HTTP Message Signatures and OAuth Proof of Possession (PoP)

Justin Richer

IETF OAuth WG Interim

October 6, 2021

A brief history

- OAuth 1.0
- OAuth 2.0 “MAC Token”
- OAuth 2.0 “Proof of Possession Architecture”
- OAuth 2.0 “Signed HTTP Requests”
  - draft-ietf-oauth-signed-http-request

How the old draft worked

1. Mash together covered headers, hash it
2. Mash together covered query, hash it
3. Hash the body
4. Add a few extra variables (tokens, timestamp, method, URI, …)
5. Wrap everything up in a JWS and send it

draft-ietf-httpbis-message-signatures

- Official document of HTTP WG
  - Starting to approach final stages
- Builds on several community drafts, including the old OAuth draft
- HTTP-native, covers more of the message
- Built to be profiled and flexible

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

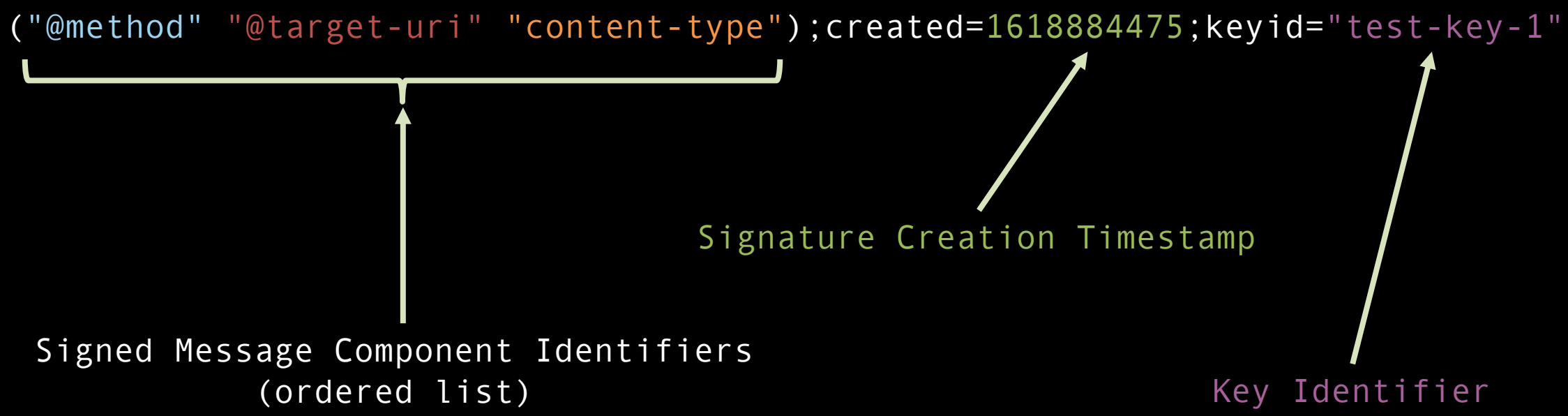
```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18

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

Sign These Components

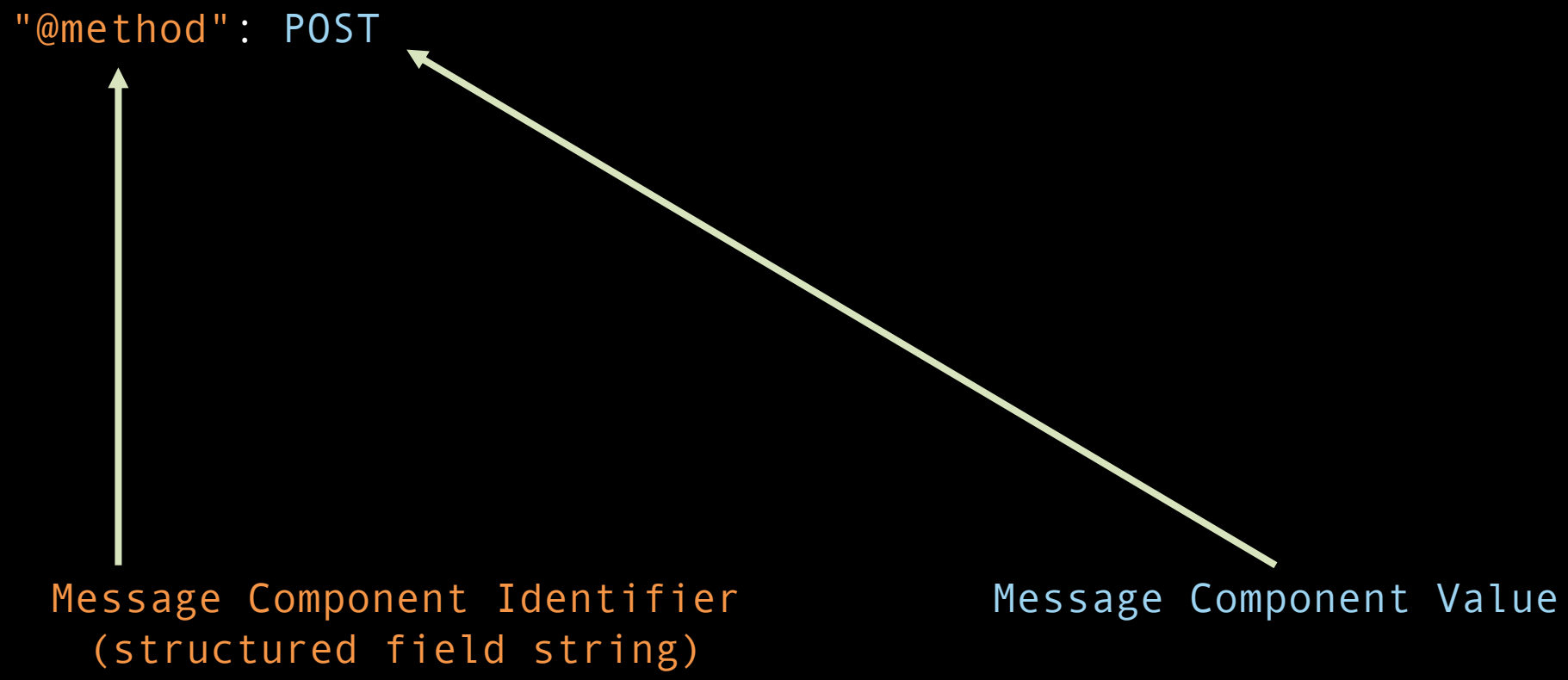
```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18

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

Message Component Identifiers

Signature Input Parameters



Signature Input String



Signature Input String

```
"@method": POST
"@target-uri": https://example.com/foo?param=value&pet=dog
"content-type": application/json
"@signature-params": ("@method" "@target-uri"
  "content-type");created=1618884475;keyid="test-key-1"
```

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=("@method" "@target-uri"
  "content-type");created=1618884475;keyid="test-key-1"
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 from the message
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

How to apply it to OAuth 2.0

- New token type: HTTPSig
- Minimum request coverage requirements
- Requirement to present Signature, Signature-Input, and Authorization headers together to the RS
- Key and algorithm determined by client context
  - Pre-registered, generated, negotiated, other?
  - Communicated in JWT or introspection to RS

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
Authorization: HTTPSig 3ZM-B0XGPQTR31UOH6XKG.WEM1N3G98L
Signature-Input: sig1=("@method" "@target-uri” "content-type"
    "authorization");created=1618884475;keyid="test-key-rsa-pss"
Signature:
sig1=:NtIKWuXjr4SBEXj97gbick4O95ff378I0CZOa2VnIeEXZ1itzAdqTpSvG91XYrq5CfxCmk8zz
1Zg7ZGYD+ngJyVn805r73rh2eFCPO+ZXDs45Is/Ex8srzGC9sfVZfqeEfApRFFe5yXDmANVUwzFWCEn
GM6+SJVmWl1/jyEn45qA6Hw+ZDHbrbp6qvD4N0S92jlPyVVEh/SmCwnkeNiBgnbt+E0K5wCFNHPbo4X
1Tj406W+bTtnKzaoKxBWKW8aIQ7rg92zqE1oqBRjqtRi5/Q6P5ZYYGGINKzNyV3UjZtxeZNnNJ+MAnW
S0mofFqcZHVgSU/1wUzP7MhzOKLca1Yg==:

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

Introspection Response


```
{
        "active": true,
        "token_type": "HTTPSig",
        "cnf": {
                "jkt": "0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
        }
}
```

What about DPoP?

- Let them co-exist!
- Two different flavors
  - DPoP: SPA, minimalism
  - DPoP: dynamic asymmetric keys made by client
  - PoP: all clients, flexibility, extensibility
  - PoP: various forms of key distribution and types

Why not just use JOSE?

- Facilitate interaction with non-JOSE systems
- JOSE excels at self-contained crypto
  - HTTP messages aren't easily containable
  - Need to either duplicate components or wrap whole message inside JOSE object
  - Result is fragile against known and expected HTTP transformations
- HTTP Message Signatures supports JWA for algorithm resolution
- HTTP Message Signatures supports advanced use cases like multiple chained signatures

Draft Status

- draft-richer-oauth-httpsig-00
- Pretty short
- Doesn’t answer how to get keys in place
  - Should it? Old OAuth draft didn’t…

Next Steps

- Adopt draft-richer-oauth-httpsig to replace draft-ietf-oauth-signed-http-request
- Open questions
  - Registration of keys (static, dynamic, transactional)
  - Key distribution (per-token, AS-generated)
  - Algorithm/feature discovery
  - Client authentication with message signatures