Object Security for Contained RESTful Environments (OSCORE)

Kenny Vu, Kobe Luong, Moshe Goldstein, Shantanu Biswas

**Abstract—Object Security for Constrained RESTful Environments, otherwise known as OSCORE, is defined as an approach to carry out application-layer protection of the Constrained Application Protocol (CoAP) by utilizing CBOR Object Signing and Encryption (COSE). It allows for end-to-end protection between multiple communicating endpoints by implementing CoAP or CoAP-mappable HTTP. Generally, OSCORE is meant for constrained nodes and networks which support many different proxy operations, including different transport protocols. Optional functionalities of OSCORE include its ability to translate between different transport protocols and the alteration of CoAP options processing and IANA registration. OSCORE is incredibly useful for lightweight networks because of its small message size and memory requirements. This paper will analyze various components of OSCORE in depth including fundamentals of the COSE object, the definition of security context, protected message fields, message binding, processing and HTTP operations, and OSCORE implementation.**

***Index Terms*—OSCORE, COSE, Security Context,**

# INTRODUCTION

The Constrained Application Protocol (CoAP) is defined as a specialized web transfer protocol that is used with constrained nodes and constrained networks such as low-power and lossy networks. The nodes usually contain 8-bit microcontrollers with miniscule amounts of ROM and RAM. CoAP gives a request/response interaction model between application endpoints, aids in the discovery of services and resources, and includes significant concepts of the web such as URIs and internet media types. It is designed to efficiently interface with HTTP integration with the web while simultaneously fulfilling alternative requirements such as multicast support, extremely low overhead, and simplicity for constrained environments.

OSCORE acts as an extension of CoAP and reuses CoAP’s serialization of messages. It protects the application layer request/response messages between the endpoints including the payload which has the value associated with the indicated resource as well as the request method, identifier of the resource, and content format of the payload. Essentially, it operates on a plaintext CoAP message and creates an OSCORE protected CoAP message.

# THE COSE OBJECT

The COSE Object is a very important part in the OSCORE process. This part of the paper will be talking about how COSE works with OSCORE. Essentially OSCORE requires that a client establishes a shared security context with the server. COSE is used to wrap and protect data in the original message and provide integrity and confidentiality. COSE is CBOR Object Signing and Encryption. CBOR is Concise Binary Object Representation, which is just a data format and makes the encoder and decoder small as well as maintaining a small message size. COSE is structured so that it can take a large amount of common code and parse then process the different types of security messages. COSE structure is built using the concept of layers for COSE\_Encrypt the message type can be separated into two layers: the content layer and the recipient layer, which contains the encrypted content encryption key. “COSE is functionally equivalent to the JSON Object Signing Encryption standard (JOSE) [1]. The unique thing about COSE is that it is optimized for constrained devices and uses binary encodings other than Base64 like JOSE. The binary encoding helps minimize the message.

## Encryption

OSCORE also uses COSE\_Encrypt0 with AEAD algorithm. AEAD is Authenticated Encryption with Associated Data cipher that is used for the bulk of encryption. Also, for TLS 1.3 protocol it mandates AEAD to be the bulk of encryption. AEAD brings two parts to OSCORE and anything that it is used in, which is integrity and confidentiality. What happens in AEAD is the message is encrypted and then a MAC (Message Authentication Control) is attached to the encrypted message. This is done to verify the integrity of the encrypted message. In OSCORE the “length of the sender key and recipient key is 128 bits; the length of AEAD nonce and Common IV is 13 bytes [1]. Another thing to note is that for the plaintext in OSCORE it is formatted as a CoAP message with a subset of the header. There are going to be five content key distribution methods that CORE uses that are influenced in the distribution of OSCORE. The five methods are direct, symmetric key-encryption keys, key agreement, key transport, passwords.

## ID Context and Signature

One thing that needs to happen with OSCORE is ID context and ‘kid context’. The reason why is because there are requirements on the security context parameters. The requirements are that the sender’s keys must be unique, which are the Master Secret, Master Salt, ID Context, and Sender ID. These requirements guarantee unique (key, nonce) pairs for the AEAD. The kid context is used to transport the ID context in request. On the other hand, the kid parameter is used to transport the Sender ID. If the client sends a request that has an ID context but without a kid context it will result in the server error not being able to find the security context. With CORE signing objects it supports one or more signatures to be applied on the same content. In CORE to create a signature the first thing is to create a Sig\_structure with the correct fields. After that, the Sig\_structure needs to be encoded with a byte string to create the ToBeSigned value. Lastly call the signature algorithm to receive a signature value. For the verifying step, the application will “perform the appropriate checks to ensure that the key is correctly paired with the signing identity and that the signing identity is authorized before performing actions” [9].

## Implementation

There was a reason why OSCORE was created because homes, facilities and cities are using a tremendous amount of power and resource requirements. What IETF (Internet Engineering Task Force) created was “a number of improvements in securing messages sent using CoAP, one of the preferred communication protocols used in LPWA” [8]. The major benefits that OSCORE provides is that it only encrypts the data part of the payload. This will decrease the overhead of the protocol, which will result in increasing bandwidth usage and battery lifetime. The second major benefit that OSCORE provides is it takes away the key negotiation part, which takes way too many resources for the constrained devices. This will help with the problem that constrained devices regularly exchange massive amounts of data. OSCORE was implemented by the Swedish Government Innovation Agency that tests end-to-end security and they collaborated with one of the contributors of the protocol from RISE (Research Institutes of Sweden).[8]

According to Mats Andersson, these are the results that were observed during the experiment. The mesh nodes handled packet fragmentation robustly with their mesh network office that has a maximum transmission unit of 12 bytes. OSCORE was also able to identify all the basic security threats when they were present. Also here are some of the highlights of OSCORE:

1. OSCORE protects each payload using pre-shared keys, which eliminates the key negotiation and session management.
2. OSCORE does not encrypt the messages completely. For OSCORE the metadata is left in the clear, which would save resources, power, and bandwidth.
3. When testing they observed that OSCORE was encrypting messages and only taking about 11-13 bytes for a message. This compared to other internet of things protocols is significantly smaller, which makes it an ideal security option.
4. With the metadata being sent in plaintext while the actual data is still encrypted. Makes it efficient because OSCORE allows for HTTP-CoAP translation. This means that “The proxy device converts a HTTP request into a CoAP request allowing powerful HTTP servers/clients to talk securely to resource constrained IoT CoAP clients/servers” [8].

# THE SECURITY CONTEXT

## Definition of Security Context

Security context is defined as the “set of information elements” that are required to carry out the cryptographic operations in OSCORE(Object Security for Contained RESTful Environments) [1]. OSCORE is for “end-to-end” encryption and “data integrity” [2]. The required data for this security protocol consists of the “‘Common Context,’ a ‘Sender Context,’ and a ‘Recipient Context’” [1]. Both communicating parties will use the Sender Context and the Recipient Context. The Sender Context protects sent “messages” [1]. Calculation of this “Sender Context” happens by performing operations on the “Sender ID” [1]. The Recipient Context, on the other hand, verifies messages “received” [1]. The Recipient Context comes from the “Recipient ID” [1]. When the endpoints communicate with each other, the Sender Context of either endpoint “matches” the other endpoint’s Recipient Context [1]. The “Common Context” aids in deriving the Sender and Recipient ones [1]. This “Common Context,” along with the Sender and Recipient Contexts, also contain various keys, algorithms, IDs, and other pieces of data that aid in OSCORE protection.

* “The Common Context contains the following parameters:
  + AEAD Algorithm. The COSE AEAD algorithm to use for encryption.
  + HKDF Algorithm. An HMAC-based key derivation function (HKDF, [RFC5869]) used to derive the Sender Key, Recipient Key, and Common IV.
  + Master Secret. Variable length, random byte string used to derive AEAD keys and Common IV.
  + Master Salt. Optional variable-length byte string containing the salt used to derive AEAD keys and Common IV.
  + ID Context. Optional variable-length byte string providing additional information to identify the Common Context and to derive AEAD keys and Common IV.
  + Common IV. Byte string derived from the Master Secret, Master Salt, and ID Context. Used to generate the AEAD nonce.
* The Sender Context contains the following parameters:
  + Sender ID. Byte string used to identify the Sender Context, to derive AEAD keys and Common IV, and to contribute to the uniqueness of AEAD nonces. Maximum length is determined by the AEAD Algorithm.
  + Sender Key. Byte string containing the symmetric AEAD key to protect messages to send. Derived from Common Context and Sender ID. Length is determined by the AEAD Algorithm.
  + Sender Sequence Number. Non-negative integer used by the sender to enumerate requests and certain responses, e.g. Observe notifications. Used as "Partial IV" [RFC8152] to generate unique AEAD nonces. Maximum value is determined by the AEAD Algorithm.
* The Recipient Context contains the following parameters:
  + Recipient ID. Byte string used to identify the Recipient Context, to derive AEAD keys and Common IV, and to contribute to the uniqueness of AEAD nonces. Maximum length is determined by the AEAD Algorithm.
  + Recipient Key. Byte string containing the symmetric AEAD key to verify messages received. Derived from Common Context and Recipient ID. Length is determined by the AEAD Algorithm.
  + Replay Window (Server only). The replay window used to verify requests received” [1]

## Security Context Establishments and Requirements

### Establishments

There are some pieces of data that need to be established beforehand by the communicating parties before OSCORE occurs. These 3 pieces of data are:

* “Master Secret
* Sender ID
* Recipient ID” [1].

Some other pieces of data may be “pre-established” by the endpoints, but this is not necessary [1].

### Requirements

In terms of requirements, the “Master Secret, Master Salt, ID Context, and Sender ID” *must* be unique in order to have unique “Sender Keys” [1].

# PROTECTED MESSAGE FIELDS

## CoAP Options

The majority of CoAP options are associated with inner message fields and have inner values; these are referred to as class E options. Options in class E are always encrypted and integrity protected between endpoints if present, and thus similar to communicating in a protected manner directly with the endpoint. The sending endpoint should write the class E option from the original CoAP message into the COSE object plaintext. Unless in special circumstances, the sending endpoint does not use the outer options of class E. Most of the time, the receiving endpoint discards outer options from the OSCOAP message and writes in the class E options present in the plaintext of the COSE object into the decrypted CoAP message. Inner Max-Age options, like its class E contemporaries, is used as defined in RFC7252 as long as it isn’t accessible to proxies. To prevent unnecessary caching from occurring, a server could possibly add an outer Max-Age option with value of zero to OSCOAP responses, and these outer Max-Age options are not integrity protected [5].

As described in RFC7959, Blockwise is an optional feature. The Block options are Block1, Block2, Size1, and Size2. They could be only inner options, only outer options, or both inner and outer options that are processed on their own. Inner block options are utilized to stabilize fragmentation of the payload into blocks and secure the protection of informative aspects about the fragmentation such as block number, block size, and last block. Outer block options are used to break apart any CoAp message or OSCOAP message should a CoAP proxy carry out block fragmentation. For this reason, outer block options aren’t encrypted or integrity protected. Similar to Blockwise, Observe (RFC7641) is also an optional feature. Observe always has an outer value to allow the proxy to support forwarding of Observe messages. This option is encoded in the OSCOAP request and integrity protected through inclusion in the external\_aad of the response with value set to the 3 least significant bytes of the Sequence Number of the response [5].

Class I options are outer options that are visible in the options portion of the OSCOAP message. Most of the time, the option value is integrity protected between endpoints and the sending endpoint usually encodes it in the OSCOAP message. Class U options are also outer values, but they support forward proxy operations. Usually, the sending endpoint encodes the class U options in the options component of the OSCOAP message similarly to Class I options [5].

## CoAP Header

Generally, CoAP header fields are required to be read and modified during a normal message exchange or when traversing a proxy. For this reason, they can’t be protected between endpoints. The header field code is required to be in plaintext to support RESTful processing, but also has to be integrity protected so that the intermediary doesn’t change from GET to DELETE. The CoAP version number must also be integrity protected. The other CoAP header fields are neither integrity protected nor encrypted. The sending endpoint will copy header fields from the original CoAP message to the header of the OSCOAP message [5].

## CoAP Payload

The CoAP payload is encrypted and integrity protected through Class E; it is an inner message field. The sending endpoint will write the payload of the original CoAP message into the COSE object’s plaintext. Then, the receiving endpoint verifies and decrypts the COSE object and recreates the payload of the original CoAP message [5].

## Signaling Messages

Signalling messages transfer information about the connection. They compose a third basic type of message in CoAP besides responses and requests. Message class 7 is utilized for these kinds of messages. Signaling messages are pertinent only for the connection they appear in [6].

# MESSAGE BINDING

OSCORE binds responses to requests to prevent response delay and mismatch attacks from on-path attackers. This is carried out by including ‘kid’ and Partial IV of the request in the AAD of the response, which need to be stored until all responses have been sent [4].

## Sequence Numbers & Maximum Sequence Number

### Sequence Numbers

AEAD nonces should not be used more than one time for each AED key. The uniqueness of the key depends on the correct use of Partial IVs, which are used to encode sender sequence numbers. When messages are processed at the same time, reading and increasing the sender sequence number is always atomic.

### Maximum Sequence Number

The maximum sender sequence number is algorithm dependent and always less than 2^40. If this maximum is exceeded, the endpoint cannot process any more messages with the given sender context. In this case, the endpoint should get a new security context before moving forward.

## Freshness

OSCORE only guarantees that the request is not older than its respective security context. If stronger demands on request freshness are necessary, OSCORE must be augmented with mechanisms providing freshness.

An honest server allows for message binding to guarantee that a response is not older than its request. Responses that are not notifications have absolute freshness. Those that are notifications acquire weaker freshness gradually as time goes on and it’s recommended that the client periodically re-register the observation

# PROCESSING and HTTP OPERATIONS

## Protecting and Verifying the Messages

OSCORE relies on the CoAP message processing implementation. Once a CoAP message is requested, the client uses that to create an OSCORE request. A client’s request is primarily protected through encrypting the COSE object using the sender’s key. The requested message is formatted per its specifications[4].

The verification is then inverse to the protection. The server will receive the OSCORE option and break it down into its individual components. In particular, the Recipient Key in the ‘kid’ parameter to verify. The server then decomposes the contents and decrypts the COSE object using the Recipient Key. The server will also need to protect its response using a similar process[4].

## Encryption of the COSE Object

Concise Binary Object Representation is a data format designed for small code size and small message size. OSCORE uses it for constrained services.

Common parameters are:

* Kty: identifies family of keys
* Alg: restricts the algorithm that is used with the key
* Kid: gives identifier for the key
* Key\_ops: restricts set of operations a key is used for
* Base IV: Carries the base portion of IV

Those are the main important fields for COSE encryption. The rest of the fields are typical to standard data encryption and are used[9].

## HTTP OSCORE Header Fields

The header fields may be mapped to HTTP and vice versa. The field value is standard base 64url-char. These messages must be kept in its specified OSCORE header field. Doing otherwise will render the message unusable. Redirects are not defined in the mappings. Instead a proxy is used to redirect the request.

CoAP-to-HTTP mapping places the value of CoAP in base64url encoding without padding. It adds in AA if the OSCORE option is empty. The CoAP Content-Type is set to ‘application/oscore’. HTTP-to-CoAP mapping needs to specify the behavior of the proxy. It has the value HTTP header field decoded from base64url without padding. OSCore can be originated or terminated at either HTTP endpoints. The sending endpoint relies on RFC8075 to translate the HTTp message into a CoAP message. The CoAP message is then processed and mapped to HTTP[4].

The receiving HTTP endpoint maps the HTTP message to a CoAP message also using RFC8075. The message is then processed and translated to HTTP in the normal endpoint[4].

## HTTP Client to Server Example

An HTTP client makes a request before client object security processing from the client to proxy initializing the communication. The application/oscore is posted to the proxy. The proxy receives the request and makes a CoAP request to the CoAP server. The CoAP server responds to the request and the message is sent to the proxy. The message is then sent back to the HTTP client. The HTTP client now does object security processing[4].

## CoAP Client and HTTP Server

A CoAP client makes the request before object security processing. The request is posted to the proxy. The proxy makes a HTTP request to the HTTP server. The HTTP server does object security processing. The response is sent back to the proxy. The proxy sends the response to the CoAP client. The client does object security processing to the response[4].

## OSCORE Implementation

Data is sent from an internet of things sensor to the cloud. OSCore is based on symmetric pre-shared keys. First, the data is encrypted with OSCORE using cryptographic information pre-shared during setup. Only the data is encrypted[3]. The metadata is still left in the clear. Encrypted messages can be as small as 11 bytes which is suitable for restrained devices. Leaving parts of the data clear improves performance and efficiency of the transmissions between the sensor and the server while maintaining secure communications. The CoAP message is then sent in a standard Bluetooth Mesh status message.

During transmission, the data is stopped by a gateway device. The CoAP message including the OSCORE encrypted data is repacked into a UDP message and sent to the cloud. In all cases, the gateway and intermediate devices are blind to the data. Once the data arrives at the cloud server, it is decrypted using the cryptographic information pre-shared with the OSCORE sensor[8].

# CONCLUSION

OSCORE and CoAP’s efficiency together is an incredibly useful asset to carry out application level protection and provide encryption between communicating endpoints. It both minimizes performance impact and is flexible enough to support different trust models [3]. OSCORE and CoAP continue to grow and become even more secure through standardization work on securing group communication, authorization and access control in secured environments, and standardizing a lightweight authenticated key exchange [3].

# **References**

1. C. Amsuess, “OSCORE Implementation Guidance draft-amsuess-lwig-oscore-00.” Internet Engineering Task Force, Fremont, 29-Apr-2020. <https://tools.ietf.org/pdf/draft-amsuess-lwig-oscore-00.pdf>
2. “End-to-End Security for the Internet of Things,” OMA SpecWorks, 26-Feb-2020. [Online]. Available: https://omaspecworks.org/end-to-end-security-for-the-internet-of-things/. [Accessed: 04-Dec-2020].
3. F. Palombini, G. Selander, and J. Mattsson, “OSCORE: A look at the new IoT security protocol,” Ericsson.com, 30-Sep-2020. [Online]. Available: https://www.ericsson.com/en/blog/2019/11/oscore-iot-security-protocol. [Accessed: 04-Dec-2020].
4. G. Selander, J. Mattsson, F. Palombini, and L. Seitz, “Object Security for Constrained RESTful Environments (OSCORE).” Internet Engineering Task Force, Fremont, Jul-2019. <https://tools.ietf.org/pdf/rfc8613.pdf>
5. G. Selander, J. Mattsson, F. Palombini, and L. Seitz, “Object Security of CoAP (OSCOAP) draft-selander-ace-object-security-06.” Internet Engineering Task Force, Fremont, 11-Oct-2016. <https://tools.ietf.org/pdf/draft-selander-ace-object-security-06.pdf>
6. K. Hartke, “CoAP Signaling Messages draft-bormann-core-coap-sig-00.” Internet Engineering Task Force, Fremont, 10-Jun-2016. <https://tools.ietf.org/pdf/draft-bormann-core-coap-sig-00.pdf>
7. What Are AEAD Ciphers? DevCentral, 2018. <https://www.youtube.com/watch?v=od44W45sCQ4>
8. M. Andersson Senior Director of Technology, P. Karlsson Senior Director of Technology, and H. Vigneswaran Senior Wireless Engineer, “Trialing OSCORE for end-to-end IoT security in resource constrained devices,” *u-blox*, 30-Nov-2020. [Online]. Available: https://www.u-blox.com/en/blogs/tech/trialing-oscore-end-end-iot-security-resource-constrained-devices. [Accessed: 06-Dec-2020].
9. Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.