a way that it will ask the client for its identity using the CertificateRequest TLS message.
4. If the TLS 1.3 handshake is successful, the request reaches the TLSKDH Satosa backend. However, if the TLS handshake is not successful no matter why, the OIDC authorisation request will not even reach OP (SATOSA) and will result in an authentication failure for the client (RP).
5. The environment variable set by mod_gnutls which carries the certificate attribute containing the client ticket's principal is processed in the TLS-KDH backend and is in compliance with Satosa's internal response data structure. Consequently, the OpenIDConnect frontend utilises this to generate the ID token as part of a successful OIDC authorisation response.
6. The OIDC id_token response is transmitted back to the TLS-KDH proxy via mod_gnutls.
7. The proxy extracts the HTTP response headers which also includes the location header with the redirect URL and id_token, the URL encoded query parameter. Then, the proxy uses its other connection with TLS 1.3 to reply to the client with the same HTTP status code as the one originally sent by the OP including all HTTP response headers.

The following high-level architecture diagram illustrates how the execution of the above-mentioned steps proceeds across various components of the PoC system:


1. user access RP
2. client makes HTTP request to RP server
3. RP redirects to OP auth endpoint
client/browser
14. HTTP response
with ID Token (TLS 1.3)
8. User clicks TLS KDH
Auth Auth button:
OP /auth/tls-kdh (TLS 1.3)
4. client makes request to
OP /auth endpoint (TLS 1.3)
mod_auth_openidc
KDC
9.1 Request
server ticket
9.2 Request
client ticket
Request receiving end with
TLS 1.3
Response receiving end with
TLS 1.3 and TLS quantum_relief extension
7. Redirection to discovery page
TLS KDH Porxy
9. OP /auth/tls-kdh (TLS-KDH)
13. HTTP response
with ID Token (TLS-KDH)
mod_gnutls with TLSKDH
support
5. Call SATOSA OIDC Frontend
10. successful TLS KDH
handshake,
sets principal
env variables
12. OAUTH2 response
with ID Token
OpenIDConnectFrontend
6. converts to internal request
Routing Microservice
11. Retrieves user principal
from the env variables,
set it to SATOSA internal response
and call frontend.
TLSKDHBackend
SATOSA mod_wsgi
Apache HTTP Server

3.3.3 KPI

The success of the implementation will be measured based on the following two goals.
For goal 1:

1. In a demo infrastructure authentication with TLS-KDH must be possible. By presenting a valid Kerberos5 ticket and using the TLS-KDH protocol flow, authentication in a simple client-server application must be possible. During the authentication the user identifier must be extracted from the Kerberos5 ticket and observable in at least the TLS server logs.
2. The same setup must block authentication in case of misusage of the TLS-KDH flow or a non-valid Kerberos5 ticket (e.g. expired)

For goal 2:

1. This solution must fulfill the same requirements, further it also must integrate well into the Satosa OIDC frontend. For this purpose proxying the authentication request via a proxy solution is acceptable. The extracted user identifier from the Kerberos5 tickets used on the proxy must be included in Satosa‘s OIDC response when using OIDC implicit flow.

3.4 Evaluation of TLS-KDH for Corteza

The evaluation of using TLS-KDH in other scenarios, such as Corteza, will be done in a later deliverable.

4 Summary

We are planning three PoC implementations of authentication methods in the open source software Satosa. TLS client certificates and Kerberos attempt to integrate present day authentication mechanisms into Satosa and can be used straight away. This further enhances the use cases for Satosa to provide more flexibility also in non-proxy (i.e. delegated authentication via SSO protocols) scenarios.

The third PoC is concerned with the evaluation of the protocol, and existing prototype implementations for TLS-KDH and their usage in Satosa. While our solution is not intended for widespread usage today, the output will be crucial moving forwards towards ultimately adopting TLS-KDH.

A fourth deliverable will be the theoretical reflection on the applicability of this avant-garde technology to a modern business work platform like Corteza.