LURK
Internet-Draft
Intended status: Standards Track
Expires: January 4, 2021

D. Migault
Ericsson
I. Boureanu
University of Surrey

July 03, 2020

LURK Extension version 1 for (D)TLS 1.2 Authentication
draft-mglt-lurk-tls12-03

Abstract

This document describes the LURK Extension 'tls12' which enables interactions between a LURK Client and a LURK Server in a context of authentication with (D)TLS 1.2.

Status of This Memo

Copyright Notice

Table of Contents

```
   1.  Introduction  . . . . . . . . . . . . . . . . . . . . . . . .   3
   2.  Terminology and Acronyms  . . . . . . . . . . . . . . . . . .   4
   3.  LURK Header . . . . . . . . . . . . . . . . . . . . . . . . .   4
   4.  rsa_master, rsa_master_with_poh . . . . . . . . . . . . . . .   6
     4.1.  Request Payload . . . . . . . . . . . . . . . . . . . . .   6
       4.1.1.  Perfect Forward Secrecy . . . . . . . . . . . . . . .   8
     4.2.  Response Payload  . . . . . . . . . . . . . . . . . . . .   9
     4.3.  LURK Client Behavior  . . . . . . . . . . . . . . . . . .  10
     4.4.  LURK Server Behavior  . . . . . . . . . . . . . . . . . .  10
   5.  rsa_extended_master, rss_extended_master_with_poh . . . . . .  11
     5.1.  Request Payload . . . . . . . . . . . . . . . . . . . . .  11
     5.2.  Response Payload  . . . . . . . . . . . . . . . . . . . .  12
     5.3.  LURK Client Behavior  . . . . . . . . . . . . . . . . . .  12
     5.4.  LURK Server Behavior  . . . . . . . . . . . . . . . . . .  12
   6.  ecdhe"  . . . . . . . . . . . . . . . . . . . . . . . . . . .  13
     6.1.  Request Payload . . . . . . . . . . . . . . . . . . . . .  13
     6.2.  Response Payload  . . . . . . . . . . . . . . . . . . . .  15
     6.3.  LURK Client Behavior  . . . . . . . . . . . . . . . . . .  15
     6.4.  LURK Server Behavior  . . . . . . . . . . . . . . . . . .  15
   7.  capabilities  . . . . . . . . . . . . . . . . . . . . . . . .  16
     7.1.  Request Payload . . . . . . . . . . . . . . . . . . . . .  16
     7.2.  Response Payload  . . . . . . . . . . . . . . . . . . . .  16
     7.3.  LURK Client Behavior  . . . . . . . . . . . . . . . . . .  18
     7.4.  LURK Server Behavior" . . . . . . . . . . . . . . . . . .  18
   8.  ping  . . . . . . . . . . . . . . . . . . . . . . . . . . . .  18
     8.1.  Request Payload . . . . . . . . . . . . . . . . . . . . .  18
     8.2.  Response Payload  . . . . . . . . . . . . . . . . . . . .  18
     8.3.  LURK Client Behavior  . . . . . . . . . . . . . . . . . .  18
     8.4.  LURK Server Behavior  . . . . . . . . . . . . . . . . . .  19
   9.  Security Considerations . . . . . . . . . . . . . . . . . . .  19
     9.1.  RSA . . . . . . . . . . . . . . . . . . . . . . . . . . .  19
     9.2.  ECDHE . . . . . . . . . . . . . . . . . . . . . . . . . .  20
     9.3.  Perfect Foward Secrecy  . . . . . . . . . . . . . . . . .  21
   10. IANA Considerations . . . . . . . . . . . . . . . . . . . . .  22
   11. Acknowledgments . . . . . . . . . . . . . . . . . . . . . . .  23
   12. Apendix . . . . . . . . . . . . . . . . . . . . . . . . . . .  24
     12.1.  LURK Exchange for TLS RSA Master Secret with Proof of
            Handshake  . . . . . . . . . . . . . . . . . . . . . . .  25
     12.2.  LURK Exchange for TLS RSA Extended Master Secret . . . .  26
     12.3.  LURK Exchange for TLS RSA Extended Master Secret with
            proof of handshake . . . . . . . . . . . . . . . . . . .  28
     12.4.  LURK Exchange for TLS ECDHE Signature  . . . . . . . . .  30
   13. References  . . . . . . . . . . . . . . . . . . . . . . . . .  31
     13.1.  Normative References . . . . . . . . . . . . . . . . . .  31
     13.2.  Informative References . . . . . . . . . . . . . . . . .  32
   Authors' Addresses  . . . . . . . . . . . . . . . . . . . . . . .  33
```

1. Introduction

This document describes the LURK Extension for TLS 1.2 so the LURK Server can implement a Cryptographic Service in a TLS 1.2 [RFC5246] and DTLS 1.2 [RFC6347] context.

More specifically, the LURK Server will be in charge of performing the cryptographic operations associated to the private key of the TLS Server, while other aspects of the termination of the TLS session is handled by other services in the same administrative domain or in a different administrative domain. Most Cryptographic Operations are related to the TLS authentication and the current document limits the Cryptographic Operations to the following authentication methods: RSA and ECDHE_RSA defined in [RFC5246], [RFC6347] as well as ECDHE_ECDSA defined in [RFC8422].

A more detailed description of some use cases foreseen in a TLS context can be found in [I-D.mglt-lurk-tls-use-cases].

HTTPS delegation has been the main concern of the Content Delivery Networks Interconnection (cdni) Working Group and several mechanisms have been designed to delegate the load from an upstream entity to a downstream entity. Entities can be of different nature and may designated differently according to the context. Typically designations includes Content Owner, CDN Provider, Domain Name Owner for example. [I-D.fieau-cdni-https-delegation] provides a details comparison of the various mechanisms applies to the CDN Interconnection, and the remaining of this section positions these mechanisms at a very high level view.

STAR [I-D.ietf-acme-star], [I-D.sheffer-acme-star-request] describes a methods where the domain name owner or the content owner orchestrates the refreshing process between a CA and the CDN (terminating the TLS session). The CDN refreshes regularly and automatically its certificates using [I-D.ietf-acme-acme], which allows the use of short term certificates.

Delegated credentials [I-D.rescorla-tls-subcerts] consists having a certificate that enables the servers to generates some "delegated credentials".

STAR and "delegated credentials" both require some changes performed by the CA - new certificate type for the delegated credentials and new interfaces for the delegated and delegating entity for STAR. In both case the TLS Client authenticates the delegated entity. While STAR does not require changes on the TLS Client, the "delegated credential" solution does. In both cases, the delegation is controlled by limiting in time (7 days), which is also the limit of

use of a stolen key or a rogue server. Such delegation provides a high scalability of the architecture and prevents additional delays when a TLS session is established.

The LURK Architecture [I-D.mglt-lurk-lurk] and the LURK Extension 'tls12' do not proceed to the delegation of the HTTPS delegation by delegating the entire TLS termination. Instead, the TLS termination is split into sub services, for example one associated to the networking part and one associated to the cryptographic operation. While micro services associated to the networking part are delegated, the micro service associated to the cryptographic operation may not be delegated. As a result, LURK Architecture is focused on the protection of the Cryptographic Material and prevents leakage of the Cryptographic Material for example by avoiding node exposed to the Internet to host the Cryptographic Material. In addition, LURK provides means to instantaneously suspend the delegation with a suspicious node. On the other hand the LURK Extension 'tls12' introduces some latency, and is not as scalable as STAR or delegated credential solutions.

The LURK Extension 'tls12' is seen as a complementary to the STAR and "delegated credentials". The LURK Extension 'tls12' is a backend solution that does not require any modifications from TLS Client or the CA. It is also aimed at protecting the Cryptographic Material.

LURK may also be deployed within an administrative domain in order to to provide a more controlled deployment of TLS Servers.

2. Terminology and Acronyms

This document re-uses the terminology defined in [I-D.mglt-lurk-lurk].

3. LURK Header

LURK / TLS 1.2 is a LURK Extension that introduces a new designation "tls12". This document assumes that Extension is defined with designation set to "tls12" and version set to 1. The LURK Extension extends the LURKHeader structure defined in [I-D.mglt-lurk-lurk] as follows:

```
   enum {
       tls12 (1), (255)
   } Designation;

   enum {
      capabilities (0), ping (1), rsa_master (2),
      rsa_master_with_poh (3), rsa_extended_master (4),
      rsa_extended_master_with_poh (5), ecdhe (6), (255)
   }TLS12Type;


   enum {
       // generic values reserved or aligned with the
       // LURK Protocol
       request (0), success (1), undefined_error (2),
       invalid_payload_format (3),

       // code points for rsa authentication
       invalid_key_id_type (4), invalid_key_id (5),
       invalid_tls_random (6), invalid_freshness_funct (7),
       invalid_encrypted_premaster (8), invalid_finished (9)

       //code points for ecdhe authentication
       invalid_ec_type (10), invalid_ec_curve (11),
       invalid_poo_prf (12), invalid_poo (13), (255)
   }TLS12Status

   struct {
        Designation designation = "tls12";
        int8 version = 1;
   } Extension;

   struct {
       Extension extension;
       select( Extension ){
           case ("tls12", 1):
               TLS12Type;
       } type;
       select( Extension ){
           case ("tls12", 1):
               TLS12Status;
       } status;
       uint64 id;
       unint32 length;
   } LURKHeader;
```

4. rsa_master, rsa_master_with_poh

An exchange of type "rsa_master" or "rsa_master_with_poh" enables the LURK Client to delegate the RSA Key Exchange and authentication as defined in [RFC5246]. The LURK Server returns the master secret.

"rsa_master" provides the necessary parameters and details to generate the master secret, as well as to hinder replaying of old handshake messages by a corrupt LURK Client. I.e., some attestation of message-freshness is acquired by the LURK Server.

In addition, the"rsa_master_with_poh" provides a proof of handshake (PoH). The proof of handshake consists in providing the Finished message of the TLS Client to the LURK Server, so that latter can perform more checks that in the "rsa_master" mode. Notably, herein, the LURK Server also checks that the LURK request is performed in a context of a TLS handshake.

While "rsa_master" and "rsa_master_with_poh" exchange have respectively different requests, the response is the same. The motivation for having different type is that the parameters provided to the LURK Server are provided using different format. "rsa_master" provides them explicitly, while "rsa_master_with_poh" provides them via handshake messages.

4.1. Request Payload

A rsa_master request payload has the following structure:

```
   enum {
       sha256_32 (0), (255)
   }KeyPairIdType;

   struct {
       KeyPairIdType type;
       opaque data; // length defined by the type
   } KeyPairID;

   enum{
       sha256 (0), (255)
   } FreshnessFunct

   enum{
       sha256 (0), sha384(1), sha512(2), (255)
   } PRFHash

   struct {
       KeyPairID key_id;
       FreshnessFunct freshness_funct;
       PRFHash prf_hash;
       Random client_random;        // see RFC5246 section 7.4.1.2
       Random server_random;
       EncryptedPreMasterSecret  pre_master;
                   // see RFC5246 section 7.4.7.1
                   // Length depends on the key.
       }
   } TLS12RSAMasterRequestPayload;
```

key_id The identifier of the public key. This document defines sha256_32 format which takes the 32 first bits of the hash of the binary ASN.1 DER representation of the public key using sha256. The binary representation of RSA keys is described in [RFC8017]. The binary representation of ECC keys is the subjectPublicKeyInfo structure defined in [RFC5480].

freshness_funct the one-way hash function (OWHF) used by LURK to implement Perfect Forward Secrecy.

prf_hash the one way hash function used by the Pseudo Random Function (PRF) to generate the master secret. PRF and hash function are defined in {!RFC5246}} Section 5.

client_random the random value associated to the TLS Client as defined in [RFC5246] Section 7.4.1.2.

server_random: the random value associated to the TLS Server as defined in [RFC5246] Section 7.4.1.2.

EncryptedPreMasterSecret : The encrypted master secret as defined in [RFC5246] Section 7.4.7.1.

A rsa_master_with_poh request payload has the following structure:

```
struct {
    KeyPairID key_id;
    FreshnessFunct freshness_funct;
    opaque handshake_messages<2...2^16-2>
              // see RFC5246 section 7.4.9
    Finished finished
} TLS12RSAMasterWithPoHRequestPayload;
```

key_id, freshness_funct are defined above

handshake_messages provides the necessary handshake messages to compute the Finished message of the TLS Client as defined in [RFC5246] section 7.4.9.

finished the TLS Client Finished message as defined by {{!RFC5246} section 7.4.9.

4.1.1. Perfect Forward Secrecy

This document defines a mechanism which uses a function called freshness_funct, to prevent an attacker to send a request to the LURK Server in such a way that the said attacker can obtain back the mastersecret for an old handshake. In other words, the use of this function helps prevent a forward-secrecy attack on an old TLS session, where the attack would make use that session's handshake-data observed by the adversary.

This design achieves PFS with freshness_funct being a collision-resistant hash function (CHRF). By CRHF, we mean a one-way hash function (OWHF) which also has collision resistance; the latter means that it is computationally infeasible to find any two inputs x1 and x2 such that freshness_funct(x1) = freshness_funct(x2). By one-way hash function (OWHF) we mean, as standard, a hash function freshness_funct that satisfies preimage resistance and 2nd-preimage resistance. That is, given a hash value y, it is computationally infeasible to find an x such that freshness_funct(x) = y, and respectively- given a value x1 and its hash freshness_funct(x1), it is computationally infeasible to find another x2 such that freshness_funct(x2) = freshness_funct(x1).

For the concrete use of our freshness_funct funtions, let S be a fresh, randomly picked value generated by the LURK Client. The value of server_random in the TLS exchange is then equal to

freshness_funct(S), i.e., server_random=freshness_funct(S). Between the TLS Client and the LURK Server only server-random is exchanged. The LURK Client sends S to the Key Server, in the query. Note that the latter SHOULD happen over a secure channel.

A man-in-the-middle attacker observing the (plaintext) TLS handshake between a TLS Client and the LURK Client does not see S, but only server_random. The preimage resistance guaranted by the freshness_funct makes it such that this man-in-the-middle cannot retrieve S out of the observed server-random. As such, this man-in-the-middle attacker cannot query the S corresponding to an (old) observed handshake to the Key Server. Moreover, the collision resistance guaranteed by the freshness_funct makes it such that if the aforementioned man-in-the-middle cannot find S' such that freshness_funct(S)=freshness_funct(S').

As discussed in Section 9, PFS may be achieved in other ways (i.e., not using a CRHF and the aforementioned exchanges but other cryptographic primitives and other exchanges). These may offer better computational efficiency. These may be standardized in future versions of the LURK extension "tls12.

The server_random MUST follow the structure of [RFC5246] section 7.4.1.2, which carries the gmt_unix_time in the first four bytes. So, the ServerHello.random of the TLS exchange is derived from the server_random of the LURK exchange as defined below:

```
gmt_unix_time = server_random[0..3];
ServerHello.random = freshness_funct( server_random + "tls12 pfs" );
ServerHello.random[0..3] = gmt_unix_time;
```

The operation MUST be performed by the LURK Server as well as the TLS Server, upon receiving the master secret or the signature of the ecdhe_params from the LURK Client.

4.2. Response Payload

The "rsa_master" response payload contains the master secret and has the following structure:

```
struct {
    opaque master[0..47];
} TLS12RSAMasterResponsePayload;
```

4.3. LURK Client Behavior

A LURK Client initiates an rsa_master or an rsa_master_with_poh exchange in order to retrieve the master secret. The LURK exchange happens on the TLS Server side (Edge Server). Upon receipt of the master_secret the Edge Server generates the session keys and finish the TLS key exchange protocol.

A LURK Client MAY use the rsa_master_with_poh to provide the LURK Server evidences that the LURK exchange is performed in the context of a TLS handshake. The Proof of TLS Hanshake (POH) helps the LURK Server to audit the context associated to the query.

The LURK Client MUST ensure that the transmitted values for server_random is S such as server_random = freshness_funct(S).

4.4. LURK Server Behavior

Upon receipt of a rsa_master or a rsa_master_with_poh request, the LURK Server proceeds according to the following steps:

1. The LURK Server checks the RSA key pair is available (key_id). If the format of the key pair identifier is not understood, an "invalid_key_id_type" error is returned. If the designated key pair is not available an "invalid_key_id" error is returned.

2. The LURK Server checks the freshness_funct. If it does not support the FreshnessFunct, an "invalid_freshness_funct" error is returned.

3. The LURK Server collects the client_random, server_random and pre_master parameters either provided explicitly (rsa_master) or within the handshake (rsa_master_with_poh).

4. The LURK Server MUST check the format of the server_random and more specifically checks the gmt_unix_time associated to the random is acceptable. Otherwise it SHOULD return an "invalid_tls_random" error. The value of the time window is implementation dependent and SHOULD be a configurable parameters. The LURK Server MAY also check the client_random. This should be considered cautiously as such check may prevent TLS Clients to set a TLS session. client_random is generated by the TLS Client whose clock might not be synchronized with the one of the LURK Server or that might have a TLS implementations that does not generate random based on gmt_unix_time.

5. The LURK Server computes the necessary ServerHello.random from the server_random when applicable as described in Section 4.1.1.

When option is set to "finished" the ServerHello.random in the handshake is replaced by its new value.

6. The LURK Server checks the length of the encrypted premaster secret and returns an "invalid_payload_format" error if the length differs from the length of binary representation of the RSA modulus.

7. The LURK Server decrypts the encrypted premaster secret as described in [RFC5246] section 7.4.7.1. When a PKCS1.5 format error is detected, or a mismatch between the TLS versions provided as input and the one indicated in the encrypted premaster secret, the Key Server returns a randomly generated master secret.

8. The LURK Server generates the master secret as described in [RFC5246] section 8.1 using the client_random, and the server_random provided by the LURK Client.

9. With a rsa_master_with_poh, the LURK Server checks the Finished message is checked as defined in [RFC5246] section 7.4.9. In case of mismatch returns an "invalid_finished" error.

10. The LURK Server returns a master secret in a TLS12RSAMasterResponsePayload.

11. Error are expected to provide the LURK Client an indication of the cause that resulted in the error. When an error occurs the LURK Server MAY ignore the request, or provide more generic error codes such as "undefined_error" or "invalid_format".

5. rsa_extended_master, rss_extended_master_with_poh

A exchange of type "rsa_extended_master" enables the LURK Client to delegate the RSA Key Exchange and authentication. The LURK Server returns the extended master secret as defined in [RFC7627].

5.1. Request Payload

The "rsa_extended_master" request has the following structure:

```
   enum { sha256 (0), (255) } FreshnessFunct

   enum { null(0), sha256_128(1), sha256_256(2),
   (255) }POOPRF

   struct {
       KeyPairID key_id
       FreshnessFunct freshness_funct        // see RFC5246 section 6.1
       opaque handshake_messages<2...2^16-2> // see RFC7627 section 4
   }TLS12ExtendedMasterRSARequestPayload;
```

The "rsa_extended_master_with_poh" request has the following structure:

```
   struct {
       KeyPairID key_id
       FreshnessFunct freshness_funct        // see RFC5246 section 6.1
       opaque handshake_messages<2...2^16-2>
                                   // see RFC5246 section 7.4.9
       Finished finished
       }
   }TLS12ExtendedMasterRSAWithPoHRequestPayload;
```

key_id, freshness_funct, option, handshake, finished are defined in Section 4.1.

handshake_messages With a the handshake message includes are those necessary to generate a extended master secret as defined in [RFC7627] section 4.

5.2. Response Payload

rsa_extended_master response payload has a similar structure as the rsa_master response payload Section 4.2.

5.3. LURK Client Behavior

The LURK Client proceeds as described in {{sec-rsa-master-clt}}. The main difference is that the necessary element to generate the master secret are included in the handshake and or not provided separately.

5.4. LURK Server Behavior

The LURK Server proceeds as described in Section 4.4 except that the generation of the extended master is processed as described in [RFC7627].

6. ecdhe"

A exchange of type "ecdhe" enables the LURK Client to delegate the ECDHE_RSA [RFC5246] or the ECDHE_ECDSA [RFC8422] authentication.

6.1. Request Payload

The "ecdhe" request payload has the following structure:

```
enum { null(0), sha256_128(1), sha256_256(2),
(255) }POOPRF

struct {
    POOPRF poo_prf;
    select( poo_prf ) {
        case ( "null" ):
        case ( "sha256_128" )
            ECPoint vG;  //RFC8422 section 5.4
            opaque R[16] r;
        case ( "sha256_256" ):
            ECPoint vG;  //RFC8422 section 5.4
            opaque R[32] r;
    }
} TLS12POOParams;

struct {
    KeyPairID key_id;
    FreshnessFunct freshness_funct;
    Random client_random;        // see RFC5246 section 7.4.1.2
    Random server_random;
    SignatureAndHashAlgorithm sig_and_hash  //RFC 5246 section 4.7
    ServerECDHParams ecdhe_params;  // RFC8422 section 5.4
    POOParams poo_params;
} TLS12ECDHERequestPayload;
```

key_id, freshness_funct, client_random, server_random is defined in
   Section 4.1.

ecdhe_params contains as defined in [RFC8422] section 5.4, the
   elliptic curve domain parameters associated with the ECDH public
   key (defined by the ECParameters structure) and the ephemeral ECDH
   public key (defined by the ECPoint structure). The public key is
   also noted in this document bG with b is a random secret generated
   by the LURK Client and G the base point of the curve.

poo_params defines the necessary parameters to provide a proof of
   ownership of the ECDHE private key. This option is intended to

prevent the LURK Server to sign bytes that do not correspond to a ECDHE public key.

poo_prf pseudo random function used to generate the necessary randoms to proof ownership of the private key. This document defines sha256_128 and sha256_256 which apply the sha256 hash function and respectively return the 128 or 256 first bits of the resulting hash.

vG are the necessary points to generate the proof of ownership.

r necessary value to create the proof of ownership.

The proof of ownership (PoO) consists in the LURK Client proving the knowledge of the private random b, while not disclosing b. With G the base point, bG represents the public value. The PoO is based on the non-interactive variant of the three-pass Schnorr identification scheme (NIZR) also designated as the Fiat-Shamir transformation described in [RFC8235]. More specifically, the LURK Client randomly generates v and then derive c and r = v - b*c. The LURK Client provides bG, vG, and r to the LURK Servers. The LURK Server first checks bG is on the curve. Then it computes c similarly to the LURK Client as well S = rG + (bG)c. This latest value S is compared to vG. The equality between S and vG proves the ownership of b.

v is randomly generated by the LURK Client. v MUST remain non-predictable with a length equivalent to the expected level of security, that is 128 bit length (resp. 256 bit length) for a 128 (resp 256) bit security level. Given b, we RECOMMEND v to be at least half the size of b.

c is computed by the LURK Client and the LURK Server as described in [RFC8235]. UserID is defined by the concatenation of the client_random and the server_random. OtherInfo is defined as the concatenation of key_id, freshness_funct, sig_and_hash, ecdhe_params, "tls12 poo". Each concatenated item is prefixed with a 4-byte integer that represents the byte length of the item.

```
UserID = client_random || server_random
OtherInfo =  key_id || freshness_funct || sig_and_hash ||
             ecdhe_params || "tls12 poo"
c = poo_prf(G || vG || bG || UserID || OtherInfo)
```

The LURK Client provides bG in ecdhe_params and vG as well as r in poo_params.

With X25519 or X448, b and r MUST be clamped and vG MUST use the Curve25519 (resp. Curve448). bG MAY also use the Curve25519 or

Curve448 representation, or the LURK Server MAY derive bG values from the provided xlined value in ecdhe_params.

6.2. Response Payload

The "ecdhe" response payload has the following structure:

```
struct {
    Signature signed_params;  // RFC8422 section 5.4
} TLS12ECDHEResponsePayload;
```

signed_params signature applied to the hash of the ecdhe_params as well as client_random and server_random as described in [RFC8422] section 5.4.

6.3. LURK Client Behavior

The LURK Client builds the base as described in Section 4.1 and in Section 6.1.

Upon receiving the response payload, the LURK Client MAY check the signature. If the signature does not match an error SHOULD be reported.

6.4. LURK Server Behavior

Upon receiving an ecdhe request, the LURK Server proceeds as follows:

1. perform steps 1 - 6 as described in Section 4.4

2. The LURK Server performs some format check of the ecdhe_params before signing them. If the ecdhe_params does not follow the expected structure. With the notations from [RFC8422], if curve_type is not set to "named_curve", the LURK Server SHOULD respond with an "invalid_ec_type" error. If the curve or namedcurve is not supported the LURK Server SHOULD be able to respond with an "invalid_ec_curve" error.

3. The LURK Server processes the poo_params. If the poo_prf is not supported, the LURK Extension returns a "invalid_poo_prf" status. If poo_prf is supported and different from "null", the LURK Server proceeds to the proof of ownership as described in Section 6.1. If the proof is not properly verified, the LURK Extension returns a "invalid_poo" status.

4. The LURK Server processes the base structure as described in Section 4.4

5. The LURK Server generates the signed_params.

Error are expected to provide the LURK Client an indication of the cause that resulted in the error. When an error occurs the LURK Server MAY ignore the request, or provide more generic error codes such as "undefined_error" or "invalid_format".

7. capabilities

A exchange of type "capabilities" enables the LURK Client to be informed of the supported operations performed by the LURK Server. The supported parameters are provided on a per type basis.

7.1. Request Payload

A LURK "capabilities" request has no payload.

7.2. Response Payload

The "capabilities" response payload lists for each supported type, the supported certificates, the supported signatures and hash associated. The "capabilities" payload has the following structure:

```
struct{
    CertificateType certificate_type  // RFC8442 section 4.4.2
    select (certificate_type) {
         case RawPublicKey:
           /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
           opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
         case X509:
           opaque cert_data<1..2^24-1>;
     };
} TypedCertificate;

struct {
     KeyPairID key_id_type_list<0..255>;
     TypedCertificate typed_certificate_list<0..255>
     FreshnessFunctList freshness_funct_list<0..255>
     CipherSuites cipher_suite_list<0..255>
     PRFHash prf_hash_list<0..255>
} TLS12RSACapability;



struct {
     KeyPairID key_id_type_list<0..255>;
     TypedCertificate typed_certificate_list<0..255>
     FreshnessFunctList freshness_funct_list<0..255>
```

```
        CipherSuites cipher_suite_list<0..255>
        SignatureAndHashAlgorithm sig_and_hash_list<0..255>
        NameCurve ecdsa_curves_list<0..255>;
        NameCurve ecdhe_curves_list<0..255>
        POOPRF poo_prf_list<0..255>
   } TLS12ECDHECapability;

   struct {
       uint32 length;
       TLS12Type type
       Select( type ) {
              case rsa_master : TLS12RSACapability,
              case rsa_master_with_poh : TLS12RSACapability,
              case rsa_extended_master : TLS12RSACapability,
              case rsa_extended_master_with_poh : TLS12RSACapability,
              case ecdhe : TLS12ECDHECapability
       } capability ;
   } TLS12Capability


   struct {
       TLS12Capability capability_list;
       opaque state<32>;
   } TLS12CapabilitiesResponsePayload;
```

typed_certificate enables to contain authentication credentials of various type, such as X09 certificate or raw public key. While different, the structure is similar of CertificateEntry defined in [RFC8446] section 4.4.2 as well as the Certificate structure defined in [RFC7250].

key_id_type_list the supported key_id_type.

freshness_funct_list designates the list of freshness_funct (see Section 4.1).

certificate_list designates the certificates associated to message type. The format is similar but different from the CertificateEntry defined in [RFC8446] in section 4.4.2 and [RFC7250] section 1. The CertificateBis format enables the use of X509 as well as Raw Public key, while the Certificate structure defined in [RFC5246] section 7.4.2 does not.

sig_and_hash_list designates supported signature algorithms as well as PRF used for the different operations. The format is defined in [RFC5246] section 7.4.1.4.1.

ecdsa_curves_list the supported signatures

ecdhe_curves_list the supported curves for ECHDE parameters.

poo_prf_list the supported message type poo_prf (see Section 6.1. to be used with the proof of ownership.

type_list the supported message type of the LURK extension.

state characterizes the configuration associated to 'tls12' on the LURK Server..

7.3. LURK Client Behavior

The LURK Client performs a capability request in order to determine the possible operations.

The LURK Client is expected to keep the state value to be able to detect a change in the LURK Server configuration when an error occurs.

7.4. LURK Server Behavior"

Upon receiving a capabilities request, the LURK Extension MUST return the capabilities payload associated to a "success" status to the LURK Server. These information are then forwarded by the LURK Server to the LURK Client.

8. ping

A exchange of type "ping" enables the LURK Client to check the reachability in a context of the defined LURK Extension.

8.1. Request Payload

A "ping" request has no payload.

8.2. Response Payload

A "ping" response has no payload.

8.3. LURK Client Behavior

The LURK Client sends a "ping" request to test the reachability of the LURK Server. The reachability is performed for the tls12 LURK Extension.

8.4. LURK Server Behavior

Upon receiving a ping request, the LURK Extension MUST return the ping response associated with a "success" status to the LURK Server. These information are then forwarded by the LURK Server to the LURK Client.

9. Security Considerations

The security considerations defined in [I-D.mglt-lurk-lurk] applies to the LURK Extension "tls12" defined in this document.

Anti-replay mechanisms rely in part on the security of channel between the LURK Client and the LURK Server. As such the channel between the LURK Client and the LURK Server MUST be ensuring confidentiality and integrity. More specifically, the exchanges between the LURK Client and the LURK Server MUST be an encrypted with authentication encryption, and the two parties had previously mutually authenticated.

The LURK Extension "tls12" is expected to have response smaller that the request or at least not significantly larger, which makes "tls12" relatively robust to amplification attacks. This is especially matters when LURK is using UDP. The use of an authenticated channel reduces also the risk of amplification attacks even when UDP is being used.

The LURK Client and the LURK Server use time in their way to generate the server_random. Care MUST be taken so the LURK Client and LURK Server remain synchronized.

9.1. RSA

The rsa_master and rsa_extended_master returns the master_secret instead of the premaster. The additional hashing operation necessary to generate the master secret is expected to improve the protection of the RSA private key against cryptographic analysis based on the observation of a set of clear text and corresponding encrypted text.

The standard TLS1.2 is robust against Bleichenbacher attack as it provides no means to detect if the error comes from a TLS version mismatch or from the premaster format. This properties remain with LURK, and so LURK does not present vulnerabilities toward Bleichenbacher attack, and cannot be used as a decryption oracle.

9.2. ECDHE

A passive attacker observing the ecdhe exchange may collect a sufficient amount of clear text and corresponding signature to perform a cryptographic analysis or to reuse the signature for other purposes. As a result, it remains important to encrypt the ecdhe exchange between the LURK Client and the LURK Server. Note that this vulnerability is present in TLS 1.2 as a TLS Client can accumulate these data as well. The difference with LURK is by listening the LURK Server, the accumulation is achieved for all TLS Clients.

As previously mentioned, the LURK Server may be used as signing oracle for the specific string:

```
    SHA(ClientHello.random + ServerHello.random +
                         ServerKeyExchange.params);
```

More specifically, the ECDHE_RSA and ECDHE_DSA mechanisms does not associate the signature to a TLS1.2 context. As a result, an attacker could re-used the signature in another context.

The attack may operate by collecting a large collection of clear text and their corresponding signature. When the attacker want to provide a signature, it checks in its database, a match occurs between the two contents to be signed. The probability of a collision increases with number of available hashes. The attack is related the pre-image and collision resistance properties of the hash function.

The attacker may also given a clear text to be signed, generate a collision such that a collision occurs which provides is related to the second pre-image and collision resistance property of the hash function.

The surface of attack is limited by:

- limiting the possibility of aggregating a collection of clear text and their corresponding signatures. This could be achieved by using multiple LURK Clients using an encrypted channel between the LURK Client and the LURK Server.

- increasing the checks and ensure that signature is performed in a TLS 1.2 context. For that purpose it is RECOMMENDED the LURK Server checks the consistency of its input parameters. This includes the proof of ownership as well as the format of the randoms and ecdhe_params for example.

- limiting the usage of a Cryptographic material to a single usage, in our case serving TLS 1.2.

9.3. Perfect Foward Secrecy

This document uses sha256 as the freshness_funct, in order to achieve PFS Section 4.1.1 as described above. By construction of the server_random, of the output of freshness_funct we will keep only the last 28 bytes. The PFS property is in place as long as this truncated version of freshness_funct can be considered a CRHF and that the 28 bytes of randomness carried by the server_random are sufficient. Otherwise, the mechanism described in this document will not be considered as safe.

Details on the truncation will be added. Alternatively, we could use a hash function like SHA3 (or, more explicitly SHAKE) which considers variable output length as part of its design. The SHAKE functions allow arbitrary output lengths and the PFS-input S can be of arbitrary length too. However, for SHAKE128-d, if the truncated output is of length d as low as 224 bits (28 bytes), then one only gets 224/2=112 bits security w.r.t. collision-resistance, > 112 bits w.r.t. preimage resistance and 112 bits security w.r.t. second preimage resistance.

One reason why we have the hash-based solution to is to reduce communication costs between the LURK Client and the LURK Server, whilst still getting more than some security w.r.t. a MiM corrupting a LURK Client and then attempting a PFS attack.

But, if we disregard the overhaed on communication costs, we can consider other mechanisms not based on CRHF for attaining PFS security. See I and II below.

I. For example, as freshness_funct, one can use an instance of a pseudo random function (PRF), keyed on a key K that the LURK Server already shares with the LURK Client. I.e., server_random=freshness_funct(S;K). In this case, the mechanisms to achieve PFS are as follows: 1. The LURK Client and the LURK Server run a key-establishment protocol before every LURK session to establish such a new key K for every LURK session. Alternatively, the export this key of the key-establishment run to secure the channel. The time-to-live of K is one session only. 2. The LURK Server generates the value S on its side and send the server_random to the LURK Client. 3. The LURK Client uses this server_random with the TLS Client 4. The LURK Server checks the correctness of the use of the said server_random when the query for the master_secret is made, with the messages forwarded therein;

II. In fact, since the channel between the LURK Client and the LURK Server MUST be encrypted by default, all for 2 steps in point I above can be combined into 1 step (without the need of a specially executed

key-establishment): a. the LURK Server sends the server_random to the LURK Client. b. the LURK Client uses this server_random with the TLS Client c. the LURK Server checks the correctness of the use of the said server_random when the query for the master_secret is made, with the messages forwarded therein;

Yet, option I and option II are more expensive on the communication than the version achieving PFS with a hash function. I.e., in I and II, the LURK Server needs to be involved on the first part of the TLS handshake to produce the S or server_random for the LURK Client. However, note that the LURK Client no longer queries S, hence the risk of a man-in-the-middle querying an old S is eliminated by design.

Option II above is akin to what "Content delivery over TLS: a cryptographic analysis of keyless SSL," by K. Bhargavan, I. Boureanu, P. A. Fouque, C. Onete and B. Richard at 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, 2017, pp. 1-16, suggested in order to amend (forward-secrecy) attacks on Keyless SSL.

10. IANA Considerations

The requested information is defined in [I-D.mglt-lurk-lurk].

LURK Extension Designation: tls12 LURK Extension Reference: [RFD-TBD] LURK Extension Description: RSA, ECDHE_RSA and ECDHE_ECDSA for (D)TLS 1.2.

LURK tls12 Extension Status

Value	Description	Reference
0 - 1	Reserved	[RFC-TBD-LURK]
2	undefined_error	[RFC-TBD]
3	invalid_payload_format	[RFC-TBD]
4	invalid_key_id_type	[RFC-TBD]
5	invalid_key_id	[RFC-TBD]
6	invalid_tls_random	[RFC-TBD]
7	invalid_freshness_funct	[RFC-TBD]
8	invalid_encrypted_premaster	[RFC-TBD]
9	invalid_finished	[RFC-TBD]
10	invalid_ec_type	[RFC-TBD]
11	invalid_ec_curve	[RFC-TBD]
12	invalid_poo_prf	[RFC-TBD]
13	invalid_poo	[RFC-TBD]
14	invalid_cipher_or_prf_hash	[RFC-TBD]
15 - 255	UNASSIGNED	

LURK tls12 Extension Type

Value	Description	Reference
0	capabilities	[RFC-TBD]
1	ping	[RFC-TBD]
2	rsa_master	[RFC-TBD]
2	rsa_master_with_poh	[RFC-TBD]
3	rsa_extended_master	[RFC-TBD]
3	rsa_extended_master_with_poh	[RFC-TBD]
4	ecdhe	[RFC-TBD]
16 - 255	UNASSIGNED	

11. Acknowledgments

We would like to thank for their very useful feed backs: Yaron Sheffer, Yoav Nir, Stephen Farrell, Eric Burger, Thomas Fossati, Eric Rescorla, Mat Naslung, Rich Salz, Ilari Liusvaara, Scott Fluhrer. Many ideas in this document are from [I-D.erb-lurk-rsalg].

We would also like to thank those that have supported LURK or raised interesting discussions. This includes among others Robert Skog, Hans Spaak, Salvatore Loreto, John Mattsson, Alexei Tumarkin, Richard Brunner, Stephane Dault, Dan Kahn Gillmor, Joe Hildebrand, Kelsey Cairns.

12. Apendix

```
   ## LURK Exchange for TLS RSA Master Secret

   TLS Client          Edge Server         Key Server

   ClientHello
      server_version
      client_random
      cipher_suite
          TLS_RSA_*, ...
   -------->
                       S = server_random
                       server_random = freshness_funct( S )

                       ServerHello
                           tls_version
                           server_random
                           Cipher_suite=TLS_RSA
                       Certificate
                           RSA Public Key
                       ServerHelloDone
                       <--------

   ClientKeyExchange
       EncryptedPremasterSecret
   [ChangeCipherSpec]
   Finished
   -------->


                       TLS12 Request Header
                       TLS12MasterRSARequestPayload
                           key_id
                           freshness_funct
                           prf_hash
                           client_random
                           S
                           EncryptedPremasterSecret
                       -------->

                                    server_random = freshness_funct( S )

                                    master_secret = PRF(\
                                    pre_master_secret + \
                                    "master secret" +\
                                    client_random +\
                                    server_random)[0..47];
```

```
                                        TLS12 Response Header
                                        TLS12MasterResponsePayload
                                            master
                                        <--------

                        [ChangeCipherSpec]
                            Finished
                        <--------
   Application Data       <------->     Application Data
```

12.1. LURK Exchange for TLS RSA Master Secret with Proof of Handshake

```
   TLS Client          Edge Server         Key Server

   ClientHello
      server_version
      client_random
      cipher_suite
          TLS_RSA_*, ...
   -------->
                        S = server_random
                        server_random = freshness_funct( S )

                        ServerHello
                            tls_version
                            server_random
                            Cipher_suite=TLS_RSA
                        Certificate
                            RSA Public Key
                        ServerHelloDone
                        <--------

   ClientKeyExchange
       EncryptedPremasterSecret
   [ChangeCipherSpec]
   Finished
   -------->


                        TLS12 Request Header
                        TLS12MasterRSAWithPoHRequestPayload
                            key_id
                            freshness_funct
                            handshake_messages
                            finished
                        -------->

                                     server_random = freshness_funct( S )
```

```
                                    master_secret = PRF(\
                                    pre_master_secret + \
                                    "master secret" +\
                                    client_random +\
                                    server_random)[0..47];

                                           TLS12 Response Header
                                           TLS12MasterResponsePayload
                                               master
                                           <--------

                       [ChangeCipherSpec]
                           Finished
                       <--------
   Application Data      <------->     Application Data
```

12.2. LURK Exchange for TLS RSA Extended Master Secret

```
   TLS Client          Edge Server         Key Server

   ClientHello
      tls_version
      cipher_suite
          TLS_RSA_*, ...
      Extension 0x0017
   -------->

                       ServerHello
                           edge_server_version
                           cipher_suite=TLS_RSA
                           Extension 0x0017
                       Certificate
                           RSA Public Key
                       ServerHelloDone
                       <--------
   ClientKeyExchange
       EncryptedPremasterSecret
   [ChangeCipherSpec]
   Finished
   -------->

                       TLS12 Request Header
                       TLS12ExtendedMasterRSARequestPayload
                           key_id
                           freshness_funct
                           handshake_messages
                           EncryptedPreMasterSecret
                       -------->

                                    1. Computing Master Secret
                                    master_secret = master_prf(
                                    pre_master_secret +\
                                    "extended master secret" +\
                                    session_hash)[0..47]

                                           TLS12 Response Header
                                           TLS12MasterPayload
                                               master
                                           <--------

                       [ChangeCipherSpec]
                           Finished
                       <--------
   Application Data      <------->     Application Data
```

12.3. LURK Exchange for TLS RSA Extended Master Secret with proof of handshake

```
   TLS Client          Edge Server         Key Server

   ClientHello
      tls_version
      cipher_suite
          TLS_RSA_*, ...
      Extension 0x0017
   -------->

                       ServerHello
                           edge_server_version
                           cipher_suite=TLS_RSA
                           Extension 0x0017
                       Certificate
                           RSA Public Key
                       ServerHelloDone
                       <--------
   ClientKeyExchange
       EncryptedPremasterSecret
   [ChangeCipherSpec]
   Finished
   -------->

                       TLS12 Request Header
                       TLS12ExtendedMasterWithPoHRequestPayload
                           key_id
                           freshness_funct
                           handshake_messages
                           finished
                       -------->

                                    1. Computing Master Secret
                                    master_secret = master_prf(
                                    pre_master_secret +\
                                    "extended master secret" +\
                                    session_hash)[0..47]

                                           TLS12 Response Header
                                           TLS12MasterPayload
                                               master
                                           <--------

                       [ChangeCipherSpec]
                           Finished
                       <--------
   Application Data      <------->     Application Data
```

12.4. LURK Exchange for TLS ECDHE Signature

```
   TLS Client          Edge Server         Key Server

   ClientHello
      tls_version
      client_random
      cipher_suite
          TLS_ECDHE_ECDSA_*, TLS_ECDHE_RSA_*, ...
          Extension Supported EC, Supported Point Format
   -------->
                       S = server_random
                       server_random = freshness_funct( S )

                       TLS12 Request Header
                       TLS12ECDHEInputPayload
                           key_id
                           client_random
                           S
                           ecdhe_params
                       -------->
                                    server_random = freshness_funct( S )

                                    signature = ECDSA( client_random +\
                                    server_random + ecdhe_params )

                                           TLS12 Response Header
                                           TLS12DigitallySignedPayloads
                                               signature
                                           <--------

                       ServerHello
                           tls_version
                           server_random
                           Cipher_suite=TLS_ECDHE_ECDSA
                           Extension Supported EC,
                           Supported Point Format
                       Certificate
                           ECDSA Public Key
                       ServerKeyExchange
                           ecdhe_params
                           signature
                       ServerHelloDone
                       <--------


   ClientKeyExchange
   [ChangeCipherSpec]
```

```
   Finished
   -------->
                       [ChangeCipherSpec]
                       Finished
                       <--------
   Application Data      <------->     Application Data
```

13. References

13.1. Normative References

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <https://www.rfc-editor.org/info/rfc7627>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

13.2. Informative References

[I-D.erb-lurk-rsalg]
Erb, S. and R. Salz, "A PFS-preserving protocol for LURK", draft-erb-lurk-rsalg-01 (work in progress), May 2016.

[I-D.fieau-cdni-https-delegation]
Fieau, F., Emile, S., and S. Mishra, "HTTPS delegation in CDNI", draft-fieau-cdni-https-delegation-02 (work in progress), July 2017.

[I-D.ietf-acme-acme]
Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", draft-ietf-acme-acme-18 (work in progress), December 2018.

[I-D.ietf-acme-star]
Sheffer, Y., Lopez, D., Dios, O., Pastor, A., and T. Fossati, "Support for Short-Term, Automatically-Renewed (STAR) Certificates in Automated Certificate Management Environment (ACME)", draft-ietf-acme-star-11 (work in progress), October 2019.

[I-D.mglt-lurk-lurk]
Migault, D., "LURK Protocol version 1", draft-mglt-lurk-lurk-00 (work in progress), February 2018.

[I-D.mglt-lurk-tls-use-cases]
Migault, D., Ma, K., Salz, R., Mishra, S., and O. Dios, "LURK TLS/DTLS Use Cases", draft-mglt-lurk-tls-use-cases-02 (work in progress), June 2016.

[I-D.rescorla-tls-subcerts]
Barnes, R., Iyengar, S., Sullivan, N., and E. Rescorla, "Delegated Credentials for TLS", draft-rescorla-tls-subcerts-02 (work in progress), October 2017.

[I-D.sheffer-acme-star-request]
Sheffer, Y., Lopez, D., Dios, O., Pastor, A., and T. Fossati, "Generating Certificate Requests for Short-Term, Automatically-Renewed (STAR) Certificates", draft-sheffer-acme-star-request-02 (work in progress), June 2018.

[RFC8235] Hao, F., Ed., "Schnorr Non-interactive Zero-Knowledge Proof", RFC 8235, DOI 10.17487/RFC8235, September 2017, <https://www.rfc-editor.org/info/rfc8235>.

Authors' Addresses

Daniel Migault
Ericsson
8275 Trans Canada Route
Saint Laurent, QC 4S 0B6
Canada

EMail: daniel.migault@ericsson.com

Ioana Boureanu
University of Surrey
Stag Hill Campus
Guildford GU2 7XH
UK

EMail: i.boureanu@surrey.ac.uk