Fenrir – DRAFT v.05

Authentication and authorization in federated environments

Luca Fulchir
Udine, IT
luker@fenrirproject.org
https://www.fenrirproject.org

Abstract

TLS is the most widely used security protocol, but it has its limitations and is rarely used to its full potential. It lacks authorization, has no support for federation, relies on the Certificate Authorities and on the complex X.509 format.

Today TLS is usually used just for encryption, while authentication, session identification, and other features are implemented (again) at application level. This pushes too much complexity on the final developers, and duplicates many features of the OSI stack. Further complexity is pushed on the developers as TLS only provides reliable, datastream transfer.

This paper proposes a new protocol that includes the features of TLS, plus fine-grained authorization, federation support, works for both reliable and unreliable connections, has multistream connection support, works with datagram or datastream transmission modes, has multihoming, mobility and multicast support, is token-based, to avoid the use and reuse of the main user password and does not require clock synchronization.

The end result is a secure, efficient and extremely flexible solution that is also very simple to use from a end user perspective, and it simplifies application and service development, by decoupling the management of the security from the application itself.

Keywords Protocols, Encryption, Authentication, Authorization, Federation, Fenrir

Introduction

TLS is a widely used and tested protocol, but few projects use its full potential, as session resumption, authentication after the connection and other features have been introduced after the initial standard. Applications do not want the added complexity of having to manage other features – not even authentication methods other than the user/password pair.

TLS is also limited as it does not include all features one can need (like authorization) and some details such as session identification are not always exposed. This required the development of higher level protocols such as OAuth.

Using TLS requires a bytestream reliable connection (TCP), which is easy to spoof. DTLS was thus needed for datagram unreliable connection. Since the two do not support any multistream capabilities, an application might have to handle different connections to the same servers (e.g.: HTTP pipelining). And all of this still excludes use cases that like reliable datagram delivery or multicast delivery. Multicast extensions have been proposed, but their efficacy is questionable as the same keys are shared among all participants.

The OSI protocol stack identifies the transport, session and presentation layer before the application layer, and each layer should be independent. Today's protocols however need to check information between various layers for example to assure that the encrypted data comes from the right source, or that the user session is tied to a specific decryption middleware and/or layer 4 connection. This introduces complexity and duplication of features, as things like session identification are done multiple times, each time with different security properties: for example in a typical web session we have different session identification at the TCP, TLS, HTTP(cookie) and OAuth level, and each of these must be tightly tied with one another, or security problems can arise. This breaks the independent-layer OSI model, and adds complexity and attack surface. Part of this complexity is also pushed upwards in the stack, and the application ends up having to manage it.

In a typical HTTP application stack authentication can be done at multiple levels (e.g.: TLS, HTTP, OAuth), but none of these support federation, and none of these include explicit support for multiple levels of authorizations.

Finally, The X.509 certificate model brings a lot of unneeded complexity, especially if the only concern is to secure a connection between the client and the server. This complexity has been exploited multiple times, and many certificate authorities have been shown to be weaker than expected.

To solve all these problems and introduce new features and flexibility, I designed the Fenrir protocol. It can run right on top of the IP protocol, but includes support for UDP tunnelling for an easier initial deployment, to avoid firewall problems.

The goals of this protocol are multiple: the first is the simplification of the OSI stack, putting together the 4th and 5th layer with encryption while trying to be as independent as possible from the 3rd layer. This way security properties can be easily checked in a single place. Secondly, we need to be flexible enough for any use, so ordered, unordered, reliable, unreliable connection, unicast and multicast, multiplexed, authenticated or anonymous, encrypted and cleartext (but always HMAC-like-authenticated) transports must be supported. A third goal is to have flexible, fine-grained control on authentication and authorization with federation support.

Future work will include secure proxy support, where the proxy works only with encrypted data, and is usable in a CDN (Content

Delivery Network). One of the problems introduced by HTTPS is that the content is no longer cacheable. To bypass this limitations CDN networks like cloudfare ask for the keys, or create special software that transmits them the connection key. This however changes the trust model, as the certificate does not assign trust to only the server anymore, but also to a hidden third entity which has complete control on the connection.

A final future work will be client-to-client communication, with a STUN-LIKE protocol that will enable direct client-to-client communication.

The end result will be a simpler application security model, were we won't have to test different interactions of different protocols. The interface to the final user will be simpler, lowering application complexity as there will be only one middle-ware to interact with, and we will have an authentication and authorization protocol that manages federation without the complications of a framework like OAuth.

Terminology

As a short introduction we need to define some terminology:

- **Authentication**: The process of determining if a user is really who he says he is
- **Authorization**: The privileges associated with the authenticated user
- **Authorization Lattice**: a complete lattice – an ordered way of representing hierarchical authorizations. See pg. 3

The Federated model

To promote a standard and interoperable login method we use a terminology similar to the kerberos federation:

- **Application**: e.g.: the mail client, or browser...
- **Client Manager**: An application running on the end user device that manages all the tokens, authentications and authorizations
- **Service**: the server application that the application wants to access.
- **Authentication Server**: The server that manages login data, authentication and authorization for a specific domain

The threat model

The goal is to grant confidentiality and integrity of the data exchanged between the Client Manager, the Authentication Server and the Service, and of the data between the Application and the Service. The attacker has complete control on the data transmitted or received. There is a trusted party (DNSSEC) that can be contacted in a secure manner.

Being in a federated environment, both the Client Manager and the service will assume that their Authentication Server is not malicious, at least for the first connection.

The client only has to trust the Authentication Server for its own domain, but the Authentication Server will not be able to impersonate the user on services where the user has already set up an account.

For everything else, the attacker is the network.

The local threats

Our environment looks a lot like Kerberos, and in fact the distinction between Service, Authentication Server and Client Manager comes from there. As in Kerberos, the end user will need to run an application that manages all tokens and authentications.

The **Client Manager** will be the target of a lot of attacks, just like any local password manager would be. This paper will not focus on those attacks as they are implementation and environment dependent. We will only focus on the protocol defence.

High-level overview

Fenrir is a federated protocol, so each domain will have its own Authentication Server (a single A.S. can support multiple domains, too), and each user is identified by its username and domain. Thanks to the integrated federation support, each user won't need to create a different user for each domain or service that they want to access.

This is a token-based authentication protocol, which means that the user, its authentication and its authorization are represented by a token, which can be thought of as a 256 bit string of random content. This will lower the amount of times a password is needed, making it easier to use from a end-user point of view, and also more secure. The protocol will not require clock synchronizations between any party.

A Client Manager will run on each user device. This will manage all the tokens for a user. Each Client Manager can be identified by the Authentication Server, so that this will be able to distinguish which device has access to which services. This way each device can be easily and securely deactivated if lost or stolen.

The Client Manager will only need the user login password once, at setup time, ad then must forget it and switch to token authentication. This will prevent password reuse and loss. To make things clearer, the Client Manager should use the user password only once in all its lifetime (except for configuration resets).

The Authentication Server will be the only one that will have access to the users login data, and once a user has authenticated, it will inform the Service that a new Client has connected, without revealing information on the key exchange or authentication data. This will further divide the Service from the user authentication.

When an application wants to connect to a service, first the Client Manager needs to contact the Services' Authentication Server. Once the authentication is verified, the Authentication Server will send to the Service the user information. The Service will generate a session key, xor it with a secret shared between the user and the service, and send it to the Authentication Server, which will rely it to the Client Manager. The information relied will include ip addresses, encryption keys, and connection identifiers.
The Client Manager can now give the encryption key, ip and connection id to the application. As for the Authentication Server and its Services, this divides the end-user authentication security from the end-user application security.

Having the Authentication Server distribute the ip address of the Services will help in managing load balancing, or geographic traffic redirection.

Each Service will be identified by a domain name and a 16-byte service identifier. This means that each service can be udp-tunnelled to any port, aiding setups of multiple different services in a single machine (virtual hosting).

Each Service will also have an **Authorization Lattice**, which represents an order of the possible authorizations. This small lattice will be transferred to the authentication server and to the Client Manager, so that a token can be tied to a particular authorization, and so that the Client Manager can further limit the scope of the token when authorizing applications. This means that users won't have to rely on the applications self-limiting themselves, but the protocol will actually enforce the authorization limitation.

The end-user view

The aim is to have a single, secure application that manages all authentications and authorizations, and to hide all the details from the user, while only asking for confirmations.

The user will register its username with its Client Manager, and then for each service we need to ask the user 2 usernames:

- **authentication username**: One of the usernames registered in the Client Manager . We need to ask this because the user could have multiple accounts registered on its Client Manager, and because providing a list of possible usernames to each application might be considered an information leak.
- **service username**: The account to use on the service. The user could have multiple accounts for each service and we need to know which one he wants to use.

The two accounts can be the same, and should default to "anonymous@given_tld". Using two accounts might seem counterintuitive, but lets us cover all authentication scenarios:

- **anonymous** both usernames must be set to "anonymous@given_tld".
- **federated** auth. username is in a different domain than the service username, which is in the same domain of the service. This allows multiple accounts usage with the same auth.username
- **local** both usernames are the same, and in the same domain of the service.

Everything else will be automatic and requires only a user confirmation.

The Authorization Lattice

The name should already give an idea on what this is: a connected ordered graph where we have a bottom element, a top element, where the bottom element represent no privileges, and the top element represents all privileges.

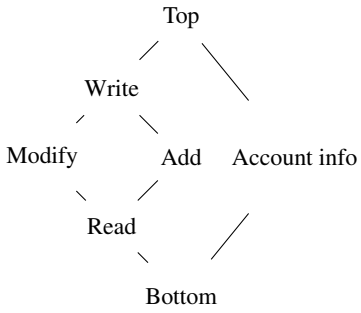


Figure 1. Example of a complete lattice of privileges

We said before that the user will have multiple tokens, one for each service where he wants to identify himself. Each token will be tied to a specific element in the Authorization Lattice, and this authorization lattice will be synchronized between the service, the authentication server and the Client Manager.

Having the lattice at the Authorization Server means that we will be able to decide the maximum permissions of each user device, for each service, both in our domain and in other domains.

Having the lattice in the Client Manager means that the Client manager itself is able to further limit each application permissions.

The end result is the finest of controls on the authorization, in a secure way: if the user device is compromised or lost, but had only read access to an account, that will be the extent of the loss.

Since each application has different needs, each service will have its own Authorization Lattice. To avoid excessive growth of the lattice, and to avoid having to store too much data, each lattice will be limited to 64 nodes, and each node will be limited to a 25-character name: this will put the maximum size of an uncompressed lattice around 2Kbytes (one or 2 packets maximum).

The Federation

The idea is to divide everything in administrative domains.

The advantages of this approach are two: we can easily standardize on a single cross-domain login algorithm, and we divide completely the services and the applications from the login data.

Since services and applications are not always checked and tested as much as security protocol implementations, this should increase the overall application security.

The service now does not need any access to the user password database, so the compromise of a service does not mean that we have to force all users to change their authentication data.

Forcing an Authentication Server might still be possible, but the overall attack surface is much lower than the whole service. Moreover the compromise of an Authentication Server should not necessarily mean that the services databases needs to be checked and reset, too. Such an attack would also have less impact as the Authentication Server itself can not impersonate its users into services where the user has logged in at least once. This is possible thanks to shared secrets that are setup between the Client Manager and the service, and between the Client Manager and the authentication server.

It should be noted that although the protocol is built with federation in mind, nothing stops developers to force an older client-server architecture (by putting together auth.server and service) or even a distributed architecture.

Services

Historically, each service was identified by the port on which its protocol run, so TCP port 80 is reserved for HTTP, 443 for HTTPS and so on.

This means that the real identifier of a service is the IP:protocol:ports tuple. This is limiting, as it can require a different IP for every service that runs on the same port as another (or for multiple instances of the same service), and we need to give administrative privileges to at least a part of those applications.

This was necessary in the past because the domain lookup did not include the service lookup: the DNS queries return an IP address, but you want to connect to a service, not just to the machine, so you need a service identifier. This was the protocol:port pair.

To avoid the firewall and port limitations today almost every application is designed to run on top of HTTP, making everything inefficient and requiring a lot of middleware software.

To solve all these problems, in Fenrir we redesign the identification of a service.

First of all, the domain lookup must include some information on where the authentication server is running. This is done by using an external directory service (DNSSEC). A lookup for the domain also returns enough information to be able to connect to the Authentication Server for that domain (public keys, udp port, ip addresses..see pg 5). Once the client connects to the Authentication Server it can ask for a specific service, and the answer can contain information such as the IP and UDP port.

This means that we can have multiple Authentication Servers and services on the same machine, on whichever UDP port we want. Even when not using UDP tunnelling, each A.S. will be identified by its key-id. To avoid conflicting ID we could design a listen syscall to succeed only if an additional password matches for the key-id we want to use, or we could more simply forward the packets to all applications that use that key-id: if they are not

the intended recipient, they will perceive it as garbage (due to the signatures) and will drop it.

As of today a typical linux distribution has a file (/etc/services) that tracks the tcp and udp ports for each service, for a total of almost 5900 protocols, and the total of the TCP and UDP tracked ports is more than 11000.

Since the applications that will make use of Fenrir are now completely UDP-port independent, and will not need to reserve other ports, in Fenrir we will track all protocols in a new public database, and we will simply assign one 16-byte ID for each protocol. The Service will then be identified by its protocol-id and its domain.

The advantages of this approach are that we can run as many services, of as many domains as we want, without any risk of collision, as the IP address is not an identifier anymore. We do not need to run services on any specific port, so any free UDP port is fine, since the UDP port of a service is given to the client by the Authentication Server.

Trust model

Fenrir breaks away from the X.509 format and from the certificate authority model.

The X.509 certificate is difficult to parse, and the current parsers in the most common TLS implementations range from 10k to 35k lines of code. It is unsurprising that many bugs have been found and exploited in the handling of such format.

X.509 certificates include a validity of the certificate, so that a certificate can not be used before or after a certain date. While this seems obvious, it creates two problems: the first is clock synchronization, which is almost always in cleartext and unreliable on the internet, and the second is certificate revocation. This was initially thought to be resolved with CRL (Certificate Revocation Lists), but these lists were almost never updated on the clients. A new protocol (OCSP) was thus introduced to check in real time if a certificate was still valid, eliminating the advantages of offline checks on the validity of the certificate.

The certificate authority model is also questionable: multiple companies have been found to have issued certificates that should not have been given, even for high-ranking domains like google's.

These problems put a very bad light on this trust model, but what other model can we use that will grant freshness and validity of the public keys? In Fenrir we require a generic directory service to get the public keys, so that future technologies might supplement what is lacking today. The current choice is DNSSEC, which was designed exactly for this problem, but on the scope of DNS queries instead of keys.

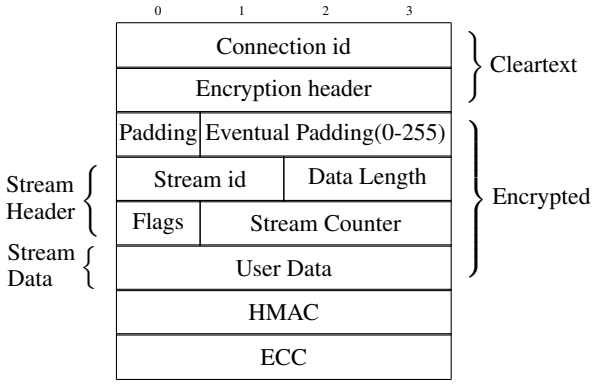
By embedding all our data as a Z85-encoded binary in a TXT record, we can have both freshness of the data and control of our keys.

Again, Fenrir only uses the DNSSEC system due to its widespread usage, but experiments with other trust models and protocols are encouraged and easily pluggable in the code.

The main DNSSEC record, ***_fenrir.domain.tld*** will hold the public key and connection information for the Authentication Server. There will be an additional record for every service, ***id._fenrir.domain.tld*** that will hold the public key of the service. This secondary public key will actually be needed only the first time a client connects to an unknown service, to provide additional security against a malicious or hacked authentication server.

The transport details

At the lower level, the protocol is designed to run on top of IP, but supports UDP as a lightweight tunnel, to bypass firewall problems.



The header is self-explanatory, but is also very flexible, and only the full version is reported here.

The handshake header has the same structure, but is unencrypted, and there is no HMAC field.

Starting at the beginning we have:

- **Connection id**: an identifier for the connection. To avoid needing to synchronize the used connection ids between communicating parties, each party will decide for a connection id and will communicate it in the handshake phase. So while transmitting each party will use the connection id the other client requested. This will also mean that a single client can not have more than 2^{32} connections, still, that is considered enough for a single device, as just keeping track of all the connections would require hundreds of gigabytes of memory.
- **Crypto header**: This will usually be only a counter, or other info to let the receiving party check and decrypt the packet immediately. The actual size is dependent on the negotiated algorithms.
- **Padding**: Padding at the beginning (0-255 bytes), random in length and content, so that it will be more difficult to infer anything just by the packet length (think CRIME/BREACH).
- **Stream**: composed of a **Header** and the **User Data**, this part can be repeated in the packet, and provides support for multiple streams in a single connection.
  - **Stream Id**: a randomly-assigned stream id. randomness is not necessary, but should provide less window for known-plaintext attacks.
  - **Data length**: simple counter of the user data transmitted in this stream. Stream header length not included.
  - **Flags**: (2 bits) used to signal beginning/end of user data fragment
  - **Stream Counter**: A total of 30 bits to support higher throughput on high latency networks. Stream length not included.
- **HMAC**: Hash-based Message Authentication Code, or general signature to confirm the authenticity and integrity of all the previous data. Actual size depends on the agreed algorithms (can be zero if the function is already provided by AEAD cipher).
- **ECC**: Error Correcting Code, to avoid retransmissions for exceptional errors. Algorithm dependent on handshake agreement.

The flags in each stream introduce support for managing the beginning and ending of the user message. This way we support both datagram-style transmission, including fragmentation and reassembly, and bytestream, TCP-like connections.

The whole packet is be encrypted following the **Encrypt-then-MAC** mode, or AEAD.

Of all the connection ids, the IDs 0, 1 and 2 are reserved. ID 0 will be used during connection setup, ID 1 is reserved for secure proxy, and 2 for multicast.

The header looks a bit like the SCTP one. The main difference with SCTP is that we do not have different headers for different stream types. For example the control stream, which will manage all the connection information is just a normal stream, with a random identifier, created during the connection setup phase.

The padding

Padding (0-255 bytes) is put at the beginning for different reasons: we put more randomness in the packet structure and we provide more initial entropy in the data, thus making known-plaintext more difficult.

The CRIME, BREACH and recent BICYCLE attacks against TLS were based on the fact that very little changes between multiple HTTPS requests. By randomizing the packet length, even by little, and the content, Fenrir should prove to be more resistant to such attacks, even with protocols with long, static headers, like HTTP.

It should be noted, however, that random padding can increase the chance of misaligned data. While this is not a big problem for newer x86 cpus, it might become one in high performance scenarios, so the alignment (1,2,4,8 bytes) can be specified in the handshake.

Multicast

Fenrir includes support for multicast communications. All multicast streams will have a unicast stream associated with them. This will provide an easy way to synchronize on things like key renewal or to transmit error recovery data.

By using the RaptorQ algorithm in both the unicast and multicast transmissions we will be able to provide a reliable multicast delivery. RaptorQ support is provided trough a separate library, libRaptorQ[1], developed for this project.

Finally, multicast transmissions can not use the usual HMAC, as this would be trivially forgeable by any other party in the multicast group. To avoid a MITM by an attacker in our multicast group we can only use a private/public scheme to sign each packet. While RSA might be slow, elliptic curve algorithms are much quicker and should not pose a performance problem.

Since each connection id is decided by the *receiving* party, and we do not want to synchronize connection ids between computers on the network, we have reserved the connection ID ”2” as the identifier for all multicast packets. The next field will be a 128bit sha3 hash of the public key used to sign the data, which will serve as an identifier of the multicast transmission. In the future Fenrir might move to shorter hashes, as long as the clashes are handled by a linked list not to lose data.

Error Recovery

There can be up to two levels of error recovery. The first is the ECC field, which aims at correcting bitflips and general transmission errors in the packet.

While the HMAC gives us assurance on the authenticity of the data received, just a bitflip during the network transmission can cause the whole packet to be dropped. An error correcting code can be used to reduce retransmissions.

The second level of error recovery is implemented through the RaptorQ algorithm and libRaptorQ[1] library. By using this forward error correcting code we can recover any lost packet, as long as we have a repair packet for every original packet we lost. This will further reduce retransmissions and reliance on the ACK mechanism. While the first level of Eror correcting code is appended to the packet, this second level is handled through in-band streams. This way we can protect only the needed streams, and not the unreliable ones.

Both error recovery mechanisms are negotiated for every connection, as they will increase the computation power needed, and new algorithms might come out in the future.

Connection Setup

When a client wants to connect to a service, it needs to know 2 things: the service id and the service domain. The Client Manager will then ask which user should be used, and will now handle the authentication phase.

We assume that the Client manager already has a connection with the Authentication Server of the domain of the authentication username. The Client Manager now has to contact the Authentication Server of the service. To know where that is, Fenrir uses an external directory service.

For example, using the existing DNSSEC infrastructure, if the client wants to connect to “sub.example.com”, it makes a DNSSEC TXT query for “_fenrir.sub.example.com”. The answer will be a z85 (base85) encoded binary including a public key and the ip address(es) of the Authentication server responsible for the “sub.example.com” domain. A single z85 string was chosen to keep the encoded binary as short as possible, and to avoid asking multiple records, which would have meant much bigger response due to multiple DNSSEC signatures. Z85 is a variant of base85 where the encoded string is parsing-safe, as there are no quotes, double quotes or backslashes in the generated output.

With these information, the Client Manager can contact the Authentication Server.

Fenrir supports 3 different Handshakes:

- **Full-Security**: TLS-like handshake, 3 Round Trips.
- **Stateful Exchange**: Only 2 RTS, but requires to store a state from the beginning.
- **Directory-Synchronized**: Only 1 RT, but requires the Authentication Server and the public key directory service to constantly synchronize on the public key to use.

The handshakes have been named in order of robustness against (D)DoS attacks.

The connection id 0 is reserved for the handshakes. The packet is not encrypted as there are no shared secrets yet. The stream id is chosen by the sender and is random, but the server will not have to keep track of it.
Each handshake packet will contain only one stream, which will contain a key id, so the server knows with which one of its keys it needs to check the signatures, and an identifier used to distinguish the various phases of the various handshakes.

On the final RT of the handshake the 2 parties exchange connection ids and authorization token. The token is not sent as cleartext. It is XORed with a shared secret between the Auth.Srv and the Client Manager. After answering, the Authentication Server and the Client Manager will XOR their session keys with the shared secret. By doing this we prevent MITM of the connection even in the case of a compromised directory service.

The above mentioned shared secret is generated by a second public key exchange during the very first login of the user on that device.

Full security

The most secure one uses syncookies, requires the least amount of state to be kept in memory, and can be explained as follows:

- **RT 1**: nonce exchange, the client sends the supported algorithms(by preference), the server reply contains the selected one plus a timestamp, supported authentication algorithms, and signature of the request and reply.
- **RT 2**: nonce exchange, the client gives back the previous signature, an ephemeral key. The server replies with the ephemeral public key and then signs everything, including the client message. A state is created with only the key generated from the public key exchange.
- **RT 3**: The client authenticates, sends connection initialization data, and the server answers with a positive/negative message and connection data.

Overall it is very similar to the TLS exchange.

Stateful Exchange

This is a quicker version of the TLS handshake, were we eliminate the syncookie, and only renew the ephemeral public key every couple of minutes. This means that all the connections in this timeframe will be derived from the same ephemeral key.

The advantage is that we avoid one RT and we have less CPU overhead due to less ephemeral key regeneration.

The disadvantage is a slightly bigger concern towards DoSes, since we need to keep some state between the 1st and 2nd RT.

The issue of slower ephemeral key regeneration should not be a big concern, as long as the time between regeneration is not too high.

A similar key exchange to this is found in Google's QUIC[3] protocol.

- **RT1**: client hello, client supported algorithms, server answers with public key, chosen algorithms, supported authentication algorithms.
- **RT2**: client answers with key exchange data and authentication data, server answers with handshake completion confirmation and connection data.

Directory-Synchronized

Taken straight from minimaLT[2], it involves strict collaboration with the directory services used to distribute the public keys.

The idea is to ditch the long term public key, and switch completely to using just ephemeral public keys.

To do this, however, we need to synchronize the distribution of the ephemeral public keys to be sure that everyone agrees on which public key to use at any time.

The ephemeral public key will be distributed directly in place of the long term public key, and the Authentication Server and the Directory Service will synchronize on when to publish the new public keys. Since we have a key-identifier, the Authentication server can support both the new and old published ephemeral keys for a short time, to avoid inconsistencies in clock synchronization.

The advantages are a quick connection with only one RT.

The disadvantage is a much bigger risk of DoS attacks.

It should be noted that the client must now be able to generate a shared key only from the data published on the directory service. Also, this way there is no algorithm agreement, so only the server-published mechanism can work.

0-RT protocols do not provide any assurance on the sender IP, and therefore create an extremely easy amplification attack. Forcing at least one RT forces the attacker to at least intercept all packets for the target network. But if the attacker already has this ability, spoofing already provides an easier and congestion-control-free way of DoSing the target, so targeting this protocol becomes much less appealing.

- **RT1**: exchange of authentication and key-exchange data.

The Federated Authentication

Once a secure connection has been established between the Client Manager and the Authentication Server, and during authentication, the client tells the Authentication Server which service on which domain it wants to use.

If the client wants to access a service in its own domain, then it only needs to connect to its own Authentication Server.

When it wants to connect to a service in an other domain, it will need one connection to its own Authentication Server and one to the other domain's Authentication Server. Once the second Authentication Server has confirmed its identity with the Authentication Server of the client's domain, it will let the Client access its Services.

To confirm the identity of the user, without revealing any data that can be used to impersonate the user, we use tokens. A token is a random, 256bit string.

In our example, the user "user@example.com" connects to the service "www.example2.org".

The Client Manager connects to the Authentication Server for "example.com" and authenticates himself as "user@example.com". This is done as soon as the Client manager is started, and the connection is persistent, and will be used for all the queries for that user/device pair.

As soon as an application wants to connect to "www.example2.com", this is what happens:

- The application tells to its Client Manager it wants to connect to "www.example2.com", service "www"
- If the Client Manager does not have a token for that Service, it asks its Authentication Server for a new token.
- The Client Manager now connects to the Authentication Server for "www.example2.com", and authenticates.
- The Authentication Server for "www.example2.com" checks the token with the "example.com" Authentication Server
- The Authentication Server for "www.example2.com" tells its "www" service that a new user has connected.
- The "www" Service gives its Authentication Service a connection id, session key xored with a secret shared with the client.
- The Authentication Server for "www.example2.com" gives back the encryption key and connection data to the Client Manager.
- The Client Manager gives the application the connection data and keys.

Note that the applications now does not have to do any user authentication, nor does it have to know how to do handshakes or handle tokens, and the same is true for the services.

Thanks to a shared secret between the client and the service, even a compromised authentication service will not be useful in impersonating the user in services where the user already registered, although new registrations will always be possible.

During the very first connection a shared secret might not be available yet. The client will therefore generate a public key, and generate a shared key with the public key of the service, found in the directory service. Then during authentication the client will send the public key to the service.

The delays

This protocol seems to include a lot of round-trip checks, especially if we consider the Full-Security handshake when none of the parties involved have an active connection.

The worst case scenario is a login in an other domain without any active connection, and has a total of:
3RTs for Client-Manager to its Authentication Server connection; 1 RT for a new token; 3 RT for Client Manager to the other Authentication Server connection; 3 RTs for intra-Authentication Server connection which includes the token check, and one RT to tell the Service about the new client.

That is a total of 11 RTs, and sounds a lot, but in reality the initial 3 RTs for Client Manager to its Authentication Server connection are done only once, even before we want to connect to any service, so they should not actually count as delay, and the total goes down to 8 RTs. The RTs between Authentication Servers are usually on networks with less delay between them. The Authentication Server and its Services are usually in the same network, too, so the RT between them are negligible, and the connection between them should be persistent. 7 RT. The client might already have a token it can use for that service, so an other RT can be discarded. 6RT. We can further reduce the RT count by using the other two handshakes.

The best-case scenario is when a Client Manager has already authenticated to its Authentication server and wants to connect to one of its services, and that is just one RT, plus an other (negligible) between the Authentication Server and the Service.

The average case should be when a Client Manager is already connected to its Authentication Server, already has the needed token, and wants to connect to the Service of an other domain. This includes one handshake to the other Authentication Server (1-3RTs), which will then check the token and inform its Service (1-3 RTs), for a total of 2-6 RTs depending on the chosen handshakes.

All in all, the delay should not be too high, and the advantages (no multiple password to remember, fine-grained authorization, separation between applications and security...) should outweigh the possible delay.

Future Work

Secure proxy

By “Secure proxy” we mean a server that can cache certain parts of connection contents, even if they are encrypted, and then serve them to clients that request this.

The “secure” part of the proxy comes from the fact that the proxy handles encrypted data for which it does not have the key.

The services that want this feature will have to explicitly use the feature, as each resource needs to be identified.

The proxy can be an explicit proxy set inside the Client Manager, or a semi-transparent proxy.

To support the semi-transparent proxy operation, the proxy will need to be between the device and the service, and the client will need to send part of the request in clear-text.

To do all of this we need to reserve the number 1 connection id.

OTP Tokens

Managing tokens as Lamport’s OTP (hashed OTP) would grant us the ability to instantly detect unauthorized usage of a token or shared secret.

The only problem to applying this to shared secrets is that the shared secret between the services and the client manager must be shared between the client managers, so the otp feature would not be useful.

Sub protocols

The multistream feature of Fenrir can lead to the design of multiple sub-protocols.

Implementing (for example) file transfer once for everyone would be easier for developers, who could automatically use advanced features of Fenrir without even be aware of them. Audio/video, chats and other use cases make this particular interesting.

Client to Client

Fenrir won’t be complete until the possibility of client-to-client direct communication will be included.

Such a feature is not extremely difficult per se, as we would only have to implement a way to publicly distinguish each device on which the user is logged in.

This however has other issues like privacy and spam abuse which need to be addressed in more detail before an actual implementation is made.

Anonymous connections

That is, authenticated connections in which the user is verified, but its username is not leaked to other Authentication Servers.

While it could be implemented through temporary usernames, it would give too much trust to the Authentication Server. This feature should be limited to pre-shared-secret generation between the service and the client, to avoid leakages.

The ramifications of this aproach have not been fully studied yet.

References

[1] libraptorq: Lgpl3 forward error correction library. https://www.fenrirproject.org/Luker/libRaptorQ.

[2] W. Michael, Petullo Xu, Zhang Jon, A. Solworth, Daniel J. Bernstein, and Tanja Lange. Minimalt: Minimal-latency networking through better security.

[3] Jim Roskind. Quic: Multiplexed stream transport over udp. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit, 2013.