Code of Local Attestation

> 由于文件数量较多，忽略各头文件

App

App.cpp

```
#include <stdio.h>
#include <map>
#include <assert.h>

#include "sgx_eid.h"
#include "sgx_urts.h"

#include "EnclaveInitiator_u.h"
#include "EnclaveResponder_u.h"
#include "sgx_utils.h"

#define ENCLAVE_INITIATOR_NAME "libenclave_initiator.signed.so"
#define ENCLAVE_RESPONDER_NAME "libenclave_responder.signed.so"

sgx_enclave_id_t initiator_enclave_id = 0, responder_enclave_id = 0;

int main(int argc, char* argv[])
{
    int update = 0;
    uint32_t ret_status;
    sgx_status_t status;
    sgx_launch_token_t token = {0};

    (void)argc;
    (void)argv;


    // load initiator and responder enclaves
    if (SGX_SUCCESS != sgx_create_enclave(ENCLAVE_INITIATOR_NAME, SGX_DEBUG_FLAG,
&token, &update, &initiator_enclave_id, NULL)
        || SGX_SUCCESS != sgx_create_enclave(ENCLAVE_RESPONDER_NAME,
SGX_DEBUG_FLAG, &token, &update, &responder_enclave_id, NULL)) {
        printf("failed to load enclave...\n");
        goto destroy_enclave;
    }
    printf("succeed to load enclaves...\n");



    // create ECDH session using initiator enclave, it would create session with
responder enclave
    status = create_session_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
```

```
        printf("failed to establish secure channel: ECALL return 0x%x, error code
is 0x%x.\n", status, ret_status);
        goto destroy_enclave;
    }
    printf("succeed to establish secure channel.\n");

    // test message exchanging between initiator enclave and responder enclave
    status = message_exchange_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
        printf("test_message_exchange Ecall failed: ECALL return 0x%x, error code
is 0x%x.\n", status, ret_status);
        goto destroy_enclave;
    }
    printf("Succeed to exchange secure message...\n");

    // close ECDH session
    status = close_session_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
        printf("test_close_session Ecall failed: ECALL return 0x%x, error code is
0x%x.\n", status, ret_status);
        goto destroy_enclave;
    }
    printf("Succeed to close Session...\n");

    destroy_enclave:
    sgx_destroy_enclave(initiator_enclave_id);
    sgx_destroy_enclave(responder_enclave_id);

    printf("Destroyed enclaves\n");

    return 0;
}
```

UntrustedEnclaveMessageExchange.cpp

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "sgx_eid.h"
#include "error_codes.h"
#include "datatypes.h"
#include "sgx_urts.h"
#include "UntrustedEnclaveMessageExchange.h"
#include "sgx_dh.h"

#include "fifo_def.h"
#include "EnclaveResponder_u.h"

extern sgx_enclave_id_t responder_enclave_id;
extern "C"
uint32_t session_request_ocall(sgx_dh_msg1_t *dh_msg1, uint32_t *session_id) {
    uint32_t retcode;
    session_request_ecall(responder_enclave_id, &retcode, dh_msg1, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
```

```
}

uint32_t exchange_report_ocall(sgx_dh_msg2_t *dh_msg2, sgx_dh_msg3_t *dh_msg3,
uint32_t session_id) {
    uint32_t retcode;
    exchange_report_ecall(responder_enclave_id, &retcode, dh_msg2, dh_msg3,
session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}

uint32_t send_request_ocall(uint32_t session_id, secure_message_t *req_message,
size_t req_message_size,
                            size_t max_payload_size, secure_message_t
*resp_message, size_t resp_message_size) {
    uint32_t retcode;
    generate_response_ecall(responder_enclave_id, &retcode, req_message,
req_message_size, max_payload_size, resp_message,
                        resp_message_size, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}


uint32_t end_session_ocall(uint32_t session_id) {
    uint32_t retcode;
    end_session_ecall(responder_enclave_id, &retcode, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}
```

EnclaveInitiator (e1)

EnclaveInitiator.edl

```
enclave {
    include "sgx_eid.h"
    include "datatypes.h"
    include "dh_session_protocol.h"

    trusted{
            public uint32_t create_session_ecall();
            public uint32_t message_exchange_ecall();
            public uint32_t close_session_ecall();
    };

    untrusted{
            uint32_t session_request_ocall([out] sgx_dh_msg1_t *dh_msg1,[out]
uint32_t *session_id);
            uint32_t exchange_report_ocall([in] sgx_dh_msg2_t *dh_msg2, [out]
sgx_dh_msg3_t *dh_msg3, uint32_t session_id);
            uint32_t send_request_ocall(uint32_t session_id, [in, size =
req_message_size] secure_message_t* req_message, size_t req_message_size, size_t
max_payload_size, [out, size=resp_message_size] secure_message_t* resp_message,
size_t resp_message_size);
            uint32_t end_session_ocall(uint32_t session_id);
```

```
    };

};
```

EnclaveInitiator.cpp

```
// Enclave1: Defines the exported functions for the .so application
#include "sgx_eid.h"
#include "EnclaveInitiator_t.h"
#include "EnclaveMessageExchange.h"
#include "error_codes.h"
#include "Utility_E1.h"
#include "sgx_dh.h"
#include "sgx_utils.h"
#include <map>

#define UNUSED(val) (void)(val)

#define RESPONDER_PRODID 1

std::map<sgx_enclave_id_t, dh_session_t>g_src_session_info_map;

dh_session_t g_session;

// This is hardcoded responder enclave's MRSIGNER for demonstration purpose. The
content aligns to responder enclave's signing key
sgx_measurement_t g_responder_mrsigner = {
    {
        0x83, 0xd7, 0x19, 0xe7, 0x7d, 0xea, 0xca, 0x14, 0x70, 0xf6, 0xba, 0xf6,
0x2a, 0x4d, 0x77, 0x43,
        0x03, 0xc8, 0x99, 0xdb, 0x69, 0x02, 0x0f, 0x9c, 0x70, 0xee, 0x1d, 0xfc,
0x08, 0xc7, 0xce, 0x9e
    }
};

/* Function Description:
 *   This is ECALL routine to create ECDH session.
 *   When it succeeds to create ECDH session, the session context is saved in
g_session.
 * */
extern "C" uint32_t create_session_ecall()
{
        return create_session(&g_session);
}

/* Function Description:
 *   This is ECALL routine to transfer message with ECDH peer
 * */
uint32_t message_exchange_ecall()
{
    uint32_t ke_status = SUCCESS;
    uint32_t target_fn_id, msg_type;
    char* marshalled_inp_buff;
    size_t marshalled_inp_buff_len;
    char* out_buff;
    size_t out_buff_len;
```

```
    size_t max_out_buff_size;
    char* secret_response;
    uint32_t secret_data;

    target_fn_id = 0;
    msg_type = MESSAGE_EXCHANGE;
    max_out_buff_size = 50; // it's assumed the maximum payload size in response
message is 50 bytes, it's for demonstration purpose
    secret_data = 0x12345678; //Secret Data here is shown only for purpose of
demonstration.

    //Marshals the secret data into a buffer
    ke_status = marshal_message_exchange_request(target_fn_id, msg_type,
secret_data, &marshalled_inp_buff, &marshalled_inp_buff_len);
    if(ke_status != SUCCESS)
    {
        return ke_status;
    }

    //Core Reference Code function
    ke_status = send_request_receive_response(&g_session, marshalled_inp_buff,
                                                marshalled_inp_buff_len,
max_out_buff_size, &out_buff, &out_buff_len);
    if(ke_status != SUCCESS)
    {
        SAFE_FREE(marshalled_inp_buff);
        SAFE_FREE(out_buff);
        return ke_status;
    }

    //Un-marshal the secret response data
    ke_status = umarshal_message_exchange_response(out_buff, &secret_response);
    if(ke_status != SUCCESS)
    {
        SAFE_FREE(marshalled_inp_buff);
        SAFE_FREE(out_buff);
        return ke_status;
    }

    SAFE_FREE(marshalled_inp_buff);
    SAFE_FREE(out_buff);
    SAFE_FREE(secret_response);
    return SUCCESS;
}


/* Function Description:
 *   This is ECALL interface to close secure session*/
uint32_t close_session_ecall()
{
    uint32_t ke_status = SUCCESS;

    ke_status = close_session(&g_session);

    //Erase the session context
    memset(&g_session, 0, sizeof(dh_session_t));
    return ke_status;
}
```

```
/* Function Description:
 *   This is to verify peer enclave's identity.
 * For demonstration purpose, we verify below points:
 *   1. peer enclave's MRSIGNER is as expected
 *   2. peer enclave's PROD_ID is as expected
 *   3. peer enclave's attribute is reasonable: it's INITIALIZED'ed enclave; in
non-debug build configuration, the enclave isn't loaded with enclave debug mode.
 **/
extern "C" uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t*
peer_enclave_identity)
{
    if (!peer_enclave_identity)
        return INVALID_PARAMETER_ERROR;

    // check peer enclave's MRSIGNER
    if (memcmp((uint8_t *)&peer_enclave_identity->mr_signer,
(uint8_t*)&g_responder_mrsigner, sizeof(sgx_measurement_t)))
        return ENCLAVE_TRUST_ERROR;

    // check peer enclave's product ID and enclave attribute (should be
INITIALIZED'ed)
    if (peer_enclave_identity->isv_prod_id != RESPONDER_PRODID || !
(peer_enclave_identity->attributes.flags & SGX_FLAGS_INITTED))
        return ENCLAVE_TRUST_ERROR;

    // check the enclave isn't loaded in enclave debug mode, except that the
project is built for debug purpose
#if defined(NDEBUG)
    if (peer_enclave_identity->attributes.flags & SGX_FLAGS_DEBUG)
        return ENCLAVE_TRUST_ERROR;
#endif

    return SUCCESS;
}

/* Function Description: Operates on the input secret and generate the output
secret
 * */
uint32_t get_message_exchange_response(uint32_t inp_secret_data)
{
    uint32_t secret_response;

    //User should use more complex encryption method to protect their secret,
below is just a simple example
    secret_response = inp_secret_data & 0x11111111;

    return secret_response;

}

//Generates the response from the request message
/* Function Description:
 *   process request message and generate response
 * Parameter Description:
 *   [input] decrypted_data: this is pointer to decrypted message
 *   [output] resp_buffer: this is pointer to response message, the buffer is
allocated inside this function
```

```
 *   [output] resp_length: this points to response length
 * */
extern "C" uint32_t message_exchange_response_generator(char* decrypted_data,
                                                        char** resp_buffer,
                                                        size_t* resp_length)
{
    ms_in_msg_exchange_t *ms;
    uint32_t inp_secret_data;
    uint32_t out_secret_data;
    if(!decrypted_data || !resp_length)
    {
        return INVALID_PARAMETER_ERROR;
    }
    ms = (ms_in_msg_exchange_t *)decrypted_data;

    if(umarshal_message_exchange_request(&inp_secret_data,ms) != SUCCESS)
        return ATTESTATION_ERROR;

    out_secret_data = get_message_exchange_response(inp_secret_data);

    if(marshal_message_exchange_response(resp_buffer, resp_length,
out_secret_data) != SUCCESS)
        return MALLOC_ERROR;

    return SUCCESS;
}
```

EnclaveMessageExchange.cpp

```
#include "sgx_trts.h"
#include "sgx_utils.h"
#include "EnclaveMessageExchange.h"
#include "sgx_eid.h"
#include "error_codes.h"
#include "sgx_ecp_types.h"
#include "sgx_thread.h"
#include <map>
#include "dh_session_protocol.h"
#include "sgx_dh.h"
#include "sgx_tcrypto.h"
#include "../EnclaveInitiator/EnclaveInitiator_t.h"
//#include "LocalAttestationCode_t.h"

#ifdef __cplusplus
extern "C" {
#endif

uint32_t message_exchange_response_generator(char *decrypted_data, char
**resp_buffer, size_t *resp_length);
uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t
*peer_enclave_identity);

#ifdef __cplusplus
}
#endif
```

```
#define MAX_SESSION_COUNT  16

//number of open sessions
uint32_t g_session_count = 0;

uint32_t generate_session_id(uint32_t *session_id);

uint32_t end_session(sgx_enclave_id_t src_enclave_id);

//Array of open session ids
session_id_tracker_t *g_session_id_tracker[MAX_SESSION_COUNT];

//Map between the source enclave id and the session information associated with
that particular session
std::map<sgx_enclave_id_t, dh_session_t> g_dest_session_info_map;

//Create a session with the destination enclave
uint32_t create_session(dh_session_t *session_info) {
    sgx_dh_msg1_t dh_msg1;            //Diffie-Hellman Message 1
    sgx_key_128bit_t dh_aek;        // Session Key
    sgx_dh_msg2_t dh_msg2;            //Diffie-Hellman Message 2
    sgx_dh_msg3_t dh_msg3;            //Diffie-Hellman Message 3
    uint32_t session_id;
    uint32_t retstatus;
    sgx_status_t status = SGX_SUCCESS;
    sgx_dh_session_t sgx_dh_session;
    sgx_dh_session_enclave_identity_t responder_identity;

    if (!session_info) {
        return INVALID_PARAMETER_ERROR;
    }

    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    memset(&dh_msg1, 0, sizeof(sgx_dh_msg1_t));
    memset(&dh_msg2, 0, sizeof(sgx_dh_msg2_t));
    memset(&dh_msg3, 0, sizeof(sgx_dh_msg3_t));
    memset(session_info, 0, sizeof(dh_session_t));

    //Intialize the session as a session initiator
    status = sgx_dh_init_session(SGX_DH_SESSION_INITIATOR, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
    }

    //Ocall to request for a session with the destination enclave and obtain
session id and Message 1 if successful
    status = session_request_ocall(&retstatus, &dh_msg1, &session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }
    //Process the message 1 obtained from desination enclave and generate message
2
    status = sgx_dh_initiator_proc_msg1(&dh_msg1, &dh_msg2, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
```

```
    }

    //Send Message 2 to Destination Enclave and get Message 3 in return
    status = exchange_report_ocall(&retstatus, &dh_msg2, &dh_msg3, session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }

    //Process Message 3 obtained from the destination enclave
    status = sgx_dh_initiator_proc_msg3(&dh_msg3, &sgx_dh_session, &dh_aek,
&responder_identity);
    if (SGX_SUCCESS != status) {
        return status;
    }

    // Verify the identity of the destination enclave
    if (verify_peer_enclave_trust(&responder_identity) != SUCCESS) {
        return INVALID_SESSION;
    }

    memcpy(session_info->active.AEK, &dh_aek, sizeof(sgx_key_128bit_t));
    session_info->session_id = session_id;
    session_info->active.counter = 0;
    session_info->status = ACTIVE;
    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    return status;
}

//Request for the response size, send the request message to the destination
enclave and receive the response message back
uint32_t send_request_receive_response(dh_session_t *session_info,
                                       char *inp_buff,
                                       size_t inp_buff_len,
                                       size_t max_out_buff_size,
                                       char **out_buff,
                                       size_t *out_buff_len) {
    const uint8_t *plaintext;
    uint32_t plaintext_length;
    sgx_status_t status;
    uint32_t retstatus;
    secure_message_t *req_message;
    secure_message_t *resp_message;
    uint8_t *decrypted_data;
    uint32_t decrypted_data_length;
    uint32_t plain_text_offset;
    uint8_t l_tag[TAG_SIZE];
    size_t max_resp_message_length;
    plaintext = (const uint8_t *) (" ");
    plaintext_length = 0;

    if (!session_info || !inp_buff) {
        return INVALID_PARAMETER_ERROR;
    }
    //Check if the nonce for the session has not exceeded 2^32-2 if so end
session and start a new session
```

```
    if (session_info->active.counter == ((uint32_t) -2)) {
        close_session(session_info);
        create_session(session_info);
    }

    //Allocate memory for the AES-GCM request message
    req_message = (secure_message_t *) malloc(sizeof(secure_message_t) +
inp_buff_len);
    if (!req_message)
        return MALLOC_ERROR;
    memset(req_message, 0, sizeof(secure_message_t) + inp_buff_len);

    const uint32_t data2encrypt_length = (uint32_t) inp_buff_len;

    //Set the payload size to data to encrypt length
    req_message->message_aes_gcm_data.payload_size = data2encrypt_length;

    //Use the session nonce as the payload IV
    memcpy(req_message->message_aes_gcm_data.reserved, &session_info-
>active.counter,
           sizeof(session_info->active.counter));

    //Set the session ID of the message to the current session id
    req_message->session_id = session_info->session_id;

    //Prepare the request message with the encrypted payload
    status = sgx_rijndael128GCM_encrypt(&session_info->active.AEK, (uint8_t *)
inp_buff, data2encrypt_length,
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.payload)),
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.reserved)),
                                        sizeof(req_message-
>message_aes_gcm_data.reserved), plaintext, plaintext_length,
                                        &(req_message-
>message_aes_gcm_data.payload_tag));

    if (SGX_SUCCESS != status) {
        SAFE_FREE(req_message);
        return status;
    }

    //Allocate memory for the response payload to be copied
    *out_buff = (char *) malloc(max_out_buff_size);
    if (!*out_buff) {
        SAFE_FREE(req_message);
        return MALLOC_ERROR;
    }
    memset(*out_buff, 0, max_out_buff_size);

    //Allocate memory for the response message
    resp_message = (secure_message_t *) malloc(sizeof(secure_message_t) +
max_out_buff_size);
    if (!resp_message) {
        SAFE_FREE(req_message);
        return MALLOC_ERROR;
    }
```

```
    memset(resp_message, 0, sizeof(secure_message_t) + max_out_buff_size);

    //Ocall to send the request to the Destination Enclave and get the response
message back
    status = send_request_ocall(&retstatus, session_info->session_id,
req_message,
                                (sizeof(secure_message_t) + inp_buff_len),
max_out_buff_size,
                                resp_message, (sizeof(secure_message_t) +
max_out_buff_size));
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS) {
            SAFE_FREE(req_message);
            SAFE_FREE(resp_message);
            return ((uint32_t) retstatus);
        }
    } else {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return ATTESTATION_SE_ERROR;
    }

    max_resp_message_length = sizeof(secure_message_t) + max_out_buff_size;

    if (sizeof(resp_message) > max_resp_message_length) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return INVALID_PARAMETER_ERROR;
    }

    //Code to process the response message from the Destination Enclave

    decrypted_data_length = resp_message->message_aes_gcm_data.payload_size;
    plain_text_offset = decrypted_data_length;
    decrypted_data = (uint8_t *) malloc(decrypted_data_length);
    if (!decrypted_data) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return MALLOC_ERROR;
    }
    memset(&l_tag, 0, 16);

    memset(decrypted_data, 0, decrypted_data_length);

    //Decrypt the response message payload
    status = sgx_rijndael128GCM_decrypt(&session_info->active.AEK, resp_message-
>message_aes_gcm_data.payload,
                                        decrypted_data_length, decrypted_data,
                                        reinterpret_cast<uint8_t *>(&
(resp_message->message_aes_gcm_data.reserved)),
                                        sizeof(resp_message-
>message_aes_gcm_data.reserved),
                                        &(resp_message-
>message_aes_gcm_data.payload[plain_text_offset]),
                                        plaintext_length,
                                        &resp_message-
>message_aes_gcm_data.payload_tag);
```

```
    if (SGX_SUCCESS != status) {
        SAFE_FREE(req_message);
        SAFE_FREE(decrypted_data);
        SAFE_FREE(resp_message);
        return status;
    }

    // Verify if the nonce obtained in the response is equal to the session nonce
+ 1 (Prevents replay attacks)
    if (*((uint32_t *) resp_message->message_aes_gcm_data.reserved) !=
(session_info->active.counter + 1)) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        SAFE_FREE(decrypted_data);
        return INVALID_PARAMETER_ERROR;
    }

    //Update the value of the session nonce in the source enclave
    session_info->active.counter = session_info->active.counter + 1;

    memcpy(out_buff_len, &decrypted_data_length, sizeof(decrypted_data_length));
    memcpy(*out_buff, decrypted_data, decrypted_data_length);

    SAFE_FREE(decrypted_data);
    SAFE_FREE(req_message);
    SAFE_FREE(resp_message);
    return SUCCESS;
}

//Close a current session
uint32_t close_session(dh_session_t *session_info) {
    sgx_status_t status;
    uint32_t retstatus;

    if (!session_info) {
        return INVALID_PARAMETER_ERROR;
    }

    //Ocall to ask the destination enclave to end the session
    status = end_session_ocall(&retstatus, session_info->session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }
    return SUCCESS;
}

//Returns a new sessionID for the source destination session
uint32_t generate_session_id(uint32_t *session_id) {
    uint32_t status = SUCCESS;

    if (!session_id) {
        return INVALID_PARAMETER_ERROR;
    }
    //if the session structure is uninitialized, set that as the next session ID
    for (int i = 0; i < MAX_SESSION_COUNT; i++) {
```

```
        if (g_session_id_tracker[i] == NULL) {
            *session_id = i;
            return status;
        }
    }

    status = NO_AVAILABLE_SESSION_ERROR;

    return status;

}
```

EnclaveResponder (e2)

EnclaveResponder.edl

```
enclave {
    include "sgx_eid.h"
    include "datatypes.h"
    include "../Include/dh_session_protocol.h"
    trusted{
            public uint32_t session_request_ecall([out] sgx_dh_msg1_t *dh_msg1,
[out] uint32_t *session_id);
            public uint32_t exchange_report_ecall([in] sgx_dh_msg2_t *dh_msg2,
[out] sgx_dh_msg3_t *dh_msg3, uint32_t session_id);
            public uint32_t generate_response_ecall([in, size = req_message_size]
secure_message_t* req_message, size_t req_message_size, size_t max_payload_size,
[out, size=resp_message_size] secure_message_t* resp_message, size_t
resp_message_size, uint32_t session_id);
            public uint32_t end_session_ecall(uint32_t session_id);
    };
};
```

EnclaveResponder.cpp

```
// Enclave2 : Defines the exported functions for the DLL application
#include "sgx_eid.h"
#include "EnclaveResponder_t.h"
#include "EnclaveMessageExchange.h"
#include "error_codes.h"
#include "Utility_E2.h"
#include "sgx_dh.h"
#include "sgx_utils.h"
#include <map>

#define UNUSED(val) (void)(val)

std::map<sgx_enclave_id_t, dh_session_t>g_src_session_info_map;

// this is expected initiator's MRSIGNER for demonstration purpose
```

```
sgx_measurement_t g_initiator_mrsigner = {
        {
                0xc3, 0x04, 0x46, 0xb4, 0xbe, 0x9b, 0xaf, 0x0f, 0x69, 0x72, 0x84,
0x23, 0xea, 0x61, 0x3e, 0xf8,
                0x1a, 0x63, 0xe7, 0x2a, 0xcf, 0x74, 0x39, 0xfa, 0x05, 0x49, 0x00,
0x1f, 0xd5, 0x48, 0x28, 0x35
        }
};

/* Function Description:
 *   this is to verify peer enclave's identity
 * For demonstration purpose, we verify below points:
 *   1. peer enclave's MRSIGNER is as expected
 *   2. peer enclave's PROD_ID is as expected
 *   3. peer enclave's attribute is reasonable that it should be INITIALIZED and
without DEBUG attribute (except the project is built with DEBUG option)
 * */
extern "C" uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t*
peer_enclave_identity)
{
    if(!peer_enclave_identity)
        return INVALID_PARAMETER_ERROR;

    // check peer enclave's MRSIGNER
    if (memcmp((uint8_t *)&peer_enclave_identity->mr_signer,
(uint8_t*)&g_initiator_mrsigner, sizeof(sgx_measurement_t)))
        return ENCLAVE_TRUST_ERROR;

    if(peer_enclave_identity->isv_prod_id != 0 || !(peer_enclave_identity-
>attributes.flags & SGX_FLAGS_INITTED))
        return ENCLAVE_TRUST_ERROR;

    // check the enclave isn't loaded in enclave debug mode, except that the
project is built for debug purpose
#if defined(NDEBUG)
    if (peer_enclave_identity->attributes.flags & SGX_FLAGS_DEBUG)
        return ENCLAVE_TRUST_ERROR;
#endif

    return SUCCESS;
}

/* Function Description: Operates on the input secret and generates the output
secret */
uint32_t get_message_exchange_response(uint32_t inp_secret_data)
{
    uint32_t secret_response;

    //User should use more complex encryption method to protect their secret,
below is just a simple example
    secret_response = inp_secret_data & 0x11111111;

    return secret_response;

}

/* Function Description: Generates the response from the request message
 * Parameter Description:
```

```
 * [input] decrtyped_data: pointer to decrypted data
 * [output] resp_buffer: pointer to response message, which is allocated in this
function
 * [output] resp_length: this is response length */
extern "C" uint32_t message_exchange_response_generator(char* decrypted_data,
                                                        char** resp_buffer,
                                                         size_t* resp_length)
{
    ms_in_msg_exchange_t *ms;
    uint32_t inp_secret_data;
    uint32_t out_secret_data;

    if(!decrypted_data || !resp_length)
        return INVALID_PARAMETER_ERROR;

    ms = (ms_in_msg_exchange_t *)decrypted_data;

    if(umarshal_message_exchange_request(&inp_secret_data,ms) != SUCCESS)
        return ATTESTATION_ERROR;

    out_secret_data = get_message_exchange_response(inp_secret_data);

    if(marshal_message_exchange_response(resp_buffer, resp_length,
out_secret_data) != SUCCESS)
        return MALLOC_ERROR;

    return SUCCESS;
}
```

EnclaveMessageExchange.cpp

```
#include "sgx_trts.h"
#include "sgx_utils.h"
#include "EnclaveMessageExchange.h"
#include "sgx_eid.h"
#include "error_codes.h"
#include "sgx_ecp_types.h"
#include "sgx_thread.h"
#include <map>
#include "dh_session_protocol.h"
#include "sgx_dh.h"
#include "sgx_tcrypto.h"

#ifdef __cplusplus
extern "C" {
#endif

uint32_t enclave_to_enclave_call_dispatcher(char *decrypted_data, size_t
decrypted_data_length, char **resp_buffer,
                                            size_t *resp_length);
uint32_t message_exchange_response_generator(char *decrypted_data, char
**resp_buffer, size_t *resp_length);
uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t
*peer_enclave_identity);
```

```
#ifdef __cplusplus
}
#endif

#define MAX_SESSION_COUNT  16

//number of open sessions
uint32_t g_session_count = 0;

uint32_t generate_session_id(uint32_t *session_id);

extern "C" uint32_t end_session_ecall(uint32_t session_id);

//Array of open session ids
session_id_tracker_t *g_session_id_tracker[MAX_SESSION_COUNT];

//Map between the session id and the session information associated with that
particular session
std::map<uint32_t, dh_session_t> g_dest_session_info_map;

//Create a session with the destination enclave

//Handle the request from Source Enclave for a session
extern "C" uint32_t session_request_ecall(sgx_dh_msg1_t *dh_msg1,
                                          uint32_t *session_id) {
    dh_session_t session_info;
    sgx_dh_session_t sgx_dh_session;
    sgx_status_t status = SGX_SUCCESS;

    if (!session_id || !dh_msg1) {
        return INVALID_PARAMETER_ERROR;
    }
    //Intialize the session as a session responder
    status = sgx_dh_init_session(SGX_DH_SESSION_RESPONDER, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
    }

    //get a new SessionID
    if ((status = (sgx_status_t) generate_session_id(session_id)) != SUCCESS)
        return status; //no more sessions available

    //Allocate memory for the session id tracker
    g_session_id_tracker[*session_id] = (session_id_tracker_t *)
malloc(sizeof(session_id_tracker_t));
    if (!g_session_id_tracker[*session_id]) {
        return MALLOC_ERROR;
    }

    memset(g_session_id_tracker[*session_id], 0, sizeof(session_id_tracker_t));
    g_session_id_tracker[*session_id]->session_id = *session_id;
    session_info.status = IN_PROGRESS;

    //Generate Message1 that will be returned to Source Enclave
    status = sgx_dh_responder_gen_msg1((sgx_dh_msg1_t *) dh_msg1,
&sgx_dh_session);
    if (SGX_SUCCESS != status) {
```

```
        SAFE_FREE(g_session_id_tracker[*session_id]);
        return status;
    }
    memcpy(&session_info.in_progress.dh_session, &sgx_dh_session,
sizeof(sgx_dh_session_t));
    //Store the session information under the corresponding source enclave id key
    g_dest_session_info_map.insert(std::pair<uint32_t, dh_session_t>(*session_id,
session_info));

    return status;
}

//Verify Message 2, generate Message3 and exchange Message 3 with Source Enclave
extern "C" uint32_t exchange_report_ecall(sgx_dh_msg2_t *dh_msg2,
                                          sgx_dh_msg3_t *dh_msg3,
                                          uint32_t session_id) {

    sgx_key_128bit_t dh_aek;   // Session key
    dh_session_t *session_info;
    uint32_t status = SUCCESS;
    sgx_dh_session_t sgx_dh_session;
    sgx_dh_session_enclave_identity_t initiator_identity;

    if (!dh_msg2 || !dh_msg3) {
        return INVALID_PARAMETER_ERROR;
    }

    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    do {
        //Retrieve the session information for the corresponding source enclave
id
        std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
        if (it != g_dest_session_info_map.end()) {
            session_info = &it->second;
        } else {
            status = INVALID_SESSION;
            break;
        }

        if (session_info->status != IN_PROGRESS) {
            status = INVALID_SESSION;
            break;
        }

        memcpy(&sgx_dh_session, &session_info->in_progress.dh_session,
sizeof(sgx_dh_session_t));

        dh_msg3->msg3_body.additional_prop_length = 0;
        //Process message 2 from source enclave and obtain message 3
        sgx_status_t se_ret = sgx_dh_responder_proc_msg2(dh_msg2,
                                                         dh_msg3,
                                                         &sgx_dh_session,
                                                         &dh_aek,
                                                         &initiator_identity);
        if (SGX_SUCCESS != se_ret) {
            status = se_ret;
            break;
```

```
        }

        //Verify source enclave's trust
        if (verify_peer_enclave_trust(&initiator_identity) != SUCCESS) {
            return INVALID_SESSION;
        }

        //save the session ID, status and initialize the session nonce
        session_info->session_id = session_id;
        session_info->status = ACTIVE;
        session_info->active.counter = 0;
        memcpy(session_info->active.AEK, &dh_aek, sizeof(sgx_key_128bit_t));
        memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
        g_session_count++;
    } while (0);

    if (status != SUCCESS) {
        end_session_ecall(session_id);
    }

    return status;
}

//Process the request from the Source enclave and send the response message back
to the Source enclave
extern "C" uint32_t generate_response_ecall(secure_message_t *req_message,
                                            size_t req_message_size,
                                            size_t max_payload_size,
                                            secure_message_t *resp_message,
                                            size_t resp_message_size,
                                            uint32_t session_id) {
    const uint8_t *plaintext;
    uint32_t plaintext_length;
    uint8_t *decrypted_data;
    uint32_t decrypted_data_length;
    uint32_t plain_text_offset;
    ms_in_msg_exchange_t *ms;
    size_t resp_data_length;
    size_t resp_message_calc_size;
    char *resp_data;
    uint8_t l_tag[TAG_SIZE];
    size_t header_size, expected_payload_size;
    dh_session_t *session_info;
    secure_message_t *temp_resp_message;
    uint32_t ret;
    sgx_status_t status;

    plaintext = (const uint8_t *) (" ");
    plaintext_length = 0;

    if (!req_message || !resp_message) {
        return INVALID_PARAMETER_ERROR;
    }

    //Get the session information from the map corresponding to the source
enclave id
    std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
```

```
    if (it != g_dest_session_info_map.end()) {
        session_info = &it->second;
    } else {
        return INVALID_SESSION;
    }

    if (session_info->status != ACTIVE) {
        return INVALID_SESSION;
    }

    //Set the decrypted data length to the payload size obtained from the message
    decrypted_data_length = req_message->message_aes_gcm_data.payload_size;

    header_size = sizeof(secure_message_t);
    expected_payload_size = req_message_size - header_size;

    //Verify the size of the payload
    if (expected_payload_size != decrypted_data_length)
        return INVALID_PARAMETER_ERROR;

    memset(&l_tag, 0, 16);
    plain_text_offset = decrypted_data_length;
    decrypted_data = (uint8_t *) malloc(decrypted_data_length);
    if (!decrypted_data) {
        return MALLOC_ERROR;
    }

    memset(decrypted_data, 0, decrypted_data_length);

    //Decrypt the request message payload from source enclave
    status = sgx_rijndael128GCM_decrypt(&session_info->active.AEK, req_message-
>message_aes_gcm_data.payload,
                                        decrypted_data_length, decrypted_data,
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.reserved)),
                                        sizeof(req_message-
>message_aes_gcm_data.reserved),
                                        &(req_message-
>message_aes_gcm_data.payload[plain_text_offset]),
                                        plaintext_length,
                                        &req_message-
>message_aes_gcm_data.payload_tag);

    if (SGX_SUCCESS != status) {
        SAFE_FREE(decrypted_data);
        return status;
    }

    //Casting the decrypted data to the marshaling structure type to obtain type
of request (generic message exchange/enclave to enclave call)
    ms = (ms_in_msg_exchange_t *) decrypted_data;

    // Verify if the nonce obtained in the request is equal to the session nonce
    if (*((uint32_t *) req_message->message_aes_gcm_data.reserved) !=
session_info->active.counter ||
        *((uint32_t *) req_message->message_aes_gcm_data.reserved) > ((uint32_t)
-2)) {
        SAFE_FREE(decrypted_data);
```

```
        return INVALID_PARAMETER_ERROR;
    }

    if (ms->msg_type == MESSAGE_EXCHANGE) {
        //Call the generic secret response generator for message exchange
        ret = message_exchange_response_generator((char *) decrypted_data,
&resp_data, &resp_data_length);
        if (ret != 0) {
            SAFE_FREE(decrypted_data);
            SAFE_FREE(resp_data);
            return INVALID_SESSION;
        }
    } else {
        SAFE_FREE(decrypted_data);
        return INVALID_REQUEST_TYPE_ERROR;
    }


    if (resp_data_length > max_payload_size) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return OUT_BUFFER_LENGTH_ERROR;
    }

    resp_message_calc_size = sizeof(secure_message_t) + resp_data_length;

    if (resp_message_calc_size > resp_message_size) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return OUT_BUFFER_LENGTH_ERROR;
    }

    //Code to build the response back to the Source Enclave
    temp_resp_message = (secure_message_t *) malloc(resp_message_calc_size);
    if (!temp_resp_message) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return MALLOC_ERROR;
    }

    memset(temp_resp_message, 0, sizeof(secure_message_t) + resp_data_length);
    const uint32_t data2encrypt_length = (uint32_t) resp_data_length;
    temp_resp_message->session_id = session_info->session_id;
    temp_resp_message->message_aes_gcm_data.payload_size = data2encrypt_length;

    //Increment the Session Nonce (Replay Protection)
    session_info->active.counter = session_info->active.counter + 1;

    //Set the response nonce as the session nonce
    memcpy(&temp_resp_message->message_aes_gcm_data.reserved, &session_info-
>active.counter,
           sizeof(session_info->active.counter));

    //Prepare the response message with the encrypted payload
    status = sgx_rijndael128GCM_encrypt(&session_info->active.AEK, (uint8_t *)
resp_data, data2encrypt_length,
                                        reinterpret_cast<uint8_t *>(&
(temp_resp_message->message_aes_gcm_data.payload)),
```

```
                                        reinterpret_cast<uint8_t *>(&
(temp_resp_message->message_aes_gcm_data.reserved)),
                                        sizeof(temp_resp_message-
>message_aes_gcm_data.reserved), plaintext,
                                        plaintext_length,
                                        &(temp_resp_message-
>message_aes_gcm_data.payload_tag));

    if (SGX_SUCCESS != status) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        SAFE_FREE(temp_resp_message);
        return status;
    }

    memset(resp_message, 0, sizeof(secure_message_t) + resp_data_length);
    memcpy(resp_message, temp_resp_message, sizeof(secure_message_t) +
resp_data_length);

    SAFE_FREE(decrypted_data);
    SAFE_FREE(resp_data);
    SAFE_FREE(temp_resp_message);

    return SUCCESS;
}


//Respond to the request from the Source Enclave to close the session
extern "C" uint32_t end_session_ecall(uint32_t session_id) {
    uint32_t status = SUCCESS;
    int i;
    dh_session_t session_info;
    //uint32_t session_id;

    //Get the session information from the map corresponding to the source
enclave id
    std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
    if (it != g_dest_session_info_map.end()) {
        session_info = it->second;
    } else {
        return INVALID_SESSION;
    }

    //session_id = session_info.session_id;
    //Erase the session information for the current session
    g_dest_session_info_map.erase(session_id);

    //Update the session id tracker
    if (g_session_count > 0) {
        //check if session exists
        for (i = 1; i <= MAX_SESSION_COUNT; i++) {
            if (g_session_id_tracker[i - 1] != NULL && g_session_id_tracker[i -
1]->session_id == session_id) {
                memset(g_session_id_tracker[i - 1], 0,
sizeof(session_id_tracker_t));
                SAFE_FREE(g_session_id_tracker[i - 1]);
                g_session_count--;
```

```
                break;
            }
        }
    }

    return status;

}


//Returns a new sessionID for the source destination session
uint32_t generate_session_id(uint32_t *session_id) {
    uint32_t status = SUCCESS;

    if (!session_id) {
        return INVALID_PARAMETER_ERROR;
    }
    //if the session structure is uninitialized, set that as the next session ID
    for (int i = 0; i < MAX_SESSION_COUNT; i++) {
        if (g_session_id_tracker[i] == NULL) {
            *session_id = i;
            return status;
        }
    }

    status = NO_AVAILABLE_SESSION_ERROR;

    return status;

}
```