Network Working Group S. Erb
Internet-Draft R. Salz
Intended status: Standards Track Akamai Technologies
Expires: November 29, 2016 May 28, 2016

A PFS-preserving protocol for LURK
draft-erb-lurk-rsalg-01

Abstract

This document defines a protocol between a content provider and an external key owner that enables the provider to act as a TLS termination end-point for the key owner, without having the key actually being provisioned at the provider.

The protocol between the two preserves forward secrecy, and is also designed to prevent the use of the key owner as a general-purpose signing oracle which would make it complicit in attacks against uses of the very keys it is trying to protect.

Table of Contents

1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Three entities are involved in this protocol, although only two actually

participate in the protocol exchanges:

```
      Client    <----->    Server    <---->    KeyOwner
```

The "KeyOwner" is an entity holding a Certificate and associated private Key, typically bound to an identity such as a DNS name.

The "server" acts on behalf of the KeyOwner, such as terminating TLS connections. From external appearances, such as TLS peer name verification, the server is indistinguishable from the KeyOwner.

The "client" is the end-entity that initiates a connection to the server.

2. Goals and Non-Goals

It is not a goal to protect against an active attacker who can decrypt or actively MiTM any of the traffic.

It is not a goal to protect Client-Server traffic in the event of a full compromise of a KeyOwner private key.

This protocol can support Client-Server communications from SSLv3 up through TLS 1.2. (TLS 1.3 will have to be evaluated at a later date.)

Past Client-Server communications must remain private in the event that a Server is compromised (Perfect Forward Secrecy). For Server Key Exchange signing requests, this is not an issue. For RSA decryption requests used by the TLS_RSA_* cipher suites, the "RSALG" message exchanges described below provide PFS protection.

The protocol should not become a generic signing oracle, even if it is suboptimal with regard to network bandwidth utilization. This is done by not simply signing values, but by computing the full signature hash at the KeyOwner.

3. Protocol Overview

Communication between the Server and KeyOwner MUST be over a mutually-authenticated TLS connection that uses PFS key exchange. TLS 1.2 or later SHOULD be used.

3.1. Setup

A Server can contact a KeyOwner at any time to request the state of the KeyOwner. When a Server is notified of a state change in a KeyOwner response message, it MUST then request the state of the KeyOwner.

3.2. Server Key Exchange

A KeyOwner will sign requests on behalf of the Server for the signature required for the Server Key Exchange Message. This message includes the client and server random values and key parameters.

3.3. RSALG

The basic premise of RSALG is that in the TLS_RSA_* handshakes:

- o The KeyOwner will not decrypt the PMS and provide it back to the Server. Instead, the KeyOwner will full compute the Master Secret (via the PRF function) and provide that.

- o The Server will choose a random ephemeral value, N, and provide a cryptographically-hashed value of (such as SHA256(N)) as its Server Random value. The Server sends N to KeyOwner which then computes the same hashed value and uses that hash as its input to the PRF.

An attacker who later gains access to KeyOwner would be unable to derive the same Master Secret. This attacker would be able to see the Client Random, Server Random and encrypted PMS, but would be unable to replay this to KeyOwner unless they could reverse the cryptographic hash function used to compute the server random.

3.3.1. Implementation Note - Modified Bleichenbacher Attack

If an attacker can gain access to a Server, they could mount a Bleichenbacher attack against it (REF NEEDED). The standard SSL/TLS defense against the Bleichenbacher attack (generating a string of random bytes) is not effective here, since an attacker could generate two requests with identical inputs and learn information about the validity of the padding by seeing whether it gets a consistent output in both cases. This is possible because the attacker also controls (the input to) the server random.

To avoid this variation on the Bleichenbacher attack, KeyOwner should compute the HMAC-SHA-384 over the PRF inputs as its "invalid" response, using a private key as the hash key, to ensure that the output is a deterministic function of the input and cannot be calculated by the attacker. This private key must be globally unique per keypair, therefore the RSA private key being used to decrypt the PMS is an obvious choice.

The PRF inputs to the HMAC-SHA-384 described above are the encrypted PMS, client version and server version.

3.3.2. Implementation Note - Hash Calculation

In TLS 1.2 and earlier, the first four bytes of a server random value are actually a timestamp. An implementation must use those four bytes as an input to the hash function as described above, then

overwrite them as input to the PRF calculated by the KeyOwner and the Server Random value provided to the Client.

Example:

```
server_random = N
server_random[0..3] = get_time()
```

Server communicates server_random to KeyOwner

Both Server and KeyOwner compute the following:

```
saved_time = server_random[0..3]
server_random = sha256(server_random)
server_random[0..3] = saved_time
```

3.4. Session Ticket Key Request

A Server that supports TLS session tickets for multiple KeyOwners SHOULD ensure that the ticket encryption keys are secure in the face of various compromises. Using a hash of the private key as one of the inputs to the session ticket KDF ensures that the traffic for KeyOwner is protected against compromise of, or malicious behavior by, other input parts to the session ticket KDF. It also limits the extent to which compromise of a particular session ticket key effects the Server acting on behalf of multiple KeyOwners.

After receiving a request, the KeyOwner computes an HMAC over a server-supplied salt and a fixed string using the private key for the certificate specified in the request as the hash key.

The fixed string is set by the KeyOwner, for example "LURK SESSION TICKET".

```
session_ticket_secret = HMAC-SHA-384(private_key,
                                     server_salt + fixed_string)
```

4. LURK Message Formats

The formats below are described using the TLS Presentation Language.

The following message header appears at the start of every message:

```
    enum {
        one(1), (255)
    } Version
    enum {
        setup_request(0), setup_response(1),
        request(2), session_ticket_request(3), response(4), (255)
    } Type
    struct {
        Version  version;
        Type  type;
        uint16 length;
    } lurk_msg_header;
```

version The version of this protocol.

type The message type. Details defined below.

length Length of the entire message, including header, in bytes.

4.1. Setup Response Message

A setup request message, requesting the state of the KeyOwner looks like this:

```
    struct {
        lurk_msg_header  header;
        uint64           id;
    } setup_request;
```

id A unique identifier to allow pipelining and match requests and responses.

4.2. Setup Response Message

A setup response message, returning the state of the KeyOwner looks like this:

```
struct {
    uint8  purpose<32>;
    opaque ASN.1Cert<1..2^24-1>;
} certificate;
struct {
    lurk_msg_header  header;
    uint64           id;
    SignatureAndHashAlgorithm
                     supported_signature_algorithms<2..2^16-2>;
    certificate      certificate_list<0..2^24-1>;
    uint8            state<32>;
} setup_response;
```

id A unique identifier to allow pipelining and match requests and responses.

supported_signature_algorithms A list of supported signature hash algorithms that the KeyOwner supports (see RFC5246, section 7.4.1.4.1). TODO: TLSv1.3 considerations

certificate_list A list of certificate that are supported by the KeyOwner. The purpose field is a value that MUST be pre-configured by the Server and KeyOwner so a Server can have context of where to use the corresponding ASN.1Cert. An example pre-configuration of the purpose field is: purpose = sha256(hostname)

state A hash of the current state of the server. A KeyOwner MUST provide this value in every response message and MUST update the value to let a Server know to send a setup_request message. This value MUST be consistant across multiple KeyOwners with identical configurations. An example of this value: state = sha256(supported_signature_algorithms + certificate_list)

4.3. Request Message

A request message looks like this:

```
enum {
    rsalg(0), server_kx(1), (255)
} ReqType
struct {
    lurk_msg_header  header;
    uint64           id;
    ReqType          op_type;
    uint8            cert<32>;
    uint16           client_version;
    uint16           server_version;
    uint8            client_random<32>;
    uint8            server_random<32>;
    SignatureAndHashAlgorithm sig_hash_alg;
    PRFHashAlgorithm          prf_hash_alg;
    opaque           data<0..2^16-1>;
} lurk_request;
```

id A unique identifier to allow pipelining and match requests and responses.

cert The identifier for the keypair to be used in this request. This SHOULD be the SHA256 value of the public key.

client_version The TLS Version Number provided by the Client in the clientHello message. Note that for RSALG requests, the value must be verified (see RFC5264, section 7.4.7.1)

server_version The TLS Version Number provided by the Server in the serverHello message. Note that for RSALG requests, the value must be verified (see RFC5264, section 7.4.7.1)

client_random The TLS Client Random provided by the clientHello message.

server_random The TLS Server Random provided by the serverHello message. Note that for RSALG requests, this is actually the digested value of N.

sig_hash_alg For server_kx requests, this is the signature hash value that the Server will use (see RFC5246, section 7.4.1.4.1). For rsalg requests, this field is ignored and SHOULD be NULL. TODO - TLSv1.3 considerations.

prf_hash_alg For rsalg requests, this identifies the PRF function to use. For server_kx requests, this field is ignored and SHOULD be NULL.

TODO: this likely should follow the same format as the first byte of sighashalgo above, also need md5/sha1 combo value here.

data For rsalg requests, this contains the encrypted PRF. For server_kx signing requests, this contains the key parameters to sign.

4.4. Session Ticket Request

A session ticket key input request message looks like this:

```
struct {
    lurk_msg_header  header;
    uint64           id;
    uint8            cert<32>;
    uint8            server_salt<48>;
} lurk_session_ticket_request;
```

id A unique identifier to allow pipelining and match requests and responses.

cert The identifier for the keypair to be used in this request. This SHOULD be the SHA256 value of the public key.

server_salt A server supplied random salt.

4.5. Response Message

A response message, used by both request types, looks like this:

```
enum {
    success(0), invalidParameters(1), certUnavailable(2),
    permissionDenied(3), insufficentResources(4), (255)
} ResponseStatus
struct {
    lurk_msg_header  header;
    ResponseStatus   status;
    uint64           id;
    uint8            state<32>;
    opaque           data<0..2^16-1>;
} lurk_response;
```

id The request id for which this is the response.

state A 32 byte tag identifying the current state of the server. This is expected to be the same value found in the setup_response message. If this value is different the Server MUST send a setup_request message.

data For any status other than success, the data is ignored and MUST be NULL. For rsalg requests, the data contains the master secret. For server_kx requests, the data contains the signed hash. For session ticket key requests, the data contains the computed HMAC.

5. Open Issues

The KeyOwner could choose the TLS server random. This makes RSALG even less likely to be useful as an oracle, but has turned out to be difficult to integrate into existing TLS/SSL libraries.

Should the lurk_request and lurk_response messages be padded out to eight-byte alignment?

Should we use variant for the different request/response payloads?

6. Acknowledgements

We acknowledge the cooperation of Charlie Gero and Phil Lisiecki of Akamai Technologies, and their disclosure of US Patent Application 20150106624, "Providing forward secrecy in a terminating TLS connection proxy."

7. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

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

Authors' Addresses

Samuel Erb
Akamai Technologies

Email: serb@akamai.com

Rich Salz
Akamai Technologies

Email: rsalz@akamai.com