TLS
Internet-Draft
Intended status: Experimental
Expires: July 16, 2020

D. Benjamin
Google LLC

January 13, 2020

Batch Signing for TLS
draft-ietf-tls-batch-signing-00

Abstract

This document describes a mechanism for batch signing in TLS.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 16, 2020.

Copyright Notice

Table of Contents

1. Introduction 2
2. Conventions and Definitions 2
3. Batch SignatureSchemes 3
3.1. Signing 4
3.2. Verifying 6
4. Security Considerations 7
4.1. Correctness 7
4.2. Domain Separation 7
4.3. Payload Confidentiality 8
4.4. Information Leaks 8
5. IANA Considerations 9
6. Normative References 9
Appendix A. Test Vectors 10
Acknowledgments 10
Author's Address 10

1. Introduction

TLS [RFC8446] clients and servers authenticating with certificates perform online signatures with the private key associated with their certificate. In some cases, signing throughput may be limited. For instance, RSA signing is CPU-intensive compared to many other algorithms used in TLS. The private key may also be stored on a hardware module or be accessed remotely on another server. Under load, this can result in DoS concerns or impact system performance.

To mitigate these concerns, this document introduces a mechanism for batch signing in TLS. It allows TLS implementations to satisfy many concurrent requests with a single signing operation, at a logarithmic cost to signature size. A server under load could, for instance, preferentially serve batch-capable clients as part of its DoS strategy.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here. All TLS notation comes from section 3 of [RFC8446].

3. Batch SignatureSchemes

A batch SignatureScheme signs a number of input messages from different connections concurrently and returns a corresponding batch signature for each input message.

Each SignatureScheme is parameterized by the following:

- A base signature algorithm
- A hash function

This document defines the following values:

```
enum {
    ecdsa_secp256r1_sha256_batch(TBD1),
    ecdsa_secp384r1_sha384_batch(TBD2),
    ecdsa_secp521r1_sha512_batch(TBD3),
    ed25519_batch(TBD4),
    ed448_batch(TBD5),
    rsa_pss_pss_sha256_batch(TBD6),
    rsa_pss_rsae_sha256_batch(TBD7),
    (65536)
} SignatureScheme
```

"ecdsa_secp256r1_sha256_batch", "ecdsa_secp384r1_sha384_batch", and "ecdsa_secp521r1_sha512_batch" use base signature algorithms of "ecdsa_secp256r1_sha256", "ecdsa_secp384r1_sha384", and "ecdsa_secp521r1_sha512" with SHA-256, SHA-384, and SHA-512 [SHS], respectively, as the hash function.

"ed25519_batch" uses a base signature algorithm of "ed25519" with SHA-512 as the hash function. "ed448_batch" uses a base signature algorithm of "ed448" with 64 bytes (512 bits) of SHAKE256 [FIPS202] output as the hash function.

"rsa_pss_pss_sha256_batch" and "rsa_pss_rsae_sha256_batch" use base signature algorithms of "rsa_pss_pss_sha256" and "rsa_pss_rsae_sha256" with SHA-256 as the hash function.

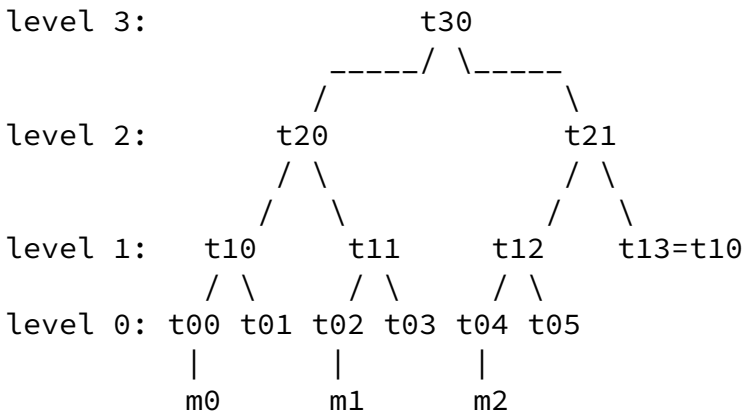
Batch signing is only defined for use with TLS 1.3. If TLS 1.2 is negotiated, the above code points MUST NOT be used in ServerKeyExchange or CertificateVerify messages. Note, however, a client which supports both TLS 1.2 and TLS 1.3 MAY offer the code points in the ClientHello.

These code points do not correspond to certificate signature algorithms. Implementations wishing to advertise support for the

base signature algorithm should send the base algorithm's corresponding code point.

3.1. Signing

Signing is performed by building a Merkle tree on top of the signing inputs, interspersed with blinding values. An example tree for three messages is shown below:



In general, let n be the number of input messages. If n is greater than 2^31, the signing procedure fails and returns an error. Otherwise, it builds a tree with l levels numbered 0 to l-1, where l is ceil(log_2(n)) + 2. Hashes in the tree are built from the following functions:

```
HashLeaf(msg) = Hash(0x00 || msg)
HashNode(left, right) = Hash(0x01 || left || right)
```

"0x00" and "0x01" denote byte strings containing a single byte with value zero and one, respectively. "||" denotes concatenation. "left" and "right" are byte strings with length Hash.length.

Tree levels are computed iteratively as follows:

1. Initialize level 0 with 2*n elements. For i between 0 and n-1, inclusive, set element 2*i to the output of HashLeaf(m[i]) and element 2*i+1 to a random string of Hash.length bytes. The random values placed at odd indices preserve signature payload confidentiality (see Section 4.3).

2. For i between 1 and l-1, inclusive, compute level i from level i-1 as follows:

    * If level i-1 has an odd number of elements, pad it to an even number of elements with a copy of its first element. That is, if the previous level contained three hashes, x, y, z, it should now contain four elements, x, y, z, x.

    * Initialize level i with half as many elements as level i-1. Set element j to the output of HashNode(left, right) where "left" is element 2*j of level i-1 and "right" is element 2*j+1 of level i-1. "left" and "right" are the left and right children of element j.

Level l-1 will contain a single element, the root of the tree. The signer then computes a digital signature using the base signature algorithm. This signature is computed over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times
- The context string "TLS batch signature"
- A single 0 byte which serves as the separator
- The batch signature algorithm's SignatureScheme code point, expressed as a big-endian 16-bit integer. Note this is the code point of the batch algorithm, not the original base algorithm.

o The value at the root of the tree

This structure is intended to provide key separation with other signatures in TLS (see Section 4.2).

The signer then constructs a BatchSignature structure, as defined below, for each input message. It encodes each to bytes to obtain the final signatures.

```
opaque Node[Hash.length];

struct {
    uint32 index;
    Node path<Hash.length..2^16-1>;
    opaque root_signature<0..2^16-1>;
} BatchSignature;
```

To assemble the BatchSignature structure for message i:

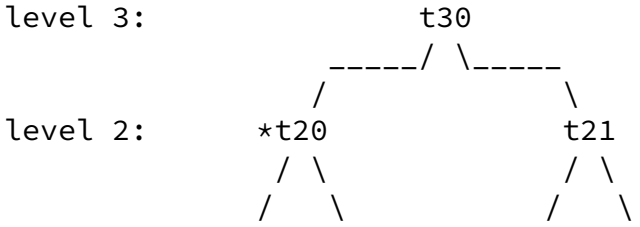
1. Set "index" to i. This will be a value between 0 and n-1, inclusive.

2. Set "path" to an array of l-1 hashes. Set element j of this array to element k of level j, where k is ((2 * i) >> j) ^ 1. ">>" denotes a bitwise right-shift, and "^" denotes a bitwise

exclusive OR (XOR) operation. This element is the sibling of the ancestor of message i in the tree. Note the root is never included.

3. Set "root_signature" to the digital signature computed above.

For example, in the diagram below, the "path" field of the signature of "m2" contains the marked nodes, in order from bottom to top.



```
level 1:   t10     t11     t12    *t13=t10
           / \     / \     / \
level 0: t00 t01 t02 t03 t04 *t05
           |       |       |
           m0      m1      m2
```

3.2. Verifying

The signature is verified by recovering the root hash from the supplied "path" and "index" fields and then verifying the signature in the "root_signature" field. This is done as follows:

1. If decoding the BatchSignature structure fails, terminate the algorithm and reject the signature.

2. If the value of the "index" field is 2^31 or higher, or if the number of elements in the "path" field is higher than 32, terminate the algorithm and reject the signature. Otherwise, set "remaining" to double this value.

3. Set "hash" to the output of HashLeaf(message).

4. For each element "v" of the "path" field, in order:

   * If "remaining" is odd, set "hash" to the output of HashNode(v, hash). Otherwise, set "hash" to the output of HashNode(hash, v)

   * Set "remaining" to remaining >> 1.

5. If "remaining" is non-zero, the signature is invalid. Terminate the algorithm and reject the signature.

6. As in the signing algorithm, concatenate the following:

   * A string that consists of octet 32 (0x20) repeated 64 times

   * The context string "TLS batch signature"

   * A single 0 byte which serves as the separator

* The batch signature algorithm's SignatureScheme code point, expressed as a big-endian 16-bit integer. Note this is the code point of the batch algorithm, not the original base algorithm.

* The value of "hash"

7. Verify that the "root_signature" field is a valid signature for the concatenation, using the base signature algorithm. If it is invalid, terminate the algorithm and reject the signature. Otherwise, accept the signature.

Note there are many possible valid signatures for a given message, depending on how many and what messages were batched together.

4. Security Considerations

4.1. Correctness

Batch signatures sign the root of a Merkle tree (see Section 3.1) so, provided the hash is collision-resistant and the base algorithm is secure, an attacker can only forge signatures of messages in the leaves of the Merkle tree. These leaves are the input messages, with the exception of padding and blinding nodes, discussed below.

When building the tree, this mechanism pads odd-length levels with extra copies of nodes already in the tree. This is equivalent to signing multiple copies of some input messages to bring the total to a power of two. This avoids introducing other messages for which the signature would also be valid. Verification (see Section 3.2) implicitly rejects odd indices in the tree to likewise ensure blinding values are not mistaken for message hashes.

4.2. Domain Separation

Signatures made by the same key in different contexts should be separated to avoid potential cross-protocol attacks. Inputs to the batch signing algorithm include any existing context strings, such as TLS 1.3's distinct client and server labels or new labels that may be

allocated by future versions of TLS. By signing over those labels,

batch signing preserves separation between those inputs.

The root signature additionally includes its own context string. This separates it from unbatched TLS 1.3 signatures, defined in section 4.4.3 of [RFC8446]. Like TLS 1.3, it additionally includes a 64-byte padding prefix to clear the ClientHello.random and ServerHello.random prefixes in the TLS 1.2 ServerKeyExchange signing payload. This allows the same key to be used for batched and unbatched signatures, simplifying deployment.

Finally, including the code point in the signature payload provides separation in case the same base signature algorithm is used in two batch constructions with, say, different hash functions.

4.3. Payload Confidentiality

The signing payload in TLS 1.3 is the handshake transcript. This contains information which is normally encrypted, such as the server certificate. Path elements in a batch signature are computed from payloads from other connections in the same batch. A naive construction could permit one peer to learn confidential information in other connections' signing payloads, such as which server certificate was selected in response to an encrypted SNI.

This mechanism avoids these attacks by pairing each input with a secret blinding value. An input's signature path will reveal the corresponding blinding value at level 0, but all other inputs in the path are incorporated in nodes at level 1 or higher. Provided the hash is preimage-resistant, these nodes do not reveal the original payload.

In the event of entropy failure when generating the blinding values, signatures remain unforgeable. The blinding values are only needed for payload confidentiality.

4.4. Information Leaks

A server observing multiple batched client signatures with the same root hash learns the two connections were created by the same client. However, the connections are already correlatable via the client certificate itself, so this does not reveal additional information in most deployments. Clients can partition the contexts in which signing requests may be batched to further mitigate these issues.

Additionally, a single batch signature reveals the number of signing requests in that batch, rounded up to a power of two. This may reveal some information about a service's signing load.

5. IANA Considerations

IANA is requested to create the following entries in the TLS SignatureScheme registry, defined in [RFC8446]. The "Reference" column should be set to this document.

Value	Description	Recommended
TBD1	ecdsa_secp256r1_sha256_batch	Y
TBD2	ecdsa_secp384r1_sha384_batch	Y
TBD3	ecdsa_secp521r1_sha512_batch	Y
TBD4	ed25519_batch	Y
TBD5	ed448_batch	Y
TBD6	rsa_pss_pss_sha256_batch	Y
TBD7	rsa_pss_rsae_sha256_batch	Y

6. Normative References

[FIPS202] Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[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>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report,

DOI 10.6028/nist.fips.180-4, July 2015.

Appendix A. Test Vectors

TODO: Include test vectors. Probably use ecdsa_secp256r1_sha256_batch. RSA signatures are big and Ed25519 isn't as common. Include some negative examples for verifying as well as intermediate values so signing code can at least compare against the tree-building vectors. (Blinding values and most of our defined signature schemes are non-deterministic.)

Acknowledgments

The mechanism described in this document is derived from a similar construction by Adam Langley in the Roughtime protocol. Adam also provided the initial suggestion to apply a similar technique to TLS.

Author's Address

David Benjamin
Google LLC

Email: davidben@google.com