HTTP Message Signatures

Justin Richer

IETF HTTP WG Interim

June 17, 2021

How HTTP Message Signing works

1. Choose covered portions and crypto parameters
2. Normalize the HTTP message components
3. Generate a signature input string
4. Sign the string creating a signature output
5. Add the signature output and parameters as structured HTTP headers

Example HTTP Message

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18

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

Sign These Parts

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18

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

Signature Base

```
"@request-target": post /foo?param=value&pet=dog
"host": example.com
"date": Tue, 20 Apr 2021 02:07:55 GMT
"content-type": application/json
"@signature-params": ("@request-target" "host" "date"
  "content-type");created=1618884475;keyid="test-key-rsa-pss"
```

Signature Bytes

Lu2cC2Ifw3hkpXt8iC9g78qppHzEUo7hPyeFmDNqkMe4AvPzhz8cRhI1+eI
BisvM7ceDh4Om0RmKjA5CUL5TFs9NuUHC0xuZZeiy5u7THftAZZU6LgwRyn
MuOZgJAYXYDsGBKfxRkoGKVVEX1lSGi7RVhYl/EgWCJzuIbJ9mLeRxzaXRr
3pZXz5xRaXcsXItpsK3AnWYHoc6YAT9hP5M3oJPeb3KRHoLAn4nheC0kFoy
LzRAf6/BNb4I7JhwqVZMZBlndnI/KTBXoTK7rzYFdpX/Cbtwv+XHgli9QtH
ktw9hXC4Kv4lp2fCGSPJPHKeyrZ0rhCcfe++eJe0Ykm3FIw==

Signed Request

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18
Signature-Input: sig1=("@request-target" "host" "date" "content-
type");created=1618884475;keyid="test-key-rsa-pss"
Signature:
sig1=:Lu2cC2Ifw3hkpXt8iC9g78qppHzEUo7hPyeFmDNqkMe4AvPzhz8cRhI1+eIBisvM7ceDh4Om0
RmKjA5CUL5TFs9NuUHC0xuZZeiy5u7THftAZZU6LgwRynMuOZgJAYXYDsGBKfxRkoGKVVEX1lSGi7RV
hYl/EgWCJzuIbJ9mLeRxzaXRr3pZXz5xRaXcsXItpsK3AnWYHoc6YAT9hP5M3oJPeb3KRHoLAn4nheC
0kFoyLzRAf6/BNb4I7JhwqVZMZBlndnI/KTBXoTK7rzYFdpX/Cbtwv+XHgli9QtHktw9hXC4Kv4lp2f
CGSPJPHKeyrZ0rhCcfe++eJe0Ykm3FIw==:

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

How HTTP Message Verification works

1. Read the Signature-Input and Signature header values
2. Validate covered portions and crypto parameters
3. Normalize the HTTP message components
4. Re-generate the signature input string
5. Verify the signature against the signature input string

Some important aspects

- Detached signature, not encapsulation
- Uses HTTP Structured Fields
- Allows multiple signatures on a message
- Can sign most HTTP parts
- Works for requests and responses
- Relatively robust against common changes

Since Last We Met

- Structured field values everywhere!
- Signature parameters are now signed
- Introduced “specialty identifiers” construct
- Removed “message signature” artifact structure
- New process for selecting keys and algorithms
- Strongly defined algorithm parameters
- Removed list-prefix processing
- Guidance for applications and profiles
- Completely reworked/regenerated examples

Current Status

- Core signature process is stable
- Implementations in several languages
- Starting to see feedback from implementors of older specs (Cavage, OAuth PoP)
- Proposed as basis for new OAuth PoP spec
  - Not written/submitted yet

Algorithm Definitions

- Could use any crypto process that can sign the string and spit out a stack of bytes
  - Draft defines input and output to sign and verify functions
  - If your application's got a signature method you can just use it within your sphere
  - Ability to use JOSE Web Algorithms without copying the registry itself
- Registry of interoperable algorithms and identifiers
  - Thanks to Kathleen Moriarty and CFRG for feedback on text

Example algorithm definition

To sign using this algorithm, the signer applies the **RSASSA-PSS-SIGN (K, M)** function [RFC8017] with the signer's private signing key (K) and the signature input string (M) (Section 2.4). The mask generation function is MGF1 as specified in [RFC8017] with a hash function of SHA-512 [RFC6234]. The salt length (sLen) is 64 bytes. The hash function (Hash) SHA-512 [RFC6234] is applied to the signature input string to create the digest content to which the digital signature is applied. The resulting signed content byte array (S) is the HTTP message signature output used in Section 3.1.

Selecting an Algorithm and Key

- External configuration or higher level protocol
  - E.g, GNAP ties the key to the client
- Figuring out the “alg” from the “key”
  - JWKs have their own “alg” field
  - Behavior of old “hs2019” pseudo-algorithm
- (Optional) Explicit “alg” and “keyid” fields
  - When you need to be able to switch at runtime

Time for Bikeshedding!

Algorithm Identifiers

- Defined strings with strict interpretations:
  - rsa-pss-sha512
  - rsa-v1_5-sha256
  - hmac-sha256
  - ecdsa-p256-sha256
- No parsing, no taking parameters from the name, no “bonus” definitions by swapping out

Proposed alternatives

- Date-based (from Manu)
  - rsa-2003
  - rsa-2005
  - ecdsa-2013
  - hmac-2006
  - Aliases: recommended-signature-2015
- JWA
  - RS256, PS512, …

Other named parts

- Signature parameters
  - alg, keyid, created, expires, nonce
- Specialty content identifiers
  - @request-target
  - @signature-params

Next Steps

- Align with HTTP Semantics terminology
  - “Covered Content” -> ??
  - “Headers” -> Fields
- Split “@request-target” into new specialty tag(s)?
- More stuff with responses (@status-code?)
- EdDSA Signing?
- Special cases: Via headers, empty headers, others?
- More examples! More code!

More Next Steps

- Branding and framing
  - Normalization is a bigger part than signing
  - It's also about verifying signatures
- Guidance to developers on choosing security parameters for their applications
- Security Considerations
- Privacy Considerations
- IANA registry guidelines