Cachable OSCORE

Work in progress towards

draft-amsuess-core-cachable-oscore-01

Christian Amsüss
Marco Tiloca, RISE

CoRE Interim Meeting, January 21st, 2021

Recap

Enable proxies to cache OSCORE responses

- Message protection with Group OSCORE
- Clients and server have to already be group members
- The proxy is untrusted and not a group member

- Clients need a same Group OSCORE “consensus request”
  - This will hit a cache entry, for the proxy to return a cached response

- Done a major revision
  - Focus on deterministic requests
  - Use dedicated option for Request-Hash
  - Build requests on pairwise mode
  - Following the discussion at the CoRE interim in November 2020
  - Aligned with the upcoming version -11 of Group OSCORE

Key concepts

- Deterministic client – C^*
  - Fictitious group member, setup by the Group Manager
  - No Sequence Number, no Recipient Context, no public/private key

- Clients obtain information about C^*
  - Sender ID (→ Sender Key), Hash algorithm $h(\cdot)$
  - From the Group Manager, e.g. when joining the group

- Each client can act as if it was C^*
  - Given a same plain CoAP request M …
  - each client computes a same protected Deterministic Request

- The Deterministic Request is sent to the server

Client side (1/2)

- Protect message M using the pairwise mode of Group OSCORE

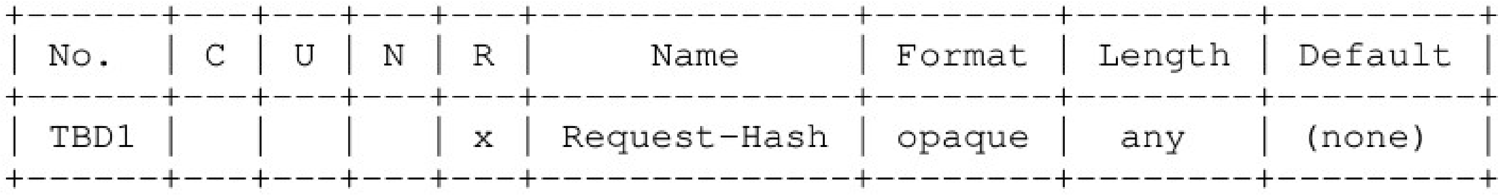
1. Prepare the OSCORE option like a pairwise request from C*
   - Use 0 as Partial IV because it is freshly derived

2. Compute a hash H using *h*(·) as
   - H = h(Sender Key of C* || AAD || COSE plaintext)

3. Derive the pairwise encryption key K as
   - K = HKDF(Sender Key of C*, H , info, L)   *// No real Diffie-Hellman secret here*
   - but otherwise just acts like a pairwise request here

Client side (2/2)

4. Include the new Request-Hash option: class U, value set to H



No.	C	U	N	R	Name	Format	Length	Default
TBD1				x	Request-Hash	opaque	any	(none)

5. In the AAD, set the value of ‘request_kid’ to H.
   - or CBOR [gid, H]? Structured ID Context being discussed.

6. Encrypt M, using the deterministic pairwise key K.

7. Set FETCH as outer code, even if Observe is not used.

8. Send the resulting Deterministic Request.

Server side (1/2)

1. Recognize what received as a Deterministic Request
   - ‘kid’ as Sender ID of the Deterministic Client C*
   - Presence of the Request-Hash option

2. Retrieve the hash H from the Request-Hash option

3. Derive the pairwise decryption key K like the client did

4. Decrypt using the pairwise mode of Group OSCORE and key K
   - In the AAD, set the value of ‘request_kid’ to H.
   - Do not perform replay checks (safety checks: see below)

5. Perform additional checks:
   - Recompute the hash. If different from H → unprotected 4.00.
   - Is the request REST-safe, without side effects? If not → protected 4.01.

Server side (2/2)

1. Set Max-Age as appropriate

2. Protect the response with the group mode of Group OSCORE
   - Use own Sender Sequence Number, set as Partial IV in the OSCORE option
   - In the AAD, set the value of ‘request_kid’ to H.

3. Set 2.05 as outer code, as it answers a FETCH.

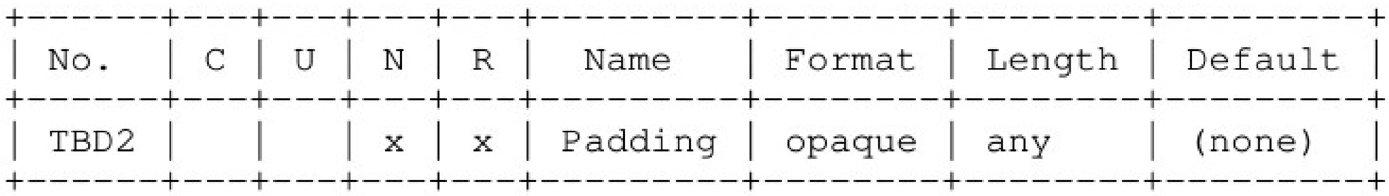
The client expects the response from a specific ‘kid’

- Check that against the ‘kid’ in the response, if included

Side features

- The Deterministic Request can be sent over IP multicast
  - Then each response MUST include the ‘kid’ of the replying server

- Traffic monitoring: easy to notice changes in the resource size
  - This can be a new privacy concern as now requests can be categorized
  - Handle with a new Padding option: class U, any content with any length
  - The client adds it to its request; the proxy may add more
  - The server ignores the option



No.	C	U	N	R	Name	Format	Length	Default
TBD2			x	x	Padding	opaque	any	(none)

Security compared to OSCORE

- Freshness is lost, including request replays
  - Relative freshness is still available
- Request confidentiality is limited
  - Identical requests have the same ciphertext
- Source authentication for clients is lost
  - But the server checks for whether the code is safe

- Loss of these properties is inherently necessary for untrusted caches
  - Source authentication for clients could be salvaged at great cost, but replay issue would make it useless
- All other properties should remain intact

Open point

- The server receives a deterministic request. Then:
  - The decryption succeeds, but …
  - The recomputed hash is different than the received one

- A 4.00 unprotected response follows
  - Deviation from usual constant-time code path
  - It tells that a forged authentication tag was correct

- Is this an actual problem here?
  - The deterministic encryption key is used only for this message

Next steps

- Polish the editor’s copy
  - https://gitlab.com/chrysn/core-cachable-oscore

- Submit version -01

- Implementation
  - Ready in aiocoap: https://github.com/chrysn/aiocoap
  - Partial embedded implementation being compared on IoT Lab
  - One more planned for Californium

Thank you!

Comments/questions?