

Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: April 22, 2020

M. Cavage  
Oracle  
M. Sporny  
Digital Bazaar  
October 20, 2019

Signing HTTP Messages  
draft-cavage-http-signatures-12

Abstract

When communicating over the Internet using the HTTP protocol, it can be desirable for a server or client to authenticate the sender of a particular message. It can also be desirable to ensure that the message was not tampered with during transit. This document describes a way for servers and clients to simultaneously add authentication and message integrity to HTTP messages by using a digital signature.

Feedback

This specification is a joint work product of the W3C Digital Verification Community Group [1] and the W3C Credentials Community Group [2]. Feedback related to this specification should be logged in the issue tracker [3] or be sent to [public-credentials@w3.org](mailto:public-credentials@w3.org) [4].

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 April 22, 2020.

## Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

## Table of Contents

1. Introduction . . . . .	3
1.1. Using Signatures in HTTP Requests . . . . .	4
1.2. Using Signatures in HTTP Responses . . . . .	4
2. The Components of a Signature . . . . .	4
2.1. Signature Parameters . . . . .	5
2.1.1. keyId . . . . .	5
2.1.2. signature . . . . .	5
2.1.3. algorithm . . . . .	5
2.1.4. created . . . . .	6
2.1.5. expires . . . . .	6
2.1.6. headers . . . . .	6
2.2. Ambiguous Parameters . . . . .	6
2.3. Signature String Construction . . . . .	7
2.4. Creating a Signature . . . . .	9
2.5. Verifying a Signature . . . . .	9
3. The 'Signature' HTTP Authentication Scheme . . . . .	10
3.1. Authorization Header . . . . .	10
3.1.1. Initiating Signature Authorization . . . . .	11
3.1.2. RSA Example . . . . .	11
3.1.3. HMAC Example . . . . .	12
4. The 'Signature' HTTP Header . . . . .	12
4.1. Signature Header . . . . .	12
4.1.1. RSA Example . . . . .	13
4.1.2. HMAC Example . . . . .	14
5. References . . . . .	14
5.1. Normative References . . . . .	14
5.2. Informative References . . . . .	14
5.3. URIs . . . . .	15
Appendix A. Security Considerations . . . . .	16
Appendix B. Extensions . . . . .	16
Appendix C. Test Values . . . . .	17
C.1. Default Test . . . . .	18
C.2. Basic Test . . . . .	18
C.3. All Headers Test . . . . .	19

Appendix D. Acknowledgements . . . . .	19
Appendix E. IANA Considerations . . . . .	20
E.1. Signature Authentication Scheme . . . . .	20
E.2. HTTP Signatures Algorithms Registry . . . . .	20
Authors' Addresses . . . . .	21

## 1. Introduction

This protocol extension is intended to provide a simple and standard way for clients to sign HTTP messages.

HTTP Authentication [RFC2617] defines Basic and Digest authentication mechanisms, TLS 1.2 [RFC5246] defines cryptographically strong transport layer security, and OAuth 2.0 [RFC6749] provides a fully-specified alternative for authorization of web service requests. Each of these approaches are employed on the Internet today with varying degrees of protection. However, none of these schemes are designed to cryptographically sign the HTTP messages themselves, which is required in order to ensure end-to-end message integrity. An added benefit of signing the HTTP message for the purposes of end-to-end message integrity is that the client can be authenticated using the same mechanism without the need for multiple round-trips.

Several web service providers have invented their own schemes for signing HTTP messages, but to date, none have been standardized. While there are no techniques in this proposal that are novel beyond the previous art, it is useful to standardize a simple and cryptographically strong mechanism for digitally signing HTTP messages.

This specification presents two mechanisms with distinct purposes:

1. The "Signature" scheme which is intended primarily to allow a sender to assert the contents of the message sent are correct and have not been altered during transmission or storage in a way that alters the meaning expressed in the original message as signed. Any party reading the message (the verifier) may independently confirm the validity of the message signature. This scheme is agnostic to the client/server direction and can be used to verify the contents of either HTTP requests, HTTP responses, or both.
2. The "Authorization" scheme which is intended primarily to allow a sender to request access to a resource or resources by proving that they control a secret key. This specification allows for this both with a shared secret (using HMAC) or with public/private keys. The "Authorization" scheme is typically used in authentication processes and not directly for message signing.

As a consequence 'Authorization' header is normally generated (and the message signed) by the HTTP client and the message verified by the HTTP server.

### 1.1. Using Signatures in HTTP Requests

It is common practice to protect sensitive website and API functionality via authentication mechanisms. Often, the entity accessing these APIs is a piece of automated software outside of an interactive human session. While there are mechanisms like OAuth and API secrets that are used to grant API access, each have their weaknesses such as unnecessary complexity for particular use cases or the use of shared secrets which may not be acceptable to an implementer. Shared secrets also prohibit any possibility for non-repudiation, while secure transports such as TLS do not provide for this at all.

Digital signatures are widely used to provide authentication and integrity assurances without the need for shared secrets. They also do not require a round-trip in order to authenticate the client, and allow the integrity of a message to be verified independently of the transport (e.g. TLS). A server need only have an understanding of the key (e.g. through a mapping between the key being used to sign the content and the authorized entity) to verify that a message was signed by that entity.

When optionally combined with asymmetric keys associated with an identity, this specification can also enable authentication of a client and server with or without prior knowledge of each other.

### 1.2. Using Signatures in HTTP Responses

HTTP messages are routinely altered as they traverse the infrastructure of the Internet, for mostly benign reasons. Gateways and proxies add, remove and alter headers for operational reasons, so a sender cannot rely on the recipient receiving exactly the message transmitted. By allowing a sender to sign specified headers, and recipient or intermediate system can confirm that the original intent of the sender is preserved, and including a Digest header can also verify the message body is not modified. This allows any recipient to easily confirm both the sender's identity, and any incidental or malicious changes that alter the content or meaning of the message.

## 2. The Components of a Signature

There are a number of components in a signature that are common between the 'Signature' HTTP Authentication Scheme and the

'Signature' HTTP Header. This section details the components of the digital signature parameters common to both schemes.

## 2.1. Signature Parameters

The following section details the Signature Parameters.

### 2.1.1. keyId

REQUIRED. The 'keyId' field is an opaque string that the server can use to look up the component they need to validate the signature. It could be an SSH key fingerprint, a URL to machine-readable key data, an LDAP DN, etc. Management of keys and assignment of 'keyId' is out of scope for this document. Implementations MUST be able to discover metadata about the key from the 'keyId' such that they can determine the type of digital signature algorithm to employ when creating or verifying signatures.

### 2.1.2. signature

REQUIRED. The 'signature' parameter is a base 64 encoded digital signature, as described in RFC 4648 [RFC4648], Section 4 [5]. The client uses the 'algorithm' and 'headers' Signature Parameters to form a canonicalized 'signing string'. This 'signing string' is then signed using the key associated with the 'keyId' according to its digital signature algorithm. The 'signature' parameter is then set to the base 64 encoding of the signature.

### 2.1.3. algorithm

RECOMMENDED. The 'algorithm' parameter is used to specify the signature string construction mechanism. Valid values for this parameter can be found in the HTTP Signatures Algorithms Registry [6] and MUST NOT be marked "deprecated". Implementers SHOULD derive the digital signature algorithm used by an implementation from the key metadata identified by the 'keyId' rather than from this field. If 'algorithm' is provided and differs from the key metadata identified by the 'keyId', for example 'rsa-sha256' but an EdDSA key is identified via 'keyId', then an implementation MUST produce an error. Implementers should note that previous versions of the 'algorithm' parameter did not use the key information to derive the digital signature type and thus could be utilized by attackers to expose security vulnerabilities.

#### 2.1.4. created

RECOMMENDED. The 'created' field expresses when the signature was created. The value MUST be a Unix timestamp integer value. A signature with a 'created' timestamp value that is in the future MUST NOT be processed. Using a Unix timestamp simplifies processing and avoids timezone management required by specifications such as RFC3339. Subsecond precision is not supported. This value is useful when clients are not capable of controlling the 'Date' HTTP Header such as when operating in certain web browser environments.

#### 2.1.5. expires

OPTIONAL. The 'expires' field expresses when the signature ceases to be valid. The value MUST be a Unix timestamp integer value. A signature with an 'expires' timestamp value that is in the past MUST NOT be processed. Using a Unix timestamp simplifies processing and avoid timezone management existing in RFC3339. Subsecond precision is allowed using decimal notation.

#### 2.1.6. headers

OPTIONAL. The 'headers' parameter is used to specify the list of HTTP headers included when generating the signature for the message. If specified, it SHOULD be a lowercased, quoted list of HTTP header fields, separated by a single space character. If not specified, implementations MUST operate as if the field were specified with a single value, '(created)', in the list of HTTP headers. Note:

1. The list order is important, and MUST be specified in the order the HTTP header field-value pairs are concatenated together during Signature String Construction (Section 2.3) used during signing and verifying.
2. A zero-length 'headers' parameter value MUST NOT be used, since it results in a signature of an empty string.

#### 2.2. Ambiguous Parameters

If any of the parameters listed above are erroneously duplicated in the associated header field, then the the signature MUST NOT be processed. Any parameter that is not recognized as a parameter, or is not well-formed, MUST be ignored.

### 2.3. Signature String Construction

A signed HTTP message needs to be tolerant of some trivial alterations during transmission as it goes through gateways, proxies, and other entities. These changes are often of little consequence and very benign, but also often not visible to or detectable by either the sender or the recipient. Simply signing the entire message that was transmitted by the sender is therefore not feasible: Even very minor changes would result in a signature which cannot be verified.

This specification allows the sender to select which headers are meaningful by including their names in the `'headers'` Signature Parameter. The headers appearing in this parameter are then used to construct the intermediate Signature String, which is the data that is actually signed.

In order to generate the string that is signed with a key, the client MUST use the values of each HTTP header field in the `'headers'` Signature Parameter, in the order they appear in the `'headers'` Signature Parameter. It is out of scope for this document to dictate what header fields an application will want to enforce, but implementers SHOULD at minimum include the `'(request-target)'` and `'(created)'` header fields if `'algorithm'` does not start with `'rsa'`, `'hmac'`, or `'ecdsa'`. Otherwise, `'(request-target)'` and `'date'` SHOULD be included in the signature.

To include the HTTP request target in the signature calculation, use the special `'(request-target)'` header field name. To include the signature creation time, use the special `'(created)'` header field name. To include the signature expiration time, use the special `'(expires)'` header field name.

1. If the header field name is `'(request-target)'` then generate the header field value by concatenating the lowercased `:method`, an ASCII space, and the `:path` pseudo-headers (as specified in HTTP/2, Section 8.1.2.3 [7]). Note: For the avoidance of doubt, lowercasing only applies to the `:method` pseudo-header and not to the `:path` pseudo-header.
2. If the header field name is `'(created)'` and the `'algorithm'` parameter starts with `'rsa'`, `'hmac'`, or `'ecdsa'` an implementation MUST produce an error. If the `'created'` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the `'created'` signature parameter.

3. If the header field name is ``(expires)`` and the ``algorithm`` parameter starts with ``rsa``, ``hmac``, or ``ecdsa`` an implementation MUST produce an error. If the ``expires`` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the ``created`` signature parameter.
4. Create the header field string by concatenating the lowercased header field name followed with an ASCII colon ``:``, an ASCII space `` ``, and the header field value. Leading and trailing optional whitespace (OWS) in the header field value MUST be omitted (as specified in RFC7230 [RFC7230], Section 3.2.4 [8]).
  1. If there are multiple instances of the same header field, all header field values associated with the header field MUST be concatenated, separated by a ASCII comma and an ASCII space ``,``, and used in the order in which they will appear in the transmitted HTTP message.
  2. If the header value (after removing leading and trailing whitespace) is a zero-length string, the signature string line correlating with that header will simply be the (lowercased) header name, an ASCII colon ``:``, and an ASCII space `` ``.
  3. Any other modification to the header field value MUST NOT be made.
  4. If a header specified in the headers parameter is malformed or cannot be matched with a provided header in the message, the implementation MUST produce an error.
5. If value is not the last value then append an ASCII newline ``\n``.

To illustrate the rules specified above, assume a ``headers`` parameter list with the value of ``(request-target) (created) host date cache-control x-emptyheader x-example`` with the following HTTP request headers:

```
GET /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
X-Example: Example header
           with some whitespace.
X-EmptyHeader:
Cache-Control: max-age=60
Cache-Control: must-revalidate
```



For the HTTP request headers above, the corresponding signature string is:

```
(request-target): get /foo
(created): 1402170695
host: example.org
date: Tue, 07 Jun 2014 20:51:35 GMT
cache-control: max-age=60, must-revalidate
x-emptyheader:
x-example: Example header with some whitespace.
```

#### 2.4. Creating a Signature

In order to create a signature, a client MUST:

1. Use the 'headers' and 'algorithm' values as well as the contents of the HTTP message, to create the signature string.
2. Use the key associated with 'keyId' to generate a digital signature on the signature string.
3. The 'signature' is then generated by base 64 encoding the output of the digital signature algorithm.

For example, assume that the 'algorithm' value is "hs2019" and the 'keyId' refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [9], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature creation algorithm should result in a binary string, which is then base 64 encoded and placed into the 'signature' value.

#### 2.5. Verifying a Signature

In order to verify a signature, a server MUST:

1. Use the received HTTP message, the 'headers' value, and the Signature String Construction (Section 2.3) algorithm to recreate the signature.
2. The 'algorithm', 'keyId', and base 64 decoded 'signature' listed in the Signature Parameters are then used to verify the authenticity of the digital signature. Note: The application verifying the signature MUST derive the digital signature algorithm from the metadata associated with the 'keyId' and MUST NOT use the value of 'algorithm' from the signed message.

If a header specified in the 'headers' value of the Signature Parameters (or the default item '(created)' where the 'headers' value is not supplied) is absent from the message, the implementation MUST produce an error.

For example, assume that the 'algorithm' value was "hs2019" and the 'keyId' refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [10], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature verification algorithm should result in a successful verification unless the headers protected by the signature were tampered with in transit.

### 3. The 'Signature' HTTP Authentication Scheme

The "Signature" authentication scheme is based on the model that the client must authenticate itself with a digital signature produced by either a private asymmetric key (e.g., RSA) or a shared symmetric key (e.g., HMAC).

The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

#### 3.1. Authorization Header

The client is expected to send an Authorization header (as defined in RFC 7235 [RFC7235], Section 4.1 [11]) where the "auth-scheme" is "Signature" and the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

```
POST /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18
```

```
{"hello": "world"}
```

Note that the use of the 'Digest' header field is per RFC 3230 [RFC3230], Section 4.3.2 [12] and is included merely as a demonstration of how an implementer could include information about

the body of the message in the signature. The following sections also assume that the "rsa-key-1" keyId asserted by the client is an identifier meaningful to the server.

### 3.1.1. Initiating Signature Authorization

A server may notify a client when a resource is protected by requiring a signature. To initiate this process, the server will request that the client authenticate itself via a 401 response [13] code. The server may optionally specify which HTTP headers it expects to be signed by specifying the 'headers' parameter in the WWW-Authenticate header. For example:

```
HTTP/1.1 401 Unauthorized
Date: Thu, 08 Jun 2014 18:32:30 GMT
Content-Length: 1234
Content-Type: text/html
WWW-Authenticate: Signature
    realm="Example",headers="(request-target) (created)"

...
```

### 3.1.2. RSA Example

The authorization header and signature would be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
    headers="(request-target) (created) host digest content-length",
    signature="Base64(RSA-SHA512(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64 (RSA-SHA512(signing string))"
```

### 3.1.3. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Authorization: Signature keyId="hmac-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64 (HMAC-SHA512(signing string))"
```

The only difference between the RSA Example and the HMAC Example is the digital signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

## 4. The 'Signature' HTTP Header

The "Signature" HTTP Header provides a mechanism to link the headers of a message (client request or server response) to a digital signature. By including the "Digest" header with a properly formatted digest, the message body can also be linked to the signature. The signature is generated and verified either using a shared secret (e.g. HMAC) or public/private keys (e.g. RSA, EC). This allows the receiver and/or any intermediate system to immediately or later verify the integrity of the message. When the signature is generated with a private key it can also provide a measure of non-repudiation, though a full implementation of a non-repudiable statement is beyond the scope of this specification and highly dependent on implementation.

The "Signature" scheme can also be used for authentication similar to the purpose of the 'Signature' HTTP Authentication Scheme (Section 3). The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

### 4.1. Signature Header

The sender is expected to transmit a header (as defined in RFC 7230 [RFC7230], Section 3.2 [14]) where the "field-name" is "Signature", and the "field-value" contains one or more "auth-param"s (as defined

in RFC 7235 [RFC7235], Section 4.1 [15]) where the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

```
POST /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

{"hello": "world"}
```

The following sections assume that the "rsa-key-1" keyId provided by the signer is an identifier meaningful to the server.

#### 4.1.1. RSA Example

The signature header and signature would be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",
  created=1402170695, expires=1402170995,
  headers="(request-target) (created) (expires)
  host date digest content-length",
  signature="Base64(RSA-SHA256(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402170695
(expires): 1402170995
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",created=1402170695,  
  headers="(request-target) (created) host digest content-length",  
  signature="Base64 (RSA-SHA512(signing string))"
```

#### 4.1.2. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Signature: keyId="hmac-key-1",algorithm="hs2019",created=1402170695,  
  headers="(request-target) (created) host digest content-length",  
  signature="Base64 (HMAC-SHA512(signing string))"
```

The only difference between the RSA Example and the HMAC Example is the signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n  
(created): 1402170695  
host: example.org\n  
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n  
content-length: 18
```

## 5. References

### 5.1. Normative References

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/info/rfc7235>>.

### 5.2. Informative References

- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <<https://www.rfc-editor.org/info/rfc2617>>.

- [RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <<https://www.rfc-editor.org/info/rfc3230>>.
- [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>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [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>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.

### 5.3. URIs

- [1] <https://w3c-dvcg.github.io/>
- [2] <https://w3c-ccg.github.io/>
- [3] <https://github.com/w3c-dvcg/http-signatures/issues>
- [4] <mailto:public-credentials@w3.org>
- [5] <https://tools.ietf.org/html/rfc4648#section-4>
- [6] [#hsa-registry](#)
- [7] <https://tools.ietf.org/html/rfc7540#section-8.1.2.3>
- [8] <https://tools.ietf.org/html/rfc7230#section-3.2.4>
- [9] [#canonicalization](#)

- [10] #canonicalization
- [11] <https://tools.ietf.org/html/rfc7235#section-2.1>
- [12] <https://tools.ietf.org/html/rfc3230#section-4.3.2>
- [13] <https://tools.ietf.org/html/rfc7235#section-3.1>
- [14] <https://tools.ietf.org/html/rfc7230#section-3.2>
- [15] <https://tools.ietf.org/html/rfc7235#section-4.1>
- [16] <https://web-payments.org/specs/source/http-signatures-audit/>
- [17] <https://web-payments.org/specs/source/http-signature-nonces/>
- [18] <https://web-payments.org/specs/source/http-signature-trailers/>
- [19] <https://www.iana.org/assignments/http-auth-scheme-signature>
- [20] <https://www.iana.org/assignments/http-authschemes>
- [21] <https://www.iana.org/assignments/shm-algorithms>
- [22] #canonicalization
- [23] #canonicalization
- [24] #canonicalization
- [25] #canonicalization
- [26] #canonicalization

## Appendix A. Security Considerations

There are a number of security considerations to take into account when implementing or utilizing this specification. A thorough security analysis of this protocol, including its strengths and weaknesses, can be found in Security Considerations for HTTP Signatures [16].

## Appendix B. Extensions

This specification was designed to be simple, modular, and extensible. There are a number of other specifications that build on this one. For example, the HTTP Signature Nonces [17] specification details how to use HTTP Signatures over a non-secured channel like



HTTP and the HTTP Signature Trailers [18] specification explains how to apply HTTP Signatures to streaming content. Developers that desire more functionality than this specification provides are urged to ensure that an extension specification doesn't already exist before implementing a proprietary extension.

If extensions to this specification are made by adding new Signature Parameters, those extension parameters MUST be registered in the Signature Authentication Scheme Registry. The registry will be created and maintained at (the suggested URI) <https://www.iana.org/assignments/http-auth-scheme-signature> [19]. An example entry in this registry is included below:

```
Signature Parameter: nonce
Reference to specification: [HTTP_AUTH_SIGNATURE_NONCE], Section XYZ.
Notes (optional): The HTTP Signature Nonces specification details
how to use HTTP Signatures over a unsecured channel like HTTP.
```

#### Appendix C. Test Values

WARNING: THESE TEST VECTORS ARE OLD AND POSSIBLY WRONG. THE NEXT VERSION OF THIS SPECIFICATION WILL CONTAIN THE PROPER TEST VECTORS.

The following test data uses the following RSA 2048-bit keys, which we will refer to as 'keyId=Test' in the following samples:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDCFENGw33yGiHy92pDjZQh10C3
6rPJj+CvfSC8+q28hxA161QFNu13wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6
Z4UMR7EOcpfdUE9Hf3m/hs+FUR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJw
oYi+lhqplfIekaxsyQIDAQAB
-----END PUBLIC KEY-----

-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDCFENGw33yGiHy92pDjZQh10C36rPJj+CvfSC8+q28hxA161QF
NUd13wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6Z4UMR7EOcpfdUE9Hf3m/hs+F
UR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJwoYi+lhqplfIekaxsyQIDAQAB
AoGBAJR8ZkCUvx5kzv+utdl7T5MnordT1TvoXXJGXX7ZZ+UuvMNUCdn2QPc4sBiA
QWvLwlCskt5DsKZ8UETpYPy8pPYnnDEz2dDYiaew9+xEpubyeW2oH4Zx71wqBtOK
kqwrXa/pzdpiucRRjk6vE6YY7EBBs/g7uanVpGibOVAESqH1AkeA7DkjVH28WDUg
flnqvfn2Kj6CT7nIcE3jGJsZZ7z1ZmBmHFDONMLUrXR/Zm3pR5m0tCmBqa5RK95u
412jtldPIwJBANJT3v8pnkth48bQo/fKel6uEYyboRtA5/uHuHkZ6FQF7OUkGogc
mSJluOdc5t6hI1VsLn0QZEjQZMEOWr+wKSMCQQCC4kXJESHAve77oP6HtG/IiEn7
kpyUXRNVfsDE0czpJJBvL/aRFUJxuRK91jhjC68sA7NsKMG5OXb5I5Jj36xAkEA
gIT7aFOYBFwGgQAQkWNKLvySgKbAZRTElBacpHMuQdl1DfdntvAyqpAZ01Y0RKmW
G6aFKaqQfOXKCyWoUiVknQJAXrlgySFci/2ueKlIE1QqIiLSZ8V80lpFLRnb1pzI
7UlyQXnTAEFYM560yJlZUpOb1V4cScGd365tiSMvxLOvTA==
-----END RSA PRIVATE KEY-----
```

All examples use this request:

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Sun, 05 Jan 2014 21:31:40 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

{"hello": "world"}
```

#### C.1. Default Test

If a list of headers is not included, the date is the only header that is signed by default for rsa-sha256. The string to sign would be:

```
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5couls7uEGKKCs+FLEEaDV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwbldHYGETbGmtdHgVCk9SuS13F0hZ8FD0k/50xEPXe5WozsbM="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5couls7uEGKKCs+FLEEaDV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwbldHYGETbGmtdHgVCk9SuS13F0hZ8FD0k/50xEPXe5WozsbM="
```

#### C.2. Basic Test

The minimum recommended data to sign is the (request-target), host, and date. In this case, the string to sign would be:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  headers="(request-target) host date",
  signature="qdx+H7PHHDZgy4y/Ahn9Tny9V3GP6YgBPYUXMmoxWtLbHpUnXS
2mg2+SbrQDMCJypxBLSPQR2aAjn7ndmw2iicw3HMbe8VfEdKFYRqzic+efkb3
nndiv/xlxSHDJWeSWkx3ButlYSuBskLu6kd9Fswtemr3lgdEmn04swr2Os0="
```

### C.3. All Headers Test

A strong signature including all of the headers and a digest of the body of the HTTP request would result in the following signing string:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
content-type: application/json
digest: SHA-256=X48E9qOokqgrvdtS8nOJRJN3OWDUoyWxBf7kbu9DBPE=
content-length: 18
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
  host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iF1rhgYkz8+oVLEEzmYZZvRs
8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
ukWgHoBTLmHYS2Gb51gWxpeIq8knRmPnYePbF5MokR0Zkly4zKH7s1dE="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
  host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iF1rhgYkz8+oVLEEzmYZZvRs
8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
ukWgHoBTLmHYS2Gb51gWxpeIq8knRmPnYePbF5MokR0Zkly4zKH7s1dE="
```

### Appendix D. Acknowledgements

The editor would like to thank the following individuals for feedback on and implementations of the specification (in alphabetical order): Mark Adamcin, Mark Allen, Paul Annesley, Karl Boehlmark, Stephane Bortzmeyer, Sarven Capadisli, Liam Dennehy, ductm54, Stephen Farrell, Phillip Hallam-Baker, Eric Holmes, Andrey Kislyuk, Adam Knight, Dave Lehn, Dave Longley, James H. Manger, Ilari Liusvaara, Mark Nottingham, Yoav Nir, Adrian Palmer, Lucas Pardue, Roberto Polli,

Julian Reschke, Michael Richardson, Wojciech Ryskielski, Adam Scarr,  
Cory J. Slep, Dirk Stein, Henry Story, Lukasz Szewc, Chris Webber,  
and Jeffrey Yasskin

## Appendix E. IANA Considerations

### E.1. Signature Authentication Scheme

The following entry should be added to the Authentication Scheme Registry located at <https://www.iana.org/assignments/http-authschemes> [20]

Authentication Scheme Name: Signature  
Reference: [RFC\_THIS\_DOCUMENT], Section 2.  
Notes (optional): The Signature scheme is designed for clients to authenticate themselves with a server.

### E.2. HTTP Signatures Algorithms Registry

The following initial entries should be added to the Canonicalization Algorithms Registry to be created and maintained at (the suggested URI) <https://www.iana.org/assignments/shm-algorithms> [21]:

Editor's note: The references in this section are problematic as many of the specifications that they refer to are too implementation specific, rather than just pointing to the proper signature and hashing specifications. A better approach might be just specifying the signature and hashing function specifications, leaving implementers to connect the dots (which are not that hard to connect).

Algorithm Name: hs2019  
Status: active  
Canonicalization Algorithm: [RFC\_THIS\_DOCUMENT], Section 2.3:  
Signature String Construction [22]  
Hash Algorithm: RFC 6234 [RFC6234], SHA-512 (SHA-2 with 512-bits of digest output)  
Digital Signature Algorithm: Derived from metadata associated with 'keyId'. Recommend support for RFC 8017 [RFC8017], Section 8.1: RSASSA-PSS, RFC 6234 [RFC6234], Section 7.1: SHA-Based HMACs, ANSI X9.62-2005 ECDSA, P-256, and RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519.

Algorithm Name: rsa-sha1  
Status: deprecated, SHA-1 not secure.  
Canonicalization Algorithm: [RFC\_THIS\_DOCUMENT], Section 2.3:  
Signature String Construction [23]

Hash Algorithm: RFC 6234 [RFC6234], SHA-1 (SHA-1 with 160-bits of digest output)  
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1\_5

Algorithm Name: rsa-sha256  
Status: deprecated, specifying signature algorithm enables attack vector.  
Canonicalization Algorithm: [RFC\_THIS\_DOCUMENT], Section 2.3:  
Signature String Construction [24]  
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)  
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1\_5

Algorithm Name: hmac-sha256  
Status: deprecated, specifying signature algorithm enables attack vector.  
Canonicalization Algorithm: [RFC\_THIS\_DOCUMENT], Section 2.3:  
Signature String Construction [25]  
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)  
Message Authentication Code Algorithm: RFC 6234 [RFC6234],  
Section 7.1: SHA-Based HMACs

Algorithm Name: ecdsa-sha256  
Status: deprecated, specifying signature algorithm enables attack vector.  
Canonicalization Algorithm: [RFC\_THIS\_DOCUMENT], Section 2.3:  
Signature String Construction [26]  
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)  
Digital Signature Algorithm: ANSI X9.62-2005 ECDSA, P-256

#### Authors' Addresses

Mark Cavage  
Oracle  
500 Oracle Parkway  
Redwood Shores, CA 94065  
US

Phone: +1 415 400 0626  
Email: mcavage@gmail.com  
URI: <https://www.oracle.com/>

Manu Sporny  
Digital Bazaar  
203 Roanoke Street W.  
Blacksburg, VA 24060  
US

Phone: +1 540 961 4469  
Email: msporny@digitalbazaar.com  
URI: <https://manu.sporny.org/>

Internet Engineering Task Force  
Internet-Draft  
Intended status: Informational  
Expires: May 7, 2020

J. Fabini  
TU Wien  
November 4, 2019

Communication Network Perspective on Malware Lifecycle  
draft-fabini-smart-malware-lifecycle-00

Abstract

Today's systems, networks, and protocols are complex and include unknown vulnerabilities that adversaries can exploit. The large-scale deployment of network security protocols establishes an additional threat by implementing a substrate for hidden communications like covert or subliminal channels. The resulting ecosystem builds a convenient platform for malicious, automated software (malware) to infiltrate critical infrastructures, to gradually infect large parts of the system and to coordinate distributed malware operation.

Based on the observation that malware depends on network communications to discover, propagate, coordinate, and unleash its functionality, this memo recommends methods to identify potential interfaces and interactions between malware and protocols. It proposes a generic malware lifecycle model that defines a set of generic malware states and possible transitions between these states. Coordinated activities of distributed malware can be mapped to state transitions in malware instances, supporting the identification of (potentially hidden) network communication as a trigger for actions and hints on protocols that enabled the communication. Eventually, the proposed model aims at supporting the identification of architectures, protocols, interfaces, and points in time that a) either inhibit hidden malware communication or b) allow for optimized detection of anomalies as main prerequisite for timely countermeasures.

While earlier work focused on protecting single hosts from compromise, this memo adopts a holistic view and considers the health of the overall networked system to be of highest priority. Presuming vulnerable systems, we stress that components or subsystems must be disconnected on suspected infection in an attempt to continue (even partial) operation of the overall (non-infected) system after the disconnect. Containment - the isolation of an infected subsystem - becomes an essential security feature in the context of critical infrastructures that influences on deployed protocols, interfaces and architectures.

## 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 May 7, 2020.

## Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Requirements Language . . . . .	4
2. Generic Malware Lifecycle . . . . .	4
2.1. Access . . . . .	6
2.2. Infection . . . . .	6
2.3. Discovery . . . . .	7
2.4. Propagation . . . . .	7
2.5. Control . . . . .	8
2.6. Trigger . . . . .	8
2.7. Attack . . . . .	8
2.8. Cleanup . . . . .	9
3. Mapping the Lifecycle Model to Real Malware . . . . .	9
3.1. Case study: Stuxnet . . . . .	10
3.1.1. Access . . . . .	11



3.1.2.	Infection . . . . .	11
3.1.3.	Discovery . . . . .	11
3.1.4.	Propagation . . . . .	11
3.1.5.	Control . . . . .	12
3.1.6.	Trigger . . . . .	12
3.1.7.	Attack . . . . .	12
3.1.8.	Cleanup . . . . .	12
3.1.9.	Discussion: Stuxnet . . . . .	12
4.	Future work . . . . .	13
5.	Acknowledgements . . . . .	13
6.	IANA Considerations . . . . .	13
7.	Security Considerations . . . . .	14
8.	References . . . . .	14
8.1.	Normative References . . . . .	14
8.2.	Informative References . . . . .	14
	Author's Address . . . . .	15

## 1. Introduction

A central guideline of the IETF security area's activity focus is summarized in RFC 3552 [RFC3552]: "Protecting against an attack when one of the end-systems has been compromised is extraordinarily difficult". This statement is still valid today but must be seen in a historical context: in times of monolithic systems, the main goal of security is (or was) to protect one's own networked end system (PC, server) against compromise. This implies a worst case scenario and "game over" in case of a system compromise. In a distributed context, one single compromised system can be fatal whenever relying on a chain of trust, which is a common security policy within closed (corporate or enterprise) networks.

However, architectures and protocols have evolved. Emerging critical infrastructures consist of ensembles of hundreds, thousands or tens of thousands of identical networked systems like for instance smart meters or other Internet of Things (IoT) devices. These systems all run identical software and identical firmware on top of identical hardware, all of them being potentially subject to identical vulnerabilities. Likewise, most personal computers that are connected to the Internet run one of a few operating system alternatives, including Microsoft Windows, Apple MacOS, or various Linux distributions. Portable software and common Application Programming Interfaces (APIs) increase the likelihood that one vulnerability affects multiple platforms.

When viewing a system as a complex set of components and relations (Rechtin [CBCS]), there are cases when vital system functions can be performed even in the case when some subsystems (components or links) have been compromised. Therefore, today's security concepts and

research must support (a) identification and (b) containment, i.e., isolation, of compromised subsystems at an architectural and protocol level. It is important to note that these requirements *\*extend\** (and by no means contradict) the requirements stated in RFC 3552 [RFC3552] with respect to the importance of protecting systems against compromise.

On this purpose, this memo proposes to enlarge the scope of systems security, starting from two main prerequisites, namely that (a) any system (single end node, component, link) is vulnerable and (b) malware must communicate to propagate, to discover and to coordinate its distributed instances. Section 2 proposes a generic malware lifecycle model consisting of malware states and transitions. This generic state diagram is subsequently mapped to existing malware implementations to infer on malware communication needs, as well as on potential interfaces and protocols that malware may use for discovery, infection, propagation, and control through available network paths. By monitoring these interfaces, systems can detect patterns of - potentially hidden - communications as an anomalous component of the network traffic. Subsequent analysis of available architectures, interfaces, and protocols can help in identifying anomalous communications and stopping it in order to prevent malware from propagation and execution.

We consider the identification of systematic and design shortcomings of architectures and protocols with respect to hidden communications to be an essential component of the security-by-design concept. A first step is the definition of metrics and methods that can assess the degree to which protocols under investigation support -- or prevent -- hidden communications. The ability to evaluate protocols and choose the ones that are proven to be covert-channel free enables system architects to close existing gaps for hidden malware communication.

Todo: the terms used in this memo should be eventually aligned to [I-D.mcfadden-smart-endpoint-taxonomy-for-class].

### 1.1. Requirements Language

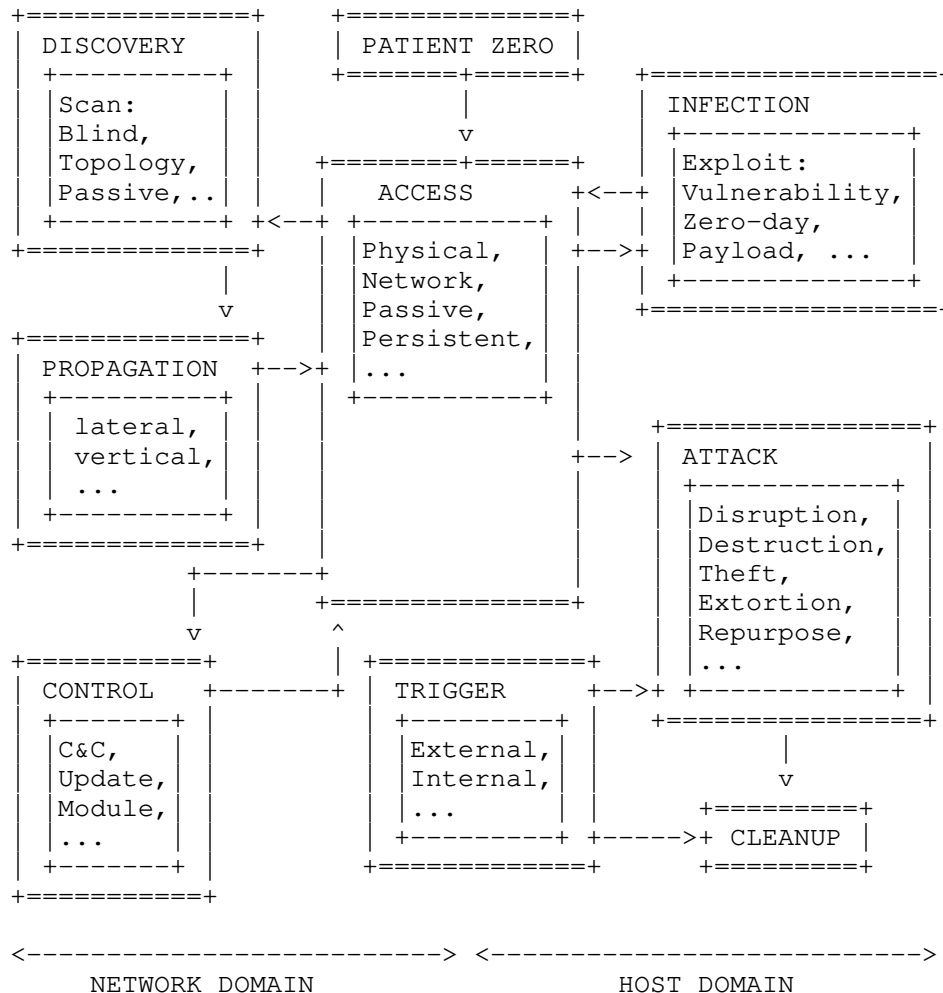
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 2. Generic Malware Lifecycle

The state diagram depicted in Figure 1 illustrates a generic malware lifecycle model. A graphical representation of the diagram along with a detailed description can be found in the original publication

[GML] or its pre-published version at  
[https://publik.tuwien.ac.at/files/publik\\_261089.pdf](https://publik.tuwien.ac.at/files/publik_261089.pdf).

Generic Stages of Malware Lifecycle.



An extended graphical representation of this diagram along with detailed descriptions can be found in [GML].

Figure 1

Malware activity in Figure 1 revolves around the concept of access to abstract resources. Essential from an defender's monitoring perspective is that, depending on their implementation and target,

malware variants differ substantially in their use of communication networks. Common to many recent malware is that it encrypts communication, attempts to obfuscate it as legitimate traffic, and/or uses hidden communication channels to stay unobserved. Aggressiveness and "noise" that malware generates while propagating, infecting and attacking differs substantially between malware types.

This is why this memo focuses on evaluating protocols, interfaces and architectures with respect to their ability to inhibit or support hidden communications. The proposed generic lifecycle model can identify the malware's need for communication to trigger state changes. ( Internal: provide hints to anomaly detection systems? estimated amount of data as an order of magnitude: transferring a malware update or additional malware modules requires more data transfer than a single command. )

The following subsections discuss briefly the generic stages of malware lifecycle in line with [GML].

## 2.1. Access

Starting point of malware operation is the so-called patient zero, denoting a device or method that triggers the initial infection within the system under observation. Examples for access options include, but are not limited to physical access (e.g., through a compromised USB stick inserted into a computer, through hard drive replacement or through starting from a temporary boot device), network access (e.g., as part of existing connections, or through a hidden communication channel), application access (e.g., by sending a legitimate email with compromised payload), or persistent access (for instance an intentional or unintentional backdoor that is installed by firmware or BIOS).

The patient zero may depend on qualified (human) support to bypass existing security barriers and gain access to the system. This may be, e.g., a staff member plugging a compromised USB stick into a computer to infiltrate an air-gapped system, or an employee of the computer manufacturer who adds a backdoor to the computer BIOS, firmware or software. Once it has gained access to the target system, the malware can start its operation.

Todo: Extend, discuss options.

## 2.2. Infection

Having gained (temporary) access to the system, malware depends on system vulnerabilities to support its attempt to infect the system and install itself persistently. Examples include the exploit of

backdoors, zero-day vulnerabilities or execution of a malicious email attachment.

Todo: Extend, discuss options.

### 2.3. Discovery

Once the local system is infected, malware has several options. The most common malware strategy is to first discover new potential victims that are reachable via the communication network. Alternatives for discovery differ in terms of communication verbosity and range from blind scans to passive monitoring of incoming network connections and many variants in between. Blind scans are the most aggressive but also the most verbose variant of discovery, malware actively scanning ranges of IPv4 or IPv6 addresses like, e.g., the current subnet or all IPv4 addresses. In typical networks monitoring devices can easily detect these blind scans because of the high volume of additional illegitimate traffic. Adding some more intelligence to the discovery process results in targeted scans to decrease the amount of traffic that is needed for probing. Examples include the support for distributed scan lists that record already scanned (and infected) devices, or a prioritization of the scan process to prefer system-critical devices like, e.g., the standard gateway. Most stealthy and most difficult to detect is malware that monitors passively its local network interfaces on incoming and outgoing traffic to infer on the network topology and potential targets. However, this stealthiness comes at the cost of reduced malware propagation speed and is typical for complex attack patterns.

It is worth mentioning that the supported IP address version has substantial impact on the discovery strategy that malware may use or prefer. Whenever targeting IPv4 addresses, distributed malware can scan the entire Internet within reasonable time. The large address space of IPv6 and the resulting sparse population of subnets will likely result in malware to prefer targeted active scans or passive scanning for the discovery process.

Todo: Extend, discuss options.

### 2.4. Propagation

Following the discovery of a potential victim, malware attempts to propagate over existing communication channels to gain access to these victims and install new instances of itself in the network.

Todo: Extend, discuss options.

## 2.5. Control

All presented malware activities or state changes happen either autonomously, which is typical for early malware variants, or guided by some command & control infrastructures that recent malware variants prefer to allow for later malware modification and coordinated attacks. Examples of the latter variant include malware that supports remotely controlled updates, loading of new modules and distributed C&C structures. Such functionality facilitates the update of encryption keys, communication patterns and functionality, as well as the support for new communication protocols. Eventually, this functionality enables offerings business models of "malware as a service": botnet owners may operate infrastructures of compromised devices that customers can rent and use to execute their custom-tailored malicious code.

Todo: Extend, discuss options.

## 2.6. Trigger

Triggers are essential for supporting the coordination of functionality in distributed malware instances, typical example being the launch of a coordinated DDoS attack. Explicit control communication (command) is one option for an external trigger, other less suspicious options include the setting of conditions that distributed malware instances can observe. Examples include timers (some malware variants implementing explicit time synchronization with dedicated time servers for improved accuracy) but also availability of specific servers at specific domain names, etc.

Internal triggers are typically hard-coded into the malware or its modules and support it in targeting and focusing its attacks. These triggers can, for instance, control malware to launch its attacks on specific hardware- or software systems only, or can limit its actions to specific IP address ranges and/or DNS domains.

Todo: Extend, discuss options.

## 2.7. Attack

Once successfully propagated, malware can start its damaging functionality that ranges from destruction and disruption to theft or extortion.

Todo: Extend, discuss options.

## 2.8. Cleanup

Recent malware variants focusing on stealthy operation include hidden communication and cleanup functionality to remove themselves from infected systems. The cleanup starts either on completing the attack or on external triggers after accomplishing their goal.

Todo: Extend, discuss options.

## 3. Mapping the Lifecycle Model to Real Malware

This section maps the known behavior of well-studied, prototypical malware variants to the Malware Lifecycle Model. Eventually, this mapping aims at identifying malware communication needs and behavioral patterns that automated processes can use to discover unknown malware.

Central observation with respect to the Malware Lifecycle Model's applicability is that malware has huge incentives to communicate, and that monitoring devices can detect this communication as anomaly. In particular, network communication is a key component for malware to unleash its full destructive potential. Infecting systems remotely and automating and coordinating their distributed activities using network communications brings huge benefits to malware authors. Most notably, being physically located in distinct geographical, jurisdictional, and/or legislative regions supports networked operations while minimizing the risk of being prosecuted for the results of these actions.

Bridging the air gap to an isolated system is conditioned by physical access to the system. Options include access to the system or to parts of it, either during the manufacturing process (e.g., by compromising a computer's BIOS and adding a backdoor) or later on, during installation or operation (e.g., by inserting a compromised USB drive into the system). From a malware author's perspective, the physical access alternative has severe drawbacks. First, the need for physical access may leave traces that help in identifying the originator. Second, the lack of updates and coordination: malware must be fully functional at the time of first infection, updates for it depending on recurring physical access to the system. However, even in the case of air-gapped systems malware may subsequently attempt to discover and infect locally connected systems (as exhibited for instance by Stuxnet). These communication attempts may be monitored and detected.

Summarizing, the main incentives for malware to communicate include the following:

- o Network-based malware coordination and control: the closer coordinated distributed malware instances can act, the higher the potential severity of their aggregated actions (for instance in the case of DDoS attacks). Malware may use coordination to reduce network traffic, too (for example by maintaining scan lists when scanning for new victims).
- o Network-based update: the complexity and sophistication of today's malware increases the effort for its programming. This drives the trend for modular malware that can install a minimum persistent foothold, update itself and can load novel functionality on demand as additional modules. Malware update can support malware authors by protecting their assets in the case of malware identification and/or takeover attempts by competing organizations. In such cases, malware updates can support in the modifications of keys, change of encryption algorithms, use of novel obfuscation methods, etc.
- o Network-based discovery of potential infection targets and propagation: Scanning for infection candidates and propagation range among the two most verbose activities of today's malware. Worth noting is that specific malware functionality is typically related to malware size, i.e., data volume that the malware must transfer. Depending on the implementation, malware can decide to transfer its entire body at propagation time or install a tiny foothold during propagation that subsequently loads the required modules. The data pattern that monitoring devices can identify differs for these two alternatives: the self-carried malware will be visible in monitoring logs only once, when transferring a large amount of data. The modular variant consists of several smaller data transfers.
- o tbc...

The remainder of this section presents prototypical case studies of existing malware variants, the mapping of their behavior to malware lifecycle model stages and how the lifecycle model can support in their detection. Tables 1-4 of [GML] compare and discuss features and peculiarities of various malware variants in more detail. Future versions of this draft are planned to structure and extend the malware communication aspects that these tables summarize, eventually building the base for a generic malware detection framework.

### 3.1. Case study: Stuxnet

Stuxnet [Stuxnet] is a computer worm that was reported for the first time in June 2010. The effort associated with the design and implementation of Stuxnet was substantial, pointing to nation states



or intelligence services as authors. This speculation is backed by Stuxnet's stealthy behavior and targeted attack against Siemens Simatic S7 Supervisory Control and Data Acquisition (SCADA) Industrial Control Systems (ICS), eventually aimed at causing physical damage. The following subsections map the known Stuxnet behavioral and communication patterns to the generic Malware Lifecycle model stages and transitions<sup>1</sup>.

#### 3.1.1. Access

The initial Stuxnet access (patient zero) targets air-gapped systems. It uses the autostart functionality of Microsoft Windows 32 bit operating system variants on inserting a USB stick. As soon as the first system has been infected, Stuxnet attempts to discover and access other computers within the same LAN. Whereas the initial infection (USB drive autostart) can not be captured by the Lifecycle Model, the access to other computers within the LAN can be monitored within the network traffic.

#### 3.1.2. Infection

Stuxnet exploits several zero-day vulnerabilities that were unknown by the time of its release and allowed for privilege escalation on several Microsoft Windows 32 bit operating system variants. In addition, Stuxnet made use of two stolen certificates to sign its drivers. The infection includes installation of dedicated RPC servers and -clients and peer-to-peer clients for communication with other infected Stuxnet instances within the same LAN, as well as infection of connected network shares. Local infection and installation is not in the scope of the Lifecycle Model, whereas the infection of network shares may lead to unexpected network traffic and monitored network anomalies.

#### 3.1.3. Discovery

Following an initial infection, Stuxnet scans the local network for potential, previously uninfected targets. Stuxnet also uses specific domains to probe for Internet connectivity. All of these network scan operations are typical and can be monitored and detected.

#### 3.1.4. Propagation

Whenever Stuxnet identifies uninfected targets in the local network with Siemens Step7 software installed, it propagates and attempts to infect these PCs. Otherwise it enters a dormant mode.

### 3.1.5. Control

Stuxnet instances within the same network use peer-to-peer RPC calls and encryption to update each other. This method allows one single USB drive infection to update distributed Stuxnet instances in air-gapped systems. Whenever Internet access is available, Stuxnet contacts command and control servers using encrypted communication to receive updates, additional features, and instructions. These update RPC calls and the traffic to command and control servers can be identified by network monitoring systems as anomalies.

### 3.1.6. Trigger

Stuxnet installation is conditioned by software (Siemens Step7) software to be installed on, and/or Siemens PLCs being connected to the Windows-based system. The Lifecycle Model can not capture these triggers as they are proprietary to the malware and do not involve network communication.

### 3.1.7. Attack

Once installed on a system that controls a PLC, Stuxnet acts as a man-in-the-middle. Faulty commands, aimed to cause physical damage, are sent to the PLC, and forged PLC response codes are forwarded by Stuxnet back to the controlling application to pretend correct operation. Complex (cross-layer) monitoring systems, featuring sensors inside the PLC, could identify the mismatch between the commands sent by the controlling application and the commands received by the PLC. Likewise, system log correlation with network traffic data could reveal anomalous behavior.

### 3.1.8. Cleanup

Stuxnet stores several encrypted copies of itself on infected systems. Whereas cleanup on the host system should be feasible, Stuxnet can not delete the malicious code that has been sent to the PLC. Therefore, Stuxnet will leave traces that may identify its presence.

### 3.1.9. Discussion: Stuxnet

An analysis of Stuxnet reveals communication patterns that can be matched to specific stages of the Malware Lifecycle Model. Depending on the specific network architecture and on the type of systems connected to the network under observation, these communications may appear more or less anomalous. In Internet of Things (IoT) networks where automated machine-to-machine communications predominate, the type of communication originated by Stuxnet will be highly visible.

The more human-triggered network communications are present in the observed traffic, the more difficult the anomaly detection becomes,

However, a word of warning is due: Stuxnet incorporates technology that was state-of-the-art more than ten years ago. Evolutions of Stuxnet like Duqu and Duqu2, but also recent malware variants like Gauss, BlackEnergy3, AdWind or Locky show that multi-layer obfuscation and encryption will become the standard for advanced malware. Moreover, malware like, e.g., Regin passively monitors the actual network traffic to select the least suspicious communication protocol as VPN tunnel for its command and control traffic. Therefore, network monitoring and subsequent anomaly detection systems will be challenged to identify anomalies in encrypted and obfuscated network traffic.

#### 4. Future work

This draft aims at defining the basic framework that advanced anomaly detection methods will build upon. Plans and ongoing work include the definition of metrics and methodologies to rate malware communications, protocols, and interfaces to applications. As an example a malware's adopted scanning strategy is commonly related to its propagation speed. On one hand, aggressive probing by a malware discovers a higher number of potential victims within a shorter time, increasing the malware's speed and likelihood of propagation. The cost of this propagation speed is an increased scanning traffic that results in malware activity being detectable through network monitoring. On the other hand, passive listening malware may spend long periods of time unobserved in a system, monitoring and learning its environment while waiting for activation through potentially hidden communication channels. Discovery of such dormant persistent threats depends, therefore, on detection of highly sporadic, hidden activation signals in almost real-time.

#### 5. Acknowledgements

Thanks to Kirsty P., Sage B., and Tanja Zseby for their comments that helped substantially in scoping, structuring and wording the initial version of this draft.

#### 6. IANA Considerations

This memo includes no request to IANA.

All drafts are required to have an IANA considerations section (see the update of RFC 2434 [I-D.narten-iana-considerations-rfc2434bis] for a guide). If the draft does not require IANA to do anything, the section contains an explicit statement that this is the case (as

above). If there are no requirements for IANA, the section will be removed during conversion into an RFC by the RFC Editor.

## 7. Security Considerations

All drafts are required to have a security considerations section. See RFC 3552 [RFC3552] for a guide.

## 8. References

### 8.1. 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, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.

### 8.2. Informative References

- [CBCS] Rechtin, E., "Systems Architecting: Creating and Building Complex Systems", Prentice Hall ISBN-13: 978-0138803452, 1991, 352 pages, 1991.
- [GML] Eder-Neuhauser, P., Zseby, T., Fabini, J., and G. Vormayr, "Cyber Attack Models for Smart Grid Environments", Elsevier Sustainable Energy, Grids and Networks Volume 12, 2017, pp 10-29, December 2017.  
  
Pre-published version available for download at  
[https://publik.tuwien.ac.at/files/publik\\_261089.pdf](https://publik.tuwien.ac.at/files/publik_261089.pdf)
- [I-D.mcfadden-smart-endpoint-taxonomy-for-cless] McFadden, M., "Endpoint Taxonomy for CLESS", draft-mcfadden-smart-endpoint-taxonomy-for-cless-00 (work in progress), July 2019.
- [I-D.narten-iana-considerations-rfc2434bis] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", draft-narten-iana-considerations-rfc2434bis-09 (work in progress), March 2008.

[Stuxnet] Falliere, N., O Murchu, L., and E. Chien, "W32.Stuxnet Dossier", February 2011.

URL:  
[https://www.symantec.com/content/en/us/enterprise/media/security\\_response/whitepapers/w32\\_stuxnet\\_dossier.pdf](https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf)

Author's Address

Joachim Fabini  
TU Wien  
Gusshausstrasse 25/E389  
Vienna 1040  
AT

Phone: +43 1 58801 38813  
Email: Joachim.Fabini@tuwien.ac.at

LAMPS  
Internet-Draft  
Intended status: Standards Track  
Expires: 12 August 2022

M. Ounsworth  
Entrust  
M. Pala  
CableLabs  
8 February 2022

Composite Signatures For Use In Internet PKI  
draft-ounsworth-pq-composite-sigs-06

Abstract

With the widespread adoption of post-quantum cryptography will come the need for an entity to possess multiple public keys on different cryptographic algorithms. Since the trustworthiness of individual post-quantum algorithms is at question, a multi-key cryptographic operation will need to be performed in such a way that breaking it requires breaking each of the component algorithms individually. This requires defining new structures for holding composite signature data.

This document defines the structures `CompositeSignatureValue`, and `CompositeParams`, which are sequences of the respective structure for each component algorithm. This document also defines processes for generating and verifying composite signatures. This document makes no assumptions about what the component algorithms are, provided that their algorithm identifiers and signature generation and verification processes are defined.

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 12 August 2022.

## Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Terminology . . . . .	3
2. Composite Identifiers and Structures . . . . .	4
2.1. Algorithm Identifier . . . . .	5
2.2. Composite Keys . . . . .	5
2.2.1. Key Usage Bits . . . . .	5
2.3. Composite Signature . . . . .	6
2.4. Encoding Rules . . . . .	6
3. Composite Signature Processes . . . . .	7
3.1. Composite Signature Generation Process . . . . .	7
3.2. Composite-OR Signature Generation Process . . . . .	8
3.3. Composite Signature Verification Process . . . . .	9
3.4. Composite-OR Signature Verification . . . . .	11
3.4.1. Composite-OR Legacy Mode . . . . .	11
4. In Practice . . . . .	12
4.1. Cryptographic protocols . . . . .	13
5. IANA Considerations . . . . .	14
6. Security Considerations . . . . .	14
6.1. Policy for Deprecated and Acceptable Algorithms . . . . .	14
7. Appendices . . . . .	14
7.1. ASN.1 Module . . . . .	14
7.2. Intellectual Property Considerations . . . . .	16
8. Contributors and Acknowledgements . . . . .	16
8.1. Making contributions . . . . .	17
9. Normative References . . . . .	17
Authors' Addresses . . . . .	18

## 1. Introduction

During the transition to post-quantum cryptography, there will be uncertainty as to the strength of cryptographic algorithms; we will no longer fully trust traditional cryptography such as RSA, Diffie-Hellman, DSA and their elliptic curve variants, but we will also not fully trust their post-quantum replacements until they have had sufficient scrutiny. Unlike previous cryptographic algorithm migrations, the choice of when to migrate and which algorithms to migrate to, is not so clear. Even after the migration period, it may be advantageous for an entity's cryptographic identity to be composed of multiple public-key algorithms.

The deployment of composite signatures using post-quantum algorithms will face two challenges

- \* Algorithm strength uncertainty: During the transition period, some post-quantum signature and encryption algorithms will not be fully trusted, while also the trust in legacy public key algorithms will start to erode. A relying party may learn some time after deployment that a public key algorithm has become untrustworthy, but in the interim, they may not know which algorithm an adversary has compromised.
- \* Backwards compatibility: During the transition period, post-quantum algorithms will not be supported by all clients.

This document provides a mechanism to address algorithm strength uncertainty by building on `~~ reference draft-ounsworth-pq-composite-pubkeys ~~` by providing formats for encoding multiple signature values into existing public signature fields, as well as the process for validating a composite signature. Backwards compatibility is addressed via the Composite-OR mechanism described herein.

This document is intended for general applicability anywhere that digital signatures are used within PKIX and CMS structures.

### 1.1. Terminology

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.

The following terms are used in this document:



**ALGORITHM:** An information object class for identifying the type of cryptographic operation to be performed. This document is primarily concerned with algorithms for producing digital signatures.

**BER:** Basic Encoding Rules (BER) as defined in [X.690].

**COMPONENT ALGORITHM:** A single basic algorithm which is contained within a composite algorithm.

**COMPOSITE ALGORITHM:** An algorithm which is a sequence of two or more component algorithms, as defined in Section 2.

**DER:** Distinguished Encoding Rules as defined in [X.690].

**LEGACY:** For the purposes of this document, a legacy key or signature is a non-composite key or signature.

**PUBLIC / PRIVATE KEY:** The public and private portion of an asymmetric cryptographic key, making no assumptions about which algorithm.

**SIGNATURE:** A digital cryptographic signature, making no assumptions about which algorithm.

## 2. Composite Identifiers and Structures

In order for signatures to be composed of multiple algorithms, we define encodings consisting of a sequence of signature primitives (aka "component algorithms") such that these structures can be used as a drop-in replacement for existing signature fields such as those found in PKCS#10 [RFC2986], CMP [RFC4210], X.509 [RFC5280], CMS [RFC5652].

This section defines the following structures:

- \* The `id-alg-composite` is an `AlgorithmIdentifier` identifying a composite signature object.

The `sa-CompositeSignature` `AlgorithmIdentifier` and the corresponding `CompositeParams` identify the algorithm(s) used in a composite signature.

- \* The `CompositeSignatureValue`, carries a sequence of signatures that are generated by a `CompositePrivateKey`, and can be verified with the corresponding `CompositePublicKey`.

**EDNOTE 2:** the choice to define composite algorithm parameters as a sequence inside the existing fields avoids the exponential proliferation of OIDs that are needed for each combination of

signature algorithms in other schemes for achieving multi-key certificates. This scheme also naturally extends from 2-keypair to n-keypair keys and certificates.

EDNOTE 2a: We have heard community feedback that the ASN.1 structures presented here are too flexible in that allow arbitrary combinations of an arbitrary number of signature algorithms. The feedback is that this is too much of a "footgun" for implementors and sysadmins. We are working on an alternative formulation using ASN.1 information object classes that allow for compiling explicit pairs of algorithmIDs. We would love community feedback on which approach is preferred. See slide 30 of this presentation: <https://datatracker.ietf.org/meeting/interim-2021-lamps-01/materials/slides-interim-2021-lamps-01-sessa-position-presentation-by-mike-ounsworth-00.pdf>

## 2.1. Algorithm Identifier

The following object identifier is used for identifying a composite signature. Additional encoding information is provided below for each of these objects.

```
id-alg-composite OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1) private(4)
    enterprise(1) OpenCA(18227) Algorithms(2) id-alg-composite(1) }
```

EDNOTE 3: this is a temporary OID for the purposes of prototyping. We are requesting IANA to assign a permanent OID, see Section 5.

## 2.2. Composite Keys

A Composite signature MUST be associated with a Composite public key as defined in `reference draft-ounsworth-pq-composite-pubkey`.

### 2.2.1. Key Usage Bits

For protocols such as X.509 [RFC5280] that specify key usage along with the public key, then the composite public key associated with a composite signature MUST have a signing-type key usage.

If the keyUsage extension is present in a Certification Authority (CA) certificate that indicates id-composite-key, then any combination of the following values MAY be present:

```
digitalSignature;
nonRepudiation;
keyCertSign; and
cRLSign.
```

If the keyUsage extension is present in an End Entity (EE) certificate that indicates id-composite-key, then any combination of the following values MAY be present:

digitalSignature; and  
nonRepudiation;

### 2.3. Composite Signature

The ASN.1 algorithm object for a composite signature is:

```
sa-CompositeSignature SIGNATURE-ALGORITHM ::= {  
  IDENTIFIER id-alg-composite  
  VALUE CompositeSignatureValue  
  PARAMS TYPE CompositeParams ARE required  
  PUBLIC-KEYS { pk-Composite }  
  SMIME-CAPS { IDENTIFIED BY id-alg-composite } }  
}
```

The following algorithm parameters MUST be included:

CompositeParams ::= SEQUENCE SIZE (2..MAX) OF AlgorithmIdentifier

The signature's CompositeParams sequence MUST contain the same component algorithms listed in the same order as in the associated CompositePrivateKey and CompositePublicKey.

The output of the composite signature algorithm is the DER encoding of the following structure:

CompositeSignatureValue ::= SEQUENCE SIZE (2..MAX) OF BIT STRING

Where each BIT STRING within the SEQUENCE is a signature value produced by one of the component keys. It MUST contain one signature value produced by each component algorithm, and in the same order as in the associated CompositeParams object.

The choice of SEQUENCE OF BIT STRING, rather than for example a single BIT STRING containing the concatenated signature values, is to gracefully handle variable-length signature values by taking advantage of ASN.1's built-in length fields.

### 2.4. Encoding Rules

Many protocol specifications will require that composite signature data structures be represented by an octet string or bit string.

When an octet string is required, the DER encoding of the composite data structure SHALL be used directly.

When a bit string is required, the octets of the DER encoded composite data structure SHALL be used as the bits of the bit string, with the most significant bit of the first octet becoming the first bit, and so on, ending with the least significant bit of the last octet becoming the last bit of the bit string.

In the interests of simplicity and avoiding compatibility issues, implementations that parse these structures MAY accept both BER and DER.

### 3. Composite Signature Processes

This section specifies the processes for generating and verifying composite signatures.

This process addresses algorithm strength uncertainty by providing the verifier with parallel signatures from all the component signature algorithms; thus breaking the composite signature would require breaking all of the component signatures.

#### 3.1. Composite Signature Generation Process

Generation of a composite signature involves applying each component algorithm's signature process to the input message according to its specification, and then placing each component signature value into the CompositeSignatureValue structure defined in Section 2.3.

The following process is used to generate composite signature values.

## Input:

K1, K2, ..., Kn      Private keys for the n component signature algorithms, a CompositePrivateKey  
M                      Message to be signed, an octet string

## Output:

S                      The signatures, a CompositeSignatureValue

## Signature Generation Process:

1. Generate the n component signatures independently, according to their algorithm specifications.  
  
    for i := 1 to n  
        Si := Sign( Ki, M )
2. Encode each component signature S1, S2, ..., Sn into a BIT STRING according to its algorithm specification.  
  
    S ::= Sequence { S1, S2, ..., Sn }
3. Output S

Since recursive composite public keys are disallowed in ~~ Reference draft-ounsworth-pq-composite-pubkeys sec-composite-pub-keys ~~, no component signature may itself be a composite; ie the signature generation process MUST fail if one of the private keys K1, K2, ..., Kn is a composite with the OID id-alg-composite.

A composite signature MUST produce and include in the output a signature value for every component key in the corresponding CompositePrivateKey. For this mode, please see Composite-OR in section Section 3.2.

### 3.2. Composite-OR Signature Generation Process

EDNOTE: This section was written with the intention of keeping the primary Composite OID reserved for the simple and strict mode; if you want to do either a simple OR, or a custom policy then we have given a different OID. We are still debating whether this is useful to specify at issuing time, or whether this is adding needless complexity to the draft.

If the algorithm ID of the public key associated with this signature is id-composite-or-key then the signer MAY use only a subset of the component keys and therefore produce fewer signatures than the number of component keys.

Composite-OR signature generation uses the same structures and algorithms as Composite, with the difference that the signature generation process may emit a null instead of a signature value in step 1 for one or more component algorithms. A Composite-OR signature MUST NOT be entirely null; it must contain at least one valid signature.

The design intent of this mode is to support migration scenarios where an end entity has been issued keys on algorithms that either itself or the peer with which it is communicating do not (yet) support. This design allows for both the mode where the signer omits signatures that it knows its peer cannot process in order to save bandwidth and performance, and the mode where it includes all component signatures and allows the verifier to choose how many to verify. The latter is RECOMMENDED for signatures that need both short-term backwards compatibility as well as long-term security.

EDNOTE: Do we want to allow a Composite-OR with only a single signature to produce non-composite signatureAlgorithm and signatureValue as per [RFC5280]? Advantages: bandwidth savings of an extra OID and some sequences with one element. Disadvantages: ambiguous whether a signature is traditional or composite until you look at the corresponding public key.

### 3.3. Composite Signature Verification Process

Verification of a composite signature involves applying each component algorithm's verification process according to its specification.

In the absence of an application profile specifying otherwise, compliant applications MUST output "Valid signature" (true) if and only if all component signatures were successfully validated, and "Invalid signature" (false) otherwise.

The following process is used to perform this verification.

## Input:

P     Signer's composite public key  
M     Message whose signature is to be verified, an octet string  
S     Composite Signature to be verified  
A     Composite Algorithm identifier

## Output:

Validity         "Valid signature" (true) if the composite signature  
                  is valid, "Invalid signature" (false) otherwise.

## Signature Verification Procedure::

1. Parse P, S, A into the component public keys, signatures,  
and algorithm identifiers

P1, P2, ..., Pn := Desequence( P )  
S1, S2, ..., Sn := Desequence( S )  
A1, A2, ..., An := Desequence( A )

If Error during Desequencing, or the three sequences have  
different numbers of elements, or any of the public keys P1, P2, ..., Pn or  
algorithm identifiers A1, A2, ..., An are composite with the OID  
id-alg-composite then output "Invalid signature" and stop.

2. Check each component signature individually, according to its  
algorithm specification.  
If any fail, then the entire signature validation fails.

for i := 1 to n  
  if not verify( Pi, M, Si ), then  
    output "Invalid signature"

if all succeeded, then  
  output "Valid signature"

Since recursive composite public keys are disallowed in ~~ Reference  
draft-ounsworth-pq-composite-keys sec-composite-pub-keys ~~, no  
component signature may be composite; ie the signature verification  
procedure MUST fail if any of the public keys P1, P2, ..., Pn or  
algorithm identifiers A1, A2, ..., An are composite with the OID id-  
alg-composite.

### 3.4. Composite-OR Signature Verification

EDNOTE: This section was written with the intention of keeping the primary Composite OID reserved for the simple and strict mode; if you want to do either a simple OR, or a custom policy then we have given a different OID. We are still debating whether this is useful to specify at issuing time, or whether this is adding needless complexity to the draft.

When the public key associated with the signature being verified has algorithm id-composite-or-key, then an alternate verification processes MAY be used, at the discretion of the implementor. In this section we provide some examples of alternate verification processes.

If the signature is a traditional (non-composite) algorithm and value or a composite signature with a single component, then it MAY be considered valid if it verifies under one of the component keys.

If the signature is composite, then the implementor MAY implement policy for which combinations are acceptable.

EDNOTE: Does this mean Composite-OR end entity certificates need to be issued by a PKI that is marked as Composite-OR all the way to the top so that verifiers that do not support all the algorithms don't fail? Need to think more about the security implications of allowing a Composite-or in an end entity cert implicitly turning all Composite algIDs into Composite-or algIDs in its cert chain.

EDNOTE: Do we need to specify the semantics of verifying an "n of m" subset signature? I suspect that specifying this in general will be a rat's nest of edge cases, so I propose to "leave this to the implementor".

#### 3.4.1. Composite-OR Legacy Mode

The Composite-OR Legacy Mode is provided to facilitate migration by allowing existing PKI entities (including root CAs, intermediate CAs, and end entities) to have their existing keys re-certified inside a Composite-OR structure along with Post-Quantum keys, and for signatures made by that key prior to the migration to remain valid. Note that Composite-OR Legacy Mode is only provided for signature verification, and not for signature generation; legacy signatures SHOULD NOT be produced from a Composite key.

EDNOTE: to further solidify this, we could add a clause that Legacy Mode signatures are to fail if the signature was produced after notBefore date of the Composite-OR certificate?



In Composite-OR Legacy Mode, a legacy signature algorithm and legacy signature value MAY be validated against a Composite-OR public key. The legacy signature algorithm is to be interpreted by the verifier as a sa-CompositeSignature with CompositeParams in the following way:

```
CompositeParams {legacyAlgorithmIdentifier, null, ..., null}
```

with the correct number of nulls to match the Composite-OR public key that the signature is being verified against. For the purposes of a signature validation under Composite-OR Legacy Mode, a null AlgorithmIdentifier is considered to be a match for the corresponding algorithm in the Composite-OR public key.

The legacy signature value is to be interpreted by the verifier as a sa-CompositeSignature with CompositeParams in the following way:

```
CompositeSignatureValue {legacySignatureValue, null, ..., null}
```

with the correct number of nulls to match the Composite-OR public key that the signature is being verified against. The verification algorithm in section Section 3.4 applies.

Security consideration: when implementing Composite-OR Legacy Mode, it is important to catch the edge case of {null, null, ..., null} for both AlgorithmIdentifier and SignatureValue and return Invalid Signature.

It is RECOMMENDED that Composite-OR Legacy Mode be implemented as an optional mode in the verifier that can be enabled or disabled by runtime configuration or policy.

EDNOTE: the signing public key is often identified in the signed document by issuer+serialNumber or by an SKI containing a hash of the public key value. Might need X.509 extensions identifying the SKI of the legacy cert it's replacing?

#### 4. In Practice

This section addresses practical issues of how this draft affects other protocols and standards.

~~~ BEGIN EDNOTE 10~~~

EDNOTE 10: Possible topics to address:

- \* The size of these certs and cert chains.

- \* In particular, implications for (large) composite keys / signatures / certs on the handshake stages of TLS and IKEv2.
- \* If a cert in the chain is a composite cert then does the whole chain need to be of composite Certs?
- \* We could also explain that the root CA cert does not have to be of the same algorithms. The root cert SHOULD NOT be transferred in the authentication exchange to save transport overhead and thus it can be different than the intermediate and leaf certs.
- \* We could talk about overhead (size and processing).
- \* We could also discuss backwards compatibility.
- \* We could include a subsection about implementation considerations.

~~~ END EDNOTE 10~~~

#### 4.1. Cryptographic protocols

This section talks about how protocols like (D)TLS and IKEv2 are affected by this specifications. It will not attempt to solve all these problems, but it will explain the rationale, how things will work and what open problems need to be solved. Obvious issues that need to be discussed.

- \* How does the protocol declare support for composite signatures? TLS has hooks for declaring support for specific signature algorithms, however it would need to be extended, because the client would need to declare support for both the composite infrastructure, as well as for the various component signature algorithms.
- \* How does the protocol use the multiple keys. The obvious way would be to have the server sign using its composite public key; is this sufficient.
- \* Overhead; including certificate size, signature processing time, and size of the signature.
- \* How to deal with crypto protocols that use public key encryption algorithms; this document only lists how to work with signature algorithms. Encoding composite public keys is straightforward; encoding composite ciphertexts is less so - we decided to put that off to another draft.

## 5. IANA Considerations

The ASN.1 module OID is TBD. The id-alg-composite OID is to be assigned by IANA. The authors suggest that IANA assign an OID on the id-pkix arc:

```
id-alg-composite OBJECT IDENTIFIER ::= {  
    iso(1) identified-organization(3) dod(6) internet(1) security(5)  
    mechanisms(5) pkix(7) algorithms(6) composite(??) }
```

## 6. Security Considerations

### 6.1. Policy for Deprecated and Acceptable Algorithms

Traditionally, a public key, certificate, or signature contains a single cryptographic algorithm. If and when an algorithm becomes deprecated (for example, RSA-512, or SHA1), it is obvious that structures using that algorithm are implicitly revoked.

In the composite model this is less obvious since a single public key, certificate, or signature may contain a mixture of deprecated and non-deprecated algorithms. Moreover, implementers may decide that certain cryptographic algorithms have complementary security properties and are acceptable in combination even though neither algorithm is acceptable by itself.

Specifying a modified verification algorithm to handle these situations is beyond the scope of this draft, but could be desirable as the subject of an application profile document, or to be up to the discretion of implementers.

2. Check policy to see whether A1, A2, ..., An constitutes a valid combination of algorithms.

```
if not checkPolicy(A1, A2, ..., An), then  
    output "Invalid signature"
```

While intentionally not specified in this document, implementors should put careful thought into implementing a meaningful policy mechanism within the context of their signature verification engines, for example only algorithms that provide similar security levels should be combined together.

## 7. Appendices

### 7.1. ASN.1 Module

<CODE STARTS>

Composite-Signatures-2019  
{ TBD }

DEFINITIONS IMPLICIT TAGS ::= BEGIN

EXPORTS ALL;

IMPORTS

PUBLIC-KEY, SIGNATURE-ALGORITHM  
FROM AlgorithmInformation-2009 -- RFC 5912 [X509ASN1]  
{ iso(1) identified-organization(3) dod(6) internet(1)  
security(5) mechanisms(5) pkix(7) id-mod(0)  
id-mod-algorithmInformation-02(58) }

SubjectPublicKeyInfo  
FROM PKIX1Explicit-2009  
{ iso(1) identified-organization(3) dod(6) internet(1)  
security(5) mechanisms(5) pkix(7) id-mod(0)  
id-mod-pkix1-explicit-02(51) }

OneAsymmetricKey  
FROM AsymmetricKeyPackageModuleV1  
{ iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)  
pkcs-9(9) smime(16) modules(0)  
id-mod-asymmetricKeyPkgV1(50) } ;

--  
-- Object Identifiers  
--

id-alg-composite OBJECT IDENTIFIER ::= { TBD }

--  
-- Public Key  
--

pk-Composite PUBLIC-KEY ::= {  
IDENTIFIER id-alg-composite  
KEY CompositePublicKey  
PARAMS ARE absent  
PRIVATE-KEY CompositePrivateKey  
}

CompositePublicKey ::= SEQUENCE SIZE (2..MAX) OF SubjectPublicKeyInfo

CompositePrivateKey ::= SEQUENCE SIZE (2..MAX) OF OneAsymmetricKey

```
--  
-- Signature Algorithm  
--  
sa-CompositeSignature SIGNATURE-ALGORITHM ::= {  
    IDENTIFIER id-alg-composite  
    VALUE CompositeSignatureValue  
    PARAMS TYPE CompositeParams ARE required  
    PUBLIC-KEYS { pk-Composite }  
    SMIME-CAPS { IDENTIFIED BY id-alg-composite } }  
  
CompositeParams ::= SEQUENCE SIZE (2..MAX) OF AlgorithmIdentifier  
  
CompositeSignatureValue ::= SEQUENCE SIZE (2..MAX) OF BIT STRING  
  
END  
  
<CODE ENDS>
```

## 7.2. Intellectual Property Considerations

The following IPR Disclosure relates to this draft:

<https://datatracker.ietf.org/ipr/3588/>

## 8. Contributors and Acknowledgements

This document incorporates contributions and comments from a large group of experts. The Editors would especially like to acknowledge the expertise and tireless dedication of the following people, who attended many long meetings and generated millions of bytes of electronic mail and VOIP traffic over the past year in pursuit of this document:

John Gray (Entrust), Serge Mister (Entrust), Scott Fluhrer (Cisco Systems), Panos Kampanakis (Cisco Systems), Daniel Van Geest (ISARA), Tim Hollebeek (Digicert), and Francois Rousseau.

We are grateful to all, including any contributors who may have been inadvertently omitted from this list.

This document borrows text from similar documents, including those referenced below. Thanks go to the authors of those documents.

"Copying always makes things easier and less error prone" -  
[RFC8411].

### 8.1. Making contributions

Additional contributions to this draft are welcome. Please see the working copy of this draft at, as well as open issues at:

<https://github.com/EntrustCorporation/draft-ounsworth-composite-sigs>

## 9. 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, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, DOI 10.17487/RFC2986, November 2000, <<https://www.rfc-editor.org/info/rfc2986>>.
- [RFC4210] Adams, C., Farrell, S., Kause, T., and T. Mononen, "Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)", RFC 4210, DOI 10.17487/RFC4210, September 2005, <<https://www.rfc-editor.org/info/rfc4210>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <<https://www.rfc-editor.org/info/rfc5652>>.
- [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>>.
- [RFC8411] Schaad, J. and R. Andrews, "IANA Registration for the Cryptographic Algorithm Object Identifier Range", RFC 8411, DOI 10.17487/RFC8411, August 2018, <<https://www.rfc-editor.org/info/rfc8411>>.
- [X.690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2015, November 2015.

Authors' Addresses

Mike Ounsworth  
Entrust Limited  
2500 Solandt Road -- Suite 100  
Ottawa, Ontario K2K 3G5  
Canada

Email: [mike.ounsworth@entrust.com](mailto:mike.ounsworth@entrust.com)

Massimiliano Pala  
CableLabs

Email: [director@openca.org](mailto:director@openca.org)

LAMPS  
Internet-Draft  
Intended status: Informational  
Expires: March 20, 2020

M. Ounsworth  
Entrust Datacard  
September 17, 2019

Post-quantum Multi-Key Mechanisms for PKIX-like protocols; Problem  
Statement and Overview of Solution Space  
draft-pq-pkix-problem-statement-01

Abstract

With the widespread adoption of post-quantum (PQ) cryptography will come uncertainty about the strength of cryptographic primitives. For example; when will RSA and ECC fall? Are Lattice schemes as strong as we believe? The cryptographic community is calling for hybrid schemes that combine classic and post-quantum crypto in ways that remain strong so long as at least one of the algorithms used remains strong.

This document defines the problem statement for digital signatures in PKIX-like protocols, and gives an overview of the general families of solutions.

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 March 20, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents



(<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

|   |   |
|---|---|
| 1. Problem Statement . . . . .                              | 2 |
| 1.1. Formal Security Requirements . . . . .                 | 2 |
| 2. Solution Space . . . . .                                 | 3 |
| 2.1. "Composite" concatenated keys and signatures . . . . . | 4 |
| 2.2. Application: X.509 . . . . .                           | 5 |
| 2.2.1. Multiple Certificate Chains . . . . .                | 5 |
| 2.2.2. "Hybrid" v3 extensions . . . . .                     | 6 |
| 3. Conclusion . . . . .                                     | 7 |
| 4. Intellectual Property Considerations . . . . .           | 8 |
| 5. Contributors and Acknowledgements . . . . .              | 8 |
| 6. Informative References . . . . .                         | 8 |
| Author's Address . . . . .                                  | 8 |

## 1. Problem Statement

In general terms, "hybrid" or "layered" signatures means that the document signer performs multiple, parallel, signatures over the document and provides them all to the verifier. The verifier's job is to check that all signatures are valid.

The general concept is straight-forward, but the devil is in the details.

### 1.1. Formal Security Requirements

A solution to the PQ multi-signature problem MUST meet the following "security" properties:

- o S1. In order to break the overall signature, an attacker must break all the signatures.
- o S2. Robust to stripping and substitution attacks, which in most cases reduces to the following statement: the verifier needs a way to know which and how many signature algorithms to expect on a given message.

A solution to the PQ multi-signature problem MAY meet the following "desirable" properties, depending on context of the protocol in which

the signature is being performed. And yes, some of these conflict with each other:

- o D1. The set of signing algorithms is negotiated dynamically between client and server.
- o D2. Since post-quantum public keys and signatures are large, they should only be sent when you know the client will verify them. For bonus points, the protocol should control when and how the large PQ data is transmitted. Note that all three schemes below could include a hash and external delivery of the large post-quantum data (for example, HTTP URL, delivered via protocol extension, etc). Since this trick is orthogonal to and applies equally to all three schemes, it is not considered here.
- o D3. Backwards compatibility: there is a mechanism either for the client to negotiate whether it wants multi-key signatures, or the signature mechanism is designed in such a way that a legacy verifier will ignore the PQ parts.
- o D4. Protocol compatibility: a multi-key solution should "drop-in" to existing protocol message formats. For example, a solution where any PKIX-like protocol simply needs to pick up new OIDs with no other code changes would meet this property. Note that this conflicts with D2 which (probably) requires protocol-level awareness of the multiple keys.
- o D5. A mechanism that supports 3 or more algorithms is desirable to hedge our bets against algorithm compromise; possibly allowing a set of public keys / signatures to continue being used so long as there remain an acceptable number of un-broken algorithms.
- o D6. Some applications may be limited to a single signing certificate and/or key without significant redesign (for example smart cards).

## 2. Solution Space

At the time of writing, there have been proposed three broad families of solutions being considered: concatenating multiple public keys and signatures together into a large single public key / signature object, multiple independent traditional certificate chains, and placing PQ data into v3 extensions. Since this discussion came out of LAMPS, the solutions that have drafts are X.509-focused, but we note that the first solution ("composite") is more general.

Each solution space is described below. I am trying to keep these abstract and not solution-specific.

### 2.1. "Composite" concatenated keys and signatures

Concatenate public keys, algorithmIDs, and signatureValues into "composite" versions of those structures.

#### Pros:

- o D4: "Drop-in" to most protocols because once the underlying crypto layer supports the composite primitive, then the protocol only needs to add support for the composite algorithmID.
- o D5: Trivially extends to 3 or more algorithms.
- o Complexity of composite signature generation and verification is moved to crypto layer, and therefore protocol designers and implementers do not need to worry about it. Note this is also a con, see below. Crypto layer maintainers have control of which combinations of algorithms are acceptable. This could be exposed to the application layer, for example, through config files or APIs.
- o D6: - D6: Single certificate and key object should be a near drop-in replacement for applications such as smart cards whose architectures limit them to a single certificate.

#### Cons:

- o D1: Does not apply well to negotiated protocols, at least not without a quadratic or exponential proliferation of certificates using every combination of algorithms (possibly equip servers with two certificates: a high-security certificate, and a high-compatibility one). Another option is to allow servers to perform signatures with a subset of keys in the certificate, or allow clients to selectively ignore some signatures. Note that this requires a complex verification algorithm which is easy to mis-implement, and makes it very difficult to detect stripping attacks. So this is probably not a good direction to go.
- o D2: All public keys and signatures are contained in the certificate, and therefore need to be transferred.
- o D3: Is not immediately backwards-compatible since clients need to be patched to understand the composite message structures and OID. But once that's done the verification algorithm can be designed to skip unknown algorithmIDs (this is not unique to this scheme, and violates D2, but it's possible \_shrug\_).

- o Policy moved to crypto layer; protocol can not (easily) be negotiate the algorithms, or control what counts as an "acceptable" combination of algorithms.

Neutral:

- o S1 & S2: Substitution and stripping attacks; if an attacker can break one or more of the signature algorithms, then they can strip out the other algorithmIDs and re-generate the signatures they have broken. This can be mitigated by requiring that a composite public key produces signatures using all keys. A verifier can check against their local trust store to see that the EndEntity certificate contains the same algorithms as its issuer.
- o D2: Large PQ data is always sent to the client. This can be either viewed as a bad thing (bandwidth) or a good thing (protect data now, patch clients later).

This family of solutions has been instantiated in [draft-ounsworth-pq-composite-sigs-01].

## 2.2. Application: X.509

### 2.2.1. Multiple Certificate Chains

If you want multiple public keys on multiple cryptographic algorithms, then get multiple certificates from multiple PKIs. The protocol has control of negotiating which algorithms get used, how to encapsulate the large PQ data, etc. In many ways, this is the most obvious solution.

Pros:

- o D1: Highly friendly to negotiated protocols since it allows the server to sign a transaction with the combination of algorithms that was negotiated with the client.
- o D2: Protocol has control of how and when to transmit the large PQ certificates and signatures.
- o During the RSA to ECC migration, many web servers added support for loading multiple certificates, and deciding which to use based on the TLS negotiation, so this work is already half done.
- o D3: Clients will only be served certificate algorithms that they have requested.

- o D4: Does not require any modification to PKI hierarchies. Does not require any changes to X.509, and only minor changes to CMS (for example, current CMS implementations have inconsistent behaviour when multiple SignerInfos are present).
- o D5: Trivially extends to 3 or more certificates on different algorithms.

Cons:

- o S1 & S2: Unclear how this applies to non-negotiated protocols.
- o S1 & S2: mitigation against stripping and substitution attacks is left up to the protocol, and is easy to get wrong. Possibly some kind of extension could be placed in root CA certs to indicate which "sister PKIs" a verifier can expect to see, but such a solution would be fairly complex.
- o D4: Requires updates to all protocols to add or specify the behaviour of multiple SignatureAlgorithm and SignatureValue fields.
- o D6: Will have compatibility issues with applications such as smart cards that are limited to a single signing certificate.

At time of writing, I am not aware of any internet drafts or other implementations of this family of solutions.

#### 2.2.2. "Hybrid" v3 extensions

Place the PQ data into X.509v3 extensions. This has two general forms; the "PQ extension" contains a complete second certificate, or the "PQ extensions" contain an alternative public key, signature algorithm, and signature value.

Pros:

- o D3: Highly backwards compatible; if the PQ extension(s) are marked as non-critical, then legacy clients will ignore the PQ data and verify the "outer" signature as normal.
- o D6: Single certificate should be compatible with applications such as smart cards whose architectures limit them to a single certificate. Though API / communication with the card would need to support two signatures or challenge-responses.

Cons:

- o D1: Does not apply well to negotiated protocols; at least not without a quadratic proliferation of certificates on every combination of algorithms.
- o D3: May be too backwards-compatible; in a large PKI deployment, it is very hard to be know if/when clients are all using the PQ data.
- o D4: Requires updates to all protocols to add or specify the behaviour of multiple SignatureAlgorithm and SignatureValue fields.
- o D5: Does not extend to 3 or more keys + signatures on its own, but could be used in combination with the "composite" solution to do so.

Neutral:

- o S1 & S2: Substitution and stripping attacks; if an attacker can break the outer signature (assumed to be RSA for backwards compatibility reasons), then they can strip or substitute the PQ extension and re-generate the outer signature. This can be mitigated by requiring that a hybrid CA containing an alt public key always produces hybrid certificates containing a corresponding alt signature. When building a certificate chain, a verifier can check that alt signatures are present and valid all the way down the certificate chain.
- o D2: Large PQ data is always sent to the client. This can be either viewed as a bad thing (bandwidth) or a good thing (protect data now, patch clients later).

This family of solutions has been instantiated in [draft-truskovsky-lamps-pq-hybrid-x509-01], and has a related standard currently before the ITU.

### 3. Conclusion

None of these solution families are a panacea. Multi-cert looks preferable for online negotiated protocols, hybrid looks preferable for environments with strong backwards compatibility requirements for legacy clients, and composite looks preferable for controlled environments where you know the clients will support the composite message format.

#### 4. Intellectual Property Considerations

Hybrid certificates, specifically [draft-truskovsky-lamps-pq-hybrid-x509-01] has IPR held by ISARA which has an IPR statement available at <https://datatracker.ietf.org/ipr/3287/>.

Composite certificates, specifically [draft-ounsworth-pq-composite-sigs-01] has IPR held by Max Pala and CableLabs, but with open license terms, available at <https://datatracker.ietf.org/ipr/3481/>.

#### 5. Contributors and Acknowledgements

This document incorporates contributions and comments from a large group of experts, including the authors list of [draft-ounsworth-pq-composite-sigs-01] and hallway chats at IETF105 and other crypto conferences.

This document borrows text from similar documents, including those referenced below. Thanks go to the authors of those documents. "Copying always makes things easier and less error prone" - [RFC8411].

#### 6. Informative References

- [I-D.ounsworth-pq-composite-sigs]  
Ounsworth, M. and M. Pala, "Composite Keys and Signatures For Use In Internet PKI", draft-ounsworth-pq-composite-sigs-01 (work in progress), July 2019.
- [I-D.truskovsky-lamps-pq-hybrid-x509]  
Truskovsky, A., Geest, D., Fluhrer, S., Kampanakis, P., Ounsworth, M., and S. Mister, "Multiple Public-Key Algorithm X.509 Certificates", draft-truskovsky-lamps-pq-hybrid-x509-01 (work in progress), August 2018.
- [RFC8411] Schaad, J. and R. Andrews, "IANA Registration for the Cryptographic Algorithm Object Identifier Range", RFC 8411, DOI 10.17487/RFC8411, August 2018, <<https://www.rfc-editor.org/info/rfc8411>>.

Author's Address

Mike Ounsworth  
Entrust Datacard Limited  
1000 Innovation Drive  
Ottawa, Ontario K2K 1E3  
Canada

Email: [mike.ounsworth@entrustdatacard.com](mailto:mike.ounsworth@entrustdatacard.com)



Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: May 6, 2020

A. Davidson  
Cloudflare Portugal  
N. Sullivan  
Cloudflare  
November 03, 2019

The Privacy Pass Protocol  
draft-privacy-pass-00

Abstract

This document specifies the Privacy Pass protocol for anonymously authorizing clients with services on the Internet.

Note to Readers

Source for this draft and an issue tracker can be found at <https://github.com/grittygrease/draft-privacy-pass> [1].

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 May 6, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

|   |    |
|---|----|
| 1. Introduction . . . . .                               | 3  |
| 1.1. Terminology . . . . .                              | 3  |
| 1.2. Preliminaries . . . . .                            | 4  |
| 1.2.1. Elliptic curve points . . . . .                  | 4  |
| 1.2.2. Protocol messages . . . . .                      | 4  |
| 1.3. Layout . . . . .                                   | 4  |
| 1.4. Requirements . . . . .                             | 5  |
| 2. Generalized protocol overview . . . . .              | 5  |
| 2.1. Key initialisation phase . . . . .                 | 6  |
| 2.2. Issuance phase . . . . .                           | 8  |
| 2.3. Redemption phase . . . . .                         | 9  |
| 2.3.1. Double-spend protection . . . . .                | 10 |
| 2.3.2. Finalization during redemption . . . . .         | 11 |
| 2.4. Error types . . . . .                              | 11 |
| 3. Key registration . . . . .                           | 11 |
| 3.1. Key rotation . . . . .                             | 12 |
| 3.2. Client retrieval . . . . .                         | 13 |
| 3.3. Key revocation . . . . .                           | 14 |
| 3.4. VOPRF ciphersuites . . . . .                       | 14 |
| 3.5. ECDSA key material . . . . .                       | 15 |
| 4. Protocol configurations . . . . .                    | 15 |
| 4.1. Single-Issuer Single-Verifier . . . . .            | 15 |
| 4.2. Single-Issuer Forwarding-Verifier . . . . .        | 16 |
| 4.3. Single-Issue Asynchronous-Verifier . . . . .       | 16 |
| 4.4. Bounded-Issuers . . . . .                          | 17 |
| 4.4.1. Fixing the bound . . . . .                       | 17 |
| 5. Privacy considerations . . . . .                     | 17 |
| 5.1. User segregation . . . . .                         | 18 |
| 5.1.1. Key rotation . . . . .                           | 18 |
| 5.1.2. Large numbers of issuers . . . . .               | 18 |
| 5.2. Tracking and identity leakage . . . . .            | 20 |
| 6. Security considerations . . . . .                    | 20 |
| 6.1. Double-spend protection . . . . .                  | 20 |
| 6.2. Key rotation . . . . .                             | 20 |
| 6.3. Token exhaustion . . . . .                         | 21 |
| 7. Summary of privacy and security parameters . . . . . | 21 |
| 7.1. Justification . . . . .                            | 22 |
| 7.2. Example parameterization . . . . .                 | 23 |
| 8. References . . . . .                                 | 23 |
| 8.1. Normative References . . . . .                     | 23 |
| 8.2. URIs . . . . .                                     | 24 |
| Authors' Addresses . . . . .                            | 24 |

## 1. Introduction

In some situations, it may only be necessary to check that a client has been previously authorized by a service; without learning any other information. Such lightweight authorization mechanisms can be useful in quickly assessing the reputation of a client in latency-sensitive communication.

The Privacy Pass protocol was initially introduced as a mechanism for authorizing clients that had already been authorized in the past, without compromising their privacy [DGSTV18]. This document seeks to standardize the usage and parametrization of the protocol.

The Internet performance company Cloudflare has already implemented server-side support for an initial version of the Privacy Pass protocol [PPSRV]. This support allows clients to bypass security mechanisms, providing that they have successfully passed these mechanisms previously. There is also a client-side implementation in the form of a browser extension that interacts with the Cloudflare network [PPEXT].

The main security requirement of the Privacy Pass protocol is to ensure that previously authenticated clients do not reveal their identity on reauthorization. The protocol uses a cryptographic primitive known as a verifiable oblivious pseudorandom function (VOPRF) for implementing the authorization mechanism. The VOPRF is implemented using elliptic curves and is currently in a separate standardization process [OPRF]. The protocol is split into three stages. The first two stages, initialisation and evaluation, are essentially equivalent to the VOPRF setup and evaluation phases from [OPRF]. The final stage, redemption, essentially amounts to revealing the client's secret inputs in the VOPRF protocol. The security (pseudorandomness) of the VOPRF protocol means that the client retains their privacy even after revealing this data.

In this document, we will give a formal specification of the Privacy Pass protocol to be used in settings that require high performance. We will specify the necessary cryptographic operations required by the underlying VOPRF, along with recommendations on how to perform key rotation

### 1.1. Terminology

The following terms are used throughout this document.

- o PRF: Pseudorandom function
- o VOPRF: Verifiable oblivious PRF [OPRF]

- o Server: A service that provides access to a certain resource (typically denoted S)
- o Client: An entity that seeks authorization from a server (typically denoted C)
- o Key: Server VOPRF key
- o Commitment: Corresponding public key to server's VOPRF key.

## 1.2. Preliminaries

Throughout this draft, let  $D$  be some object corresponding to an opaque data type (such as a group element). We write  $\text{bytes}(D)$  to denote the encoding of this data type as raw bytes (octet strings). We assume that such objects can also be interpreted as Buffer objects, with each internal slot in the buffer set to the value of the one of the bytes. For two objects  $x$  and  $y$ , we denote the concatenation of the bytes of these objects by  $(\text{bytes}(x) \parallel \text{bytes}(y))$ . We assume that all bytes are first base64-encoded before they are sent as part of a protocol message.

We use the notation " $[T_i]$ " to indicate an array of objects  $T_1, \dots, T_Q$  where the size of the array is  $Q$ , and the size of  $Q$  is implicit from context.

### 1.2.1. Elliptic curve points

When encoding elliptic curve points into existing data structures or into protocol messages, we assume that the curve points are first encoded into bytes. We allow both uncompressed and compressed encodings, as long as the client and server are aligned on the encodings that they used. Compressed encodings provide storage and communication benefits but are slightly more expensive to decode.

### 1.2.2. Protocol messages

Protocol messages can either be encoded in raw byte format, as base64-encoded string objects, or as JSON objects where all strings are represented in base64-encoded format.

## 1.3. Layout

- o Section 2: A generic overview of the Privacy Pass protocol based on VOPRFs.

- o Section 3: Describes the format of trusted registries that are used for holding public key commitments for each of the Privacy Pass issuers.
- o Section 4: Details different configurations for using the Privacy Pass protocol.
- o Section 5: Privacy considerations and recommendations arising from the usage of the Privacy Pass protocol.
- o Section 6: Additional security considerations to prevent abuse of the protocol from a malicious client.
- o Section 7: A summary of recommended parameter settings for ensuring privacy and security features of the protocol.

#### 1.4. Requirements

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 [RFC2119].

#### 2. Generalized protocol overview

In this document, we will be assuming that a client (C) is attempting to authenticate itself in a lightweight manner to a server (S). The authorization mechanism should not reveal to the server anything about the client; in addition, the client should not be able to forge valid credentials in situations where it does not possess any.

In this section, we will give a broad overview of how the Privacy Pass protocol functions in achieving these goals. The generic protocol can be split into three phases: initialisation, issuance and redemption. As we mentioned previously, the first two stages are essentially identical to the setup and evaluation phases of the VOPRF in [OPRF]. The last stage, redemption, corresponds to the client revealing their secret input data during the VOPRF protocol to the server. The server can use this data to confirm that the client has a valid VOPRF output, without being able to link the data to any individual issuance phase.

Throughout this document, we adhere to the recommendations laid out in [OPRF] in integrating the VOPRF protocol into our wider workflow. Where necessary, we lay out exactly which part of the VOPRF API we use. We stress that the generalized protocol only includes steps and messages that contain cryptographic data.

We decide against defining abstract interfaces for enclosing Privacy Pass data and functionality. Instead, we describe the Privacy Pass protocol in the same group setting that is used in [OPRF].

### 2.1. Key initialisation phase

In the initialisation phase, essentially we run the VOPRF setup phase in that the server runs `VOPRF_Setup(l)` where `l` is the required bit-length of the prime used in establishing the order of the group `GG`. This outputs the tuple  $(k, Y, p)$  where:  $p = p(l)$  is the prime order of  $GG = GG(l)$ ;  $k$  is a uniformly sampled element from  $GF(p)$ ; and  $Y = kG$  for some fixed generator of  $GG$ .

However, the server must first come to an agreement on what group instantiation to support. This involves choosing an instantiation with the required security level implied by the choice of `l`. The server has a list of supported group params (`GROUP_PARAMS`) and chooses an identifier, `id`, associated with the preferred group configuration, and also outputs the implied length of `l`. It creates a Privacy Pass key object denoted by `ppKey` that has fields "private", "public" and "group". It sets `ppKey.private = bytes(k)`, `ppKey.public = bytes(Y)` and `ppKey.group = id`.

The server creates a JSON object of the form below.

```
{
  "Y": pp_key.public,
  "expiry": <expiry_date>,
  "sig": <signature>
}
```

The field "expiry" corresponds to an expiry date for the newly sampled key. We recommend that each key has a lifetime of between 1 month and 6 months. The field "sig" holds an ASN1-encoded ECDSA signature evaluated over the contents of "Y" and "expiry". The ECDSA parameters should be equivalent to the group instantiation used for the OPRF, and the signing key (`ecdsaSK`) should be long-term with a corresponding publicly available verification key (`ecdsaVK`). We summarize the creation of this object using the algorithm `PP_key_init()`, which we define below.

```

function PP_key_init(k, Y, id) {
  var ppKey = {}
  ppKey.private = k
  ppKey.public = Y
  ppKey.group = id
  var today = new Date()
  var expiry = today.setMonth(today.getMonth() + n);
  var obj = {
    Y: ppKey.public,
    expiry: expiry,
    sig: ECDSA.sign(ecdsaSK, ppKey.public .. bytes(expiry)),
  }
  return [ppKey, obj]
}

```

Note that the variable *n* above should correspond to the number of months ahead that the expiry date should correspond to.

We give a diagrammatic representation of the initialisation phase below.

|   |  |
|---|--|
| <p>C(ecdsaVK)</p> <p>-----</p>  | <p>S(ecdsaSK)</p> <p>-----</p>   |
|   | <pre> l = GROUP_PARAMS[id] (k,Y,p) = VOPRF_Setup(l) [ppKey,obj] = PP_key_init(k,Y,id) </pre> |
|   | <p>obj</p> <p>&lt;-----</p>  |
| <pre> public = key.Y if (!ECDSA.verify(ecdsaVK, public .. bytes(obj.expiry)) {   panic(KEY_VERIFICATION_ERROR) } else if (!(new Date() &gt; obj.expiry)) {   panic(KEY_VERIFICATION_ERROR) } store(obj.id, obj.public) </pre> |  |
|   | <p>push(key)</p>   |

The variable *obj* essentially corresponds to a cryptographic commitment to the server's VOPRF key. We abstract all signing and verification of ECDSA signatures into the ECDSA.sign and ECDSA.verify functionality [DSS].

In the initialisation phase above, we require that the server contacts each viable client. In Section 3 we discuss the possibility of uploading public key material to a trusted registry that client's access when communicating with the server.

## 2.2. Issuance phase

The issuance phase allows the client to receive VOPRF evaluations from the server. The issuance phase essentially corresponds to a VOPRF evaluation phase [OPRF]. In essence, the client generates a valid VOPRF input  $x$  (a sequence of bytes from some unpredictable distribution), and runs the VOPRF evaluation phase with the server. The client receives an output  $y$  of the form:

$$y = \text{VOPRF\_Finalize}(x, N, \text{aux})$$

where  $N$  is a group element, and  $\text{aux}$  is auxiliary data that is generated by the client. More specifically,  $N$  is an unblinded group element equal to  $k \cdot H_1(x)$  where  $H_1$  is a random oracle that outputs elements in  $GG$ . The client stores  $(x, y)$  as recommended in [OPRF]. We give a diagrammatic overview of the protocol below.



```

C(x, aux)                                     S(ppKey)
-----
var ciph = retrieve(S.id)
var (r,M) = VOPRF_Blind(x)

                                M
                                ----->

                                (Z,D) = VOPRF_Eval(ppKey.private,
                                ciph.G,Y,M)
                                var resp = {
                                element: Z,
                                proof: D,
                                version: "key_version",
                                }

                                resp
                                <-----

var elt = resp.element
var proof = resp.proof
var version = resp.version
var obj = retrieve(S.id, version)
if obj == "error" {
    panic(KEY_VERIFICATION_ERROR)
}
var N = VOPRF_Unblind(ciph.G,obj.Y,M,elt,proof)
var y = VOPRF_Finalize(x,N,aux)
if (y == "error") {
    panic(CLIENT_VERIFICATION_ERROR)
}

push((ciph,x,y,aux))

```

In the diagram above, the client knows the VOPRF ciphersuite supported by the server when it retrieves in the first step. It uses this information to correctly perform group operations before sending the first message.

### 2.3. Redemption phase

The redemption phase allows the client to reauthenticate to the server, using data that it has received from a previous issuance phase. By the security of the VOPRF, even revealing the original input  $x$  that is used in the issuance phase does not affect the privacy of the client.

```

C()                                     S(ppKey)
-----
ciph1 = retrieve(S.id, "ciphersuite")
a = pop()
while (a != undefined) {
  (ciph2,x,y,aux) = a
  if (ciph1 != ciph2) {
    // ciphersuites do not match
    a = pop()
    continue
  }
}
if (a == undefined) {
  // no valid data to redeem
  return
}

                                (x,y,aux)
                                ----->

                                if (store.includes(x)) {
                                  panic(DOUBLE_SPEND_ERROR)
                                }
                                T = H1(x)
                                N' = OPRF_Eval(ppKey.private, T)
                                y' = OPRF_Finalize(x,N',aux)
                                resp = (y' == y)
                                      ? "success"
                                      : "failure"
                                store.push(x)

                                resp
                                <-----

output resp

```

Note that the server uses the API provided by `OPRF_Eval` and `OPRF_Finalize`, rather than the corresponding `VOPRF` functions. This is because the `VOPRF` functions also compute zero-knowledge proof data that we do not require at this stage of the protocol.

### 2.3.1. Double-spend protection

To protect against clients that attempt to spend a value  $x$  more than once, the server uses an index, `store`, to collect valid inputs and then check against in future protocols. Since this store needs to only be optimized for storage and querying, a structure such as a Bloom filter suffices. Importantly, the server must only eject this

storage after a key rotation occurs since all previous client data will be rendered obsolete after such an event.

#### 2.3.2. Finalization during redemption

The last step if the issuance phase for the client is to run `VOPRF_Finalize` and store the output. In some applications, it may be necessary to link the output of `VOPRF_Finalize` to the actual redemption that is occurring. This can be done by tailoring the auxiliary data "aux" to something specific.

In order to do this, it is necessary to store only (ciph, x, N) in the issuance phase of the protocol. Then during the redemption phase, generate auxiliary data "aux" and compute `VOPRF_Finalize(x,N,aux)` after retrieving the triplet above.

#### 2.4. Error types

- o `KEY_VERIFICATION_ERROR`: Error occurred when verifying signature and expiry date for a server public key
- o `CLIENT_VERIFICATION_ERROR`: Error verifying issuance response from server.
- o `DOUBLE_SPEND_ERROR`: Indicates that a client has attempted to redeem a token that has already been used for authorization

### 3. Key registration

Rather than sending the result of the key initialisation procedure directly to each client, it is preferable to upload the object obj to a trusted, tamper-proof, history-preserving registry. By trusted, we mean from the perspective of clients that use the Privacy Pass protocol. Any new keys uploaded to the registry should be appended to the list. Any keys that have expired can optionally be labelled as so, but should never be removed. A trusted registry may hold key commitments for multiple Privacy Pass service providers (servers).

Clients can either choose to:

- o poll the trusted registry and import new keys, rejecting any that throw errors;
- o retrieve the commitments for the server at the time at which they are used, throwing errors if no valid commitment is available.

To prevent unauthorized modification of the trusted registry, server's should be required to identify and authenticate themselves

before they can append data to their configuration. Moreover, only parts of the registry that correspond to the servers configuration can be modifiable.

The registry that we describe could be fulfilled by Key Transparency [keytrans] or other similar architectures.

### 3.1. Key rotation

Whenever a server seeks to rotate their key, they must append their key to the trusted registry. We recommend that the trusted registry is arranged as a JSON blob with a member for each JSON provider. Each provider appends new keys by creating a new sub-member corresponding to an incremented version label along with their new commitment object.

Concretely, we recommend that the trusted registry is a JSON file of the form below.

```
{
  "server_1": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "1.1": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
  },
  "server_2": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
  },
  ...
}
```

In this structure, "server\_1" and "server\_2" are separate service providers. The sub-member "ciphersuite" corresponds to the choice of VOPRF ciphersuite made by the server. The sub-members "1.0", "1.1" of "server\_1" correspond to the versions of commitments available to

the client. Increasing version numbers should correspond to newer keys. Each commitment should be a valid encoding of a point corresponding to the group in the VOPRF ciphersuite specified in "ciphersuite".

If "server\_2" wants to upload a new commitment with version tag "1.1", it runs the key initialisation procedure from above and adds a new sub-member "1.1" with the value set to the value of the output obj. The "server\_2" member should now take the form below.

```
{
  ...
  "server_2": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "1.1": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
  },
  ...
}
```

### 3.2. Client retrieval

We define a function "retrieve(server\_id, version\_id)" which retrieves the commitment with version label equal to version\_id, for the provider denoted by the string server\_id. For example, retrieve("server\_1", "1.1") will retrieve the member labelled with "1.1" above.

We implicitly assume that this function performs the following verification checks:

```
if (!ECDSA.verify(ecdsaVK, obj.Y .. bytes(obj.expiry)) {
  return "error"
} else if (!(new Date() < obj.expiry)) {
  return "error"
}
```

If "error" is not returned, then it instead returns the entire object. We also abuse notation and also use "ciph =

retrieve(server\_id, "ciphersuite")" to refer to retrieving the ciphersuite for the server configuration.

### 3.3. Key revocation

If a server must revoke a key, then it uses a separate member with label "revoke" corresponding to an array of revoke versions associated with key commitments. In the above example, if "server\_2" needs to revoke the key with version "1.0", then it appends a new "revoke" member with the array "[ "1.0" ]". Any future revocations can simply be appended to this array. For an example, see below.

```
{
  ...
  "server_2": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "1.1": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "revoked": [ "1.0" ],
  },
  ...
}
```

Client's are required to check the "revoked" member for new additions when they poll the trusted registry for new key data.

### 3.4. VOPRF ciphersuites

We strongly RECOMMEND that a server uses only one VOPRF ciphersuite at any one time. Should a server choose to change some aspect of the ciphersuite (e.g., the group instantiation or other cryptographic functionality) we further RECOMMEND that the server create a new identifying label (e.g. "server\_1\_\${ciphersuite\_id}") where ciphersuite\_id corresponds to the identifier of the VOPRF ciphersuite. Then "server\_1" revokes all keys for the previous ciphersuite and then only offers commitments for the current label.

An alternative arrangement would be to add a new layer of members between server identifiers and key versions in the JSON struct,

corresponding to "ciphersuite\_id". Then the client may choose commitments from the appropriate group identifying member.

We strongly recommend that service providers only operate with one group instantiation at any one time. If a server uses two VOPRF ciphersuites at any one time then this may become an avenue for segregating the user-base. User segregation can lead to privacy concerns relating to the utility of the obliviousness of the VOPRF protocol (as raised in [OPRF]). We discuss this more in Section 5.

### 3.5. ECDSA key material

For clients must also know the verification (ecdsaVK) for each service provider that they support. This enables the client to verify that the commitment is properly formed before it uses it. We do not provide any specific recommendations on how the client has access to this key, beyond that the verification key should be accessible separately from the trusted registry.

While the number of service providers associated with Privacy Pass is low, the client can simply hardcode the verification keys directly for each provider that they support. This may be cumbersome if a provider wants to rotate their signing key, but since these keys should be comparatively long-term (relative to the VOPRF key schedule), then this should not be too much of an issue.

## 4. Protocol configurations

We provide an overview of some of the possible ways of configuring the Privacy Pass protocol situation, such that it can be used as a lightweight trust attestation mechanism for clients.

### 4.1. Single-Issuer Single-Verifier

The simplest way of considering the Privacy Pass protocol is in a setting where the same server plays the role of issuer and verifier, we call this "Single-Issuer Single-Verifier" (SISV). In SISV, we consider a server S that publishes commitments for their secret key k, that a client C has access to.

When S wants to issue tokens to C, they invoke the issuance protocol where C generates their own inputs and S uses their secret key k. In this setting, C can only perform token redemption with S. When a token redemption is required, C and S invoke the redemption phase of the protocol, where C uses an issued token from a previous exchange, and S uses k as their input again.

In SISV, C proves that S has attested to the honesty of C at some point in the past (without revealing exactly when). S can use this information to inform its own decision-making about C without having to recompute the trust attestation task again.

#### 4.2. Single-Issuer Forwarding-Verifier

In this setting, each client C obtains issued tokens from a server S via the issuance phase of the protocol. The difference is that clients can prove that S has attested to their honesty in the past with any verifier V. We still only consider S to hold their own secret key.

When C interacts with V, V can ask C to provide proof that the separate issuer S has attested to their trust. The first stage of the redemption phase of the protocol is invoked between C and V, which sees C send the unused token  $(x, y, aux)$  to V. This message is then used in a redemption exchange between V and S, where V plays the role of the client. Then S sends the result of the redemption exchange to V, and V uses this result to determine whether C has the correct trust attestation.

This configuration is known as "Single-Issuer Forwarding-Verifier" or SIFV to refer to the verifier V who uses the output of the redemption phase for their own decision-making.

#### 4.3. Single-Issue Asynchronous-Verifier

This setting is inspired by recently proposed APIs such as [TRUST]. It is similar to the SIFV configuration, except that the verifiers V no longer interact with the issuer S. Only C interacts with S, and this is done asynchronously to the trust attestation request from V. Hence "Asynchronous-Verifier" (SIIV).

When V invokes a redemption for C, C then invokes a redemption exchange with S in a separate session. If verification is carried out successfully by S, S instead returns a Signed Redemption Record (SRR) that contains the following information:

```
"result": {
  "timestamp": "2019-10-09-11:06:11",
  "verifier": "V",
},
"signature": sig,
```

The "signature" field carries a signature evaluated over the contents of "result" using a long-term signing key for the issuer S, of which the corresponding public key is well-known to C and V. Then C can



prove that their trust attestation from S to V by sending the SRR to V. The SRR can be verified by V by verifying the signature using the well-known public key for S.

Such records can be cached to display again in the future. The issuer can also add an expiry date to the record to determine when the client must refresh the record.

#### 4.4. Bounded-Issuers

Each of the configurations above can be generalized to settings where a bounded number of issuers are allowed, and verifiers can invoke trust attestations for any of the available issuers. Subsequently, this leads to three new configurations known as BISV, BIFV, BIAV.

As we will discuss later in Section 5.1.2, configuring a large number of issuers can lead to privacy concerns for the clients in the ecosystem. Therefore, we are careful to ensure that the number of issuers is kept strictly bounded by a fixed small number  $M$ . The actual issuers can be replaced with different issuers as long as the total never exceeds  $M$ . Moreover, issuer replacements also have an effect on client privacy that is similar to when a key rotation occurs, so replacement should only be permitted at similar intervals.

See Section 5.1.2 for more details about safe choices of  $M$ .

##### 4.4.1. Fixing the bound

Configuring any number of issuers greater than 1 effectively reduces privacy by an extra bit. As a result, we see an exponential decrease in privacy in the number of issuers that are currently active. Therefore the value of  $M$  should be kept very low (we recommend no higher than 4).

#### 5. Privacy considerations

We intentionally encode no special information into redemption tokens to prevent a vendor from learning anything about the client. We also have cryptographic guarantees via the VOPRF construction that a vendor can learn nothing about a client beyond which issuers trust it. Still there are ways that malicious servers can try and learn identifying information about clients that it interacts with.

We discuss a number of privacy considerations made in [OPRF] that are relevant to the Privacy Pass protocol use-case, along with additional considerations arising from the specific ways of using the Privacy Pass protocol in Section 4.

### 5.1. User segregation

An inherent features of using cryptographic primitives like VOPRFs is that any client can only remain private relative to the entire space of users using the protocol. In principle, we would hope that the server can link any client redemption to any specific issuance invocation with a probability that is equivalent to guessing. However, in practice, the server can increase this probability using a number of techniques that can segregate the user space into smaller sets.

#### 5.1.1. Key rotation

As introduced in [OPRF], such techniques to introduce segregation are closely linked to the type of key schedule used by the server. When a server rotates their key, any client that invokes the issuance protocol shortly afterwards will be part of a small number of possible clients that can redeem. To mechanize this attack strategy, a server could introduce a fast key rotation policy which would force clients into small key windows. This would mean that client privacy would only have utility with respect to the smaller group of users that have Trust Tokens for a particular key window.

In the [OPRF] draft it is recommended that great care is taken over key rotations, in particular server's should only invoke key rotation for fairly large periods of time such as between 1 and 12 months. Key rotations represent a trade-off between client privacy and continued server security. Therefore, it is still important that key rotations occur on a fairly regular cycle to reduce the harmfulness of a server key compromise.

Trusted registries for holding Privacy Pass key commitments can be useful in policing the key schedule that a server uses. Each key must have a corresponding commitment in this registry so that clients can verify issuance responses from servers. Clients may choose to inspect the history of the registry before first accepting redemption tokens from the server. If a server has updated the registry with many unexpired keys, or in very quick intervals a client can choose to reject the tokens.

TODO: Can client's flag bad server practices?

#### 5.1.2. Large numbers of issuers

Similarly to the key rotation issue raised above, if there are a large number of issuers, similar user segregation can occur. In the BISV, BIFV, BIAV configurations of using the Privacy Pass protocol (Section 4), a verifier OV can trigger redemptions for any of the

available issuers. Each redemption token that a client holds essentially corresponds to a bit of information about the client that OV can learn. Therefore, there is an exponential loss in privacy relative to the number of issuers that there are.

For example, if there are 32 issuers, then OV learns 32 bits of information about the client. If the distribution of issuer trust is anything close to a uniform distribution, then this is likely to uniquely identify any client amongst all other Internet users. Assuming a uniform distribution is clearly the worst-case scenario, and unlikely to be accurate, but it provides a stark warning against allowing too many issuers at any one time.

As we noted in Section 4.4, a strict bound should be applied to the active number of issuers that are allowed at one time. We propose that allowing no more than 6 issuers at any one time is highly preferable (leading to a maximum of 64 possible user segregations). Issuer replacements should only occur with the same frequency as key rotations as they can lead to similar losses in privacy if users still hold redemption tokens for previously active issuers.

In addition, we recommend that trusted registries indicate at all times which issuers are deemed to be active. If a client is asked to invoke any Privacy Pass exchange for an issuer that is not declared active, then the client should refuse to participate in the protocol.

#### 5.1.2.1. Selected trusted registries

One recommendation is that only a fixed number (TODO: how many?) of issuers are sanctioned to provide redemption tokens at any one time. This could be enforced by the trusted registry that is being used. Client's can then choose which registries to trust and only accept redemption tokens from issuers accepted into those registries.

#### 5.1.2.2. Maximum number of issuers inferred by client

A second recommendation is that clients only store redemption tokens for a fixed number of issuers at any one time. This would prevent a malicious verifier from being able to invoke redemptions for many issuers since the client would only be holding redemption tokens for a small set of issuers. When a client is issued tokens from a new issuer and already has tokens from the maximum number of issuers, it simply deletes the oldest set of redemption tokens in storage and then stores the newly acquired tokens.

## 5.2. Tracking and identity leakage

While redemption tokens themselves encode no information about the client redeeming them, there may be problems if we allow too many redemptions on a single page. For instance, the first-party cookie for user U on domain A can be encoded in the trust token information channel and decoded on domain B, allowing domain B to learn the user's domain A cookie until either first-party cookie is cleared. Mitigations for this issue are similar to those proposed in Section 5.1.2 for tackling the problem of having large number of issuers.

In SIAV, cached SRRs and their associated issuer public keys have a similar tracking potential to first party cookies in the browser setting. Therefore these should be clearable by the client using standard deletion methods.

## 6. Security considerations

We present a number of security considerations that prevent a malicious actors from abusing the protocol.

### 6.1. Double-spend protection

All issuing server should implement a robust storage-query mechanism for checking that tokens sent by clients have not been spent before. Such tokens only need to be checked for each issuer individually. But all issuers must perform global double-spend checks to avoid clients from exploiting the possibility of spending tokens more than once against distributed token checking systems. For the same reason, the global data storage must have quick update times. While an update is occurring it may be possible for a malicious client to spend a token more than once.

### 6.2. Key rotation

We highlighted previously that short key-cycles can be used to reduce client privacy. However, regular key rotations are still recommended to maintain good server key hygiene. The key material that we consider to be important are:

- o the VOPRF key;
- o the signing key used to sign commitment information;
- o the signing key used to sign SRRs in the SIAV configuration.

In summary, our recommendations are that VOPRF keys are rotated from anywhere between a month and a single year. With an active user-base, a month gives a fairly large window for clients to participate in the Privacy Pass protocol and thus enjoy the privacy guarantees of being part of a larger group. The low ceiling of a year prevents a key compromise from being too destructive. If a server realizes that a key compromise has occurred then the server should revoke the previous key in the trusted registry and specify a new key to be used.

For the two signing keys, these should both be well-known keys associated with the issuer (TODO: where should they be stored?). Issuers may choose to use the same key for both signing purposes. The rotation schedules for these keys can be much longer, if necessary.

### 6.3. Token exhaustion

When a client holds tokens for an issuer, it is possible for any verifier to invoke that client to redeem tokens for that issuer. This can lead to an attack where a malicious verifier can force a client to spend all of their tokens for a given issuer. To prevent this from happening, methods should be put into place to prevent many tokens from being redeemed at once.

For example, it may be possible to cache a redemption for the entity that is invoking a token redemption. In SISV/SIFV, if the verifier requests more tokens than the client simply returns the cached token that it returned previously. This could also be handled by simply not redeeming any tokens for the entity if a redemption had already occurred in a given time window.

In SIAV, the client instead caches the SRR that it received in the asynchronous redemption exchange with the issuer. If the same verifier attempts another trust attestation request, then the client simply returns the cached SRR. The SRRs can be revoked by the issuer, if need be, by providing an expiry date or by signaling that records from a particular window need to be refreshed.

## 7. Summary of privacy and security parameters

We provide a summary of the parameters that we use in the Privacy Pass protocol. These parameters are informed by both privacy and security considerations that are highlighted in Section 5 and Section 6, respectively. These parameters are intended as a single reference point for implementers when implementing the protocol.

Firstly, let  $U$  be the total number of users,  $I$  be the total number of issuers. Assuming that each user accept tokens from a uniform sampling of all the possible issuers, as a worst-case analysis, this segregates users into a total of  $2^I$  buckets. As such, we see an exponential reduction in the size of the anonymity set for any given user. This allows us to specify the privacy constraints of the protocol below, relative to the setting of  $A$ .

| parameter                                  | value             |
|--|-------------------|
| Minimum anonymity set size ( $A$ )         | 5000              |
| Recommended key lifetime ( $L$ )           | 1 - 6 months      |
| Recommended key rotation frequency ( $F$ ) | $L/2$             |
| Maximum allowed issuers ( $I$ )            | $\log_2(U/A) - 1$ |
| Maximum active issuance keys               | 1                 |
| Maximum active redemption keys             | 2                 |
| Minimum security parameter                 | 196 bits          |

#### 7.1. Justification

We make the following assumptions in these parameter choices.

- o Inferring the identity of a user in a 5000-strong anonymity set is difficult
- o After 2 weeks, all clients in a system will have rotated to the new key

The maximum choice of  $I$  is based on the equation  $1/2 * U/2^I = A$ . This is because  $I$  issuers lead to  $2^I$  segregations of the total user-base  $U$ . By reducing  $I$  we limit the possibility of performing the attacks mentioned in Section 5.1.

We must also account for each user holding issued data for more then one possible active keys. While this may also be a vector for monitoring the access patterns of clients, it is likely to unavoidable that clients hold valid issuance data for the previous key epoch. This also means that the server can continue to verify redemption data for a previously used key. This makes the rotation period much smoother for clients.

For privacy reasons, it is recommended that key epochs are chosen that limit clients to holding issuance data for a maximum of two keys. By choosing  $F = L/2$  then the minimum value of  $F$  is 1/2 a month, since the minimum recommended value of  $L$  is 1 month. Therefore, by the initial assumption, then all users should only have access to only two keys at any given time. This reduces the anonymity set by another half at most.

Finally, the minimum security parameter size is related to the cryptographic security offered by the group instantiation that is chosen. For example, if we use an elliptic curve over a 256-bit prime field, then the actual group instantiation offers 128 bits of security (or a security parameter of size 128 bits). However, as noted in [OPRF], OPRF protocols reduce the effective security of the group by  $\log_2(M)$  where  $M$  is the number of queries. As such, we choose the minimum size of the security parameter to be 196 bits, so that it is difficult for a malicious client to exploit this.

## 7.2. Example parameterization

Using the specification above, we can give some example parameterizations. For example, the current Privacy Pass browser extension [PPEXT] has over 150,000 active users (from Chrome and Firefox). Then  $\log_2(U/A)$  is approximately 5 and so the maximum value of  $I$  should be 4.

If the value of  $U$  is much bigger (e.g. 5 million) then this would permit  $I = \log_2(5000000/5000) - 1 = 8$  issuers.

## 8. References

### 8.1. Normative References

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", n.d., <<https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>>.

[DSS] Federal Information Processing Standards Publication, ., "FIPS PUB 186-4: Digital Signature Standard (DSS)", n.d., <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.

[keytrans] "Security Through Transparency", n.d., <<https://security.googleblog.com/2017/01/security-through-transparency.html>>.

- [OPRF] "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", n.d.,  
<<https://tools.ietf.org/html/draft-irrf-cfrg-voprf-01>>.
- [PPEXT] "Privacy Pass Browser Extension", n.d.,  
<<https://github.com/privacypass/challenge-bypass-extension>>.
- [PPSRV] "Cloudflare Supports Privacy Pass", n.d.,  
<<https://blog.cloudflare.com/cloudflare-supports-privacy-pass/>>.
- [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>>.
- [TRUST] "Trust Token API", n.d., <<https://github.com/WICG/trust-token-api#security-considerations>>.

## 8.2. URIs

- [1] <https://github.com/grittygrease/draft-privacy-pass>

### Authors' Addresses

Alex Davidson  
Cloudflare Portugal  
Largo Rafael Bordalo Pinheiro 29  
Lisbon  
Portugal

Email: [alex.davidson92@gmail.com](mailto:alex.davidson92@gmail.com)

Nick Sullivan  
Cloudflare  
101 Townsend Street  
San Francisco  
United States of America

Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)