Communication Protocol for Sec-Flux

This document describes the communication protocol between the Sec-Flux middleware and its clients. Because the web server outside the enclave is an untrusted component, we cannot rely on plain HTTP for communicating securely between the enclave and its clients. It would have been possible to terminate HTTPS connections inside an SGX enclave [2], but we do not consider this approach as suitable for our system because of two main issues. The first is linked with TLS certificates validation on web browsers. For a browser to communicate with the enclave, it is required to accept certificates generated within the enclave and signed by the application owner. However, browsers are typically configured to use a chain of trust and accept a certificate as long as its root was signed by a Certificate Authority (CA) known to the browser. As a consequence, a malicious cloud owner could use their own certificate instead, as long as it is signed by a CA and is valid for the same domain. This would trick the client into opening a communication channel with the attacker instead of the enclave. The second reason why we used application-layer encryption is that it allows the untrusted server to perform load balancing on the requests. While we do not use load balancing in our current solution, that remains a possibility for future system development. For these reasons, we designed a simple application-layer protocol over HTTP using well-established web standards. The application and the clients first establish a session and then encrypt their messages using a symmetric key, referred to as the session key.

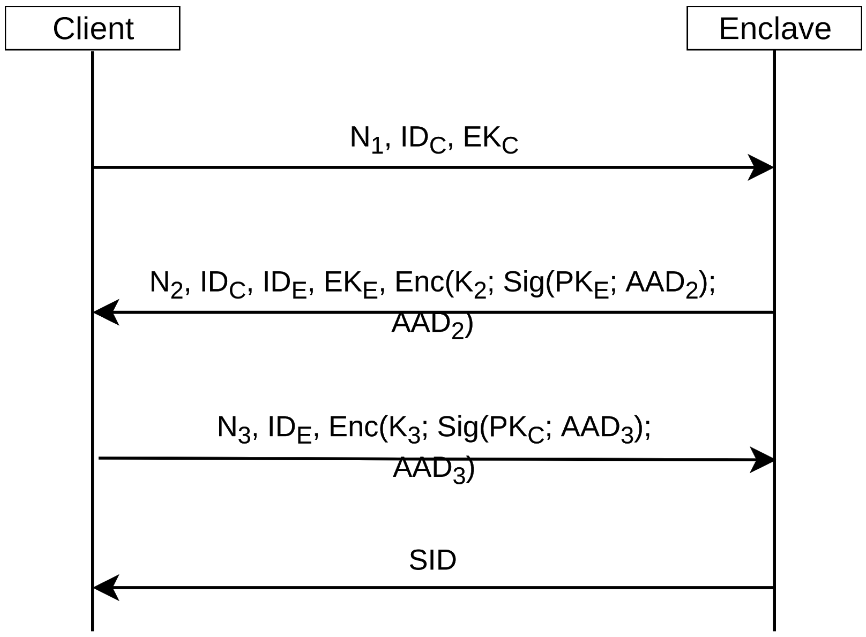
Client
Enclave
N_1, ID_C, EK_C
N_2, ID_C, ID_E, EK_E, Enc(K_2; Sig(PK_E; AAD_2); AAD_2)
N_3, ID_E, Enc(K_3; Sig(PK_C; AAD_3); AAD_3)
SID

Figure 1: The session establishment flow. *Sig(K_S;S)* represents a signature of data *S* using K_S as the signing key. *Enc(K_E;P;A)* represents the encryption of plaintext *P* with the key K_E and the AAD *A*.

Session Key

In order to exchange messages, a client and the system need to establish a session. The session key is generated using a SIGn and MAc (SIGMA) key exchange protocol using

Diffie-Hellman with Ephemeral keys (DHE). This type of key exchange allows for Perfect Forward Secrecy (PFS), which means that an attacker gaining access to the private key of one of the parties cannot decipher any message sent using previous session keys. Our key exchange protocol is an adaptation of the Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol draft [13], which defines a key exchange protocol for the Constrained Application Protocol (CoAP). This document is used as the basis to design our protocol because it does for CoAP what we want to achieve for HTTP. All messages for this exchange use a JSON format, and a JSON Web Key (JWK) structure is used to represent the ephemeral keys of each party. For now, our system only supports the key exchange using Elliptic Curves (ECDHE).

The exchange flow for the establishment of a session is shown in Figure 1. We will not detail the encoding of these messages but we will explain their content. The first message is sent unencrypted by the client and consists of a nonce value N1, an identifier for the exchange IDC, and the public component EKC of the ephemeral key generated by the client. This message is sent as a request to a specific endpoint that identifies the client.

The second message is contained in the enclave response to the first request. It consists of two parts: an unprotected one and an encrypted signature. The unprotected part consists of a nonce value N2, the exchange identifier of the client IDC, an exchange identifier for the enclave IDE , and the public component EKE of the ephemeral key of the enclave. The Additional Authenticated Data (AAD) value AAD2 is calculated by combining the first message and the unprotected part of the second message. This value is signed using the (permanent) private key of the enclave PKE . The shared secret obtained from the two ephemeral keys is then calculated, and an encryption key K2 and an Initialization Vector (IV) are derived from it, using AAD2 as context. The signature is finally encrypted using the generated key and IV.

After the client verifies the authenticity of the response using the enclave (permanent) public key, it sends the third message. Similar to the second message, it contains an unprotected part and an encrypted signature. The unprotected part consists of a nonce value N3 and the server exchange identifier IDE . A new value AAD3 is computed by combining AAD2 and the unprotected part of message 3. This AAD is signed using the (permanent) private key of the client PKC. A new encryption key K3 and a new IV are derived using AAD3 and are used to encrypt the generated signature.

Upon reception of the third message, the enclave is able to verify its authenticity using the permanent public key of the client. Finally, the session key is derived from the shared secret. The server response to the third message is the ID of the newly created session SID.

The identity of each client is known to the enclave in the form of a public key, provided by the system owner via the API. Similarly, the public key of the enclave is known to each client. The specification of this key provisioning to the clients is out of the scope of this project. Once a session has been established, the session key is kept in the enclave memory; it is never stored on disk, to ensure PFS.

Message Exchange

Once a session key has been generated, clients are able to communicate with the middleware. For now, the system only supports AES-GCM encryption with 256 bits keys. Base 64 URL encoding is used to format binary values, as well as URL-encoded strings. All requests use a JSON Web Encryption (JWE) structure to represent the message and are of the following form:

The "dir" value of the algorithm header field denotes the fact that the JWE is using a direct encryption and is a standard field of JWE. The "kid" field indicates which session key to use; it is also used to identify the client. The IV is a randomly-generated 12 bytes nonce used by AES encryption. The tag is a 16 bytes value used to authenticate the ciphertext. The plaintext corresponding to the ciphertext consists of the request payload. In addition, we used an AAD containing all request information passed to the enclave by the server, such as the HTTP method, the request headers, and its URI. Verifying this information is crucial since it is passed from an untrusted source. Encoding them in the AAD instead of replicating them in the plaintext reduces significantly the message size. The JWE can be either used as the request payload, or encoded as a JSON Web Token (JWT) in the request URI if the request does not contain a payload (e.g. GET or DELETE).

Similarly to the request, the server response is a JWE structure where the plaintext corresponds to the enclave response message. The AAD used for the response JWE consist of the response headers, the response status code, and the tag of the request JWE. This allows the client to match the server response with its request. The JWE is either used as the request response body or encoded as a JWT in a custom response header.