Code of Remote Attestation (Server)

App

app.cpp

```
#include <stdio.h>
#include <limits.h>
#include <unistd.h>
// Needed for definition of remote attestation messages.
#include "remote_attestation_result.h"

#include "isv_enclave_u.h"

// Needed to call untrusted key exchange library APIs, i.e. sgx_ra_proc_msg2.
#include "sgx_ukey_exchange.h"

// Needed to get service provider's information, in your real project, you will
// need to talk to real server.
#include "network_ra.h"

// Needed to create enclave and do ecall.
#include "sgx_urts.h"

// Needed to query extended epid group id.
#include "sgx_uae_service.h"

#include "service_provider.h"

#ifndef SAFE_FREE
#define SAFE_FREE(ptr)     \
    {                      \
        if (NULL != (ptr)) \
        {                  \
            free(ptr);     \
            (ptr) = NULL;  \
        }                  \
    }
#endif

// In addition to generating and sending messages, this application
// can use pre-generated messages to verify the generation of
// messages and the information flow.
#include "sample_messages.h"

#define ENCLAVE_PATH "isv_enclave.signed.so"

#define LENOFMSE 16

uint8_t *msg1_samples[] = {msg1_sample1, msg1_sample2};
uint8_t *msg2_samples[] = {msg2_sample1, msg2_sample2};
```

```
uint8_t *msg3_samples[] = {msg3_sample1, msg3_sample2};
uint8_t *attestation_msg_samples[] =
    {attestation_msg_sample1, attestation_msg_sample2};

// Some utility functions to output some of the data structures passed between
// the ISV app and the remote attestation service provider.
void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if (!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for (i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if (i % 8 == 7)
            fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}

void PRINT_ATTESTATION_SERVICE_RESPONSE(
    FILE *file,
    ra_samp_response_header_t *response)
{
    if (!response)
    {
        fprintf(file, "\t\n( null )\n");
        return;
    }

    fprintf(file, "RESPONSE TYPE:   0x%x\n", response->type);
    fprintf(file, "RESPONSE STATUS: 0x%x 0x%x\n", response->status[0],
            response->status[1]);
    fprintf(file, "RESPONSE BODY SIZE: %u\n", response->size);

    if (response->type == TYPE_RA_MSG2)
    {
        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)(response->body);

        fprintf(file, "MSG2 gb - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->g_b), sizeof(p_msg2_body->g_b));

        fprintf(file, "MSG2 spid - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->spid), sizeof(p_msg2_body->spid));

        fprintf(file, "MSG2 quote_type : %hx\n", p_msg2_body->quote_type);

        fprintf(file, "MSG2 kdf_id : %hx\n", p_msg2_body->kdf_id);

        fprintf(file, "MSG2 sign_gb_ga - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sign_gb_ga),
```

```
                        sizeof(p_msg2_body->sign_gb_ga));

        fprintf(file, "MSG2 mac - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->mac), sizeof(p_msg2_body->mac));

        fprintf(file, "MSG2 sig_rl - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sig_rl),
                        p_msg2_body->sig_rl_size);
    }
    else if (response->type == TYPE_RA_ATT_RESULT)
    {
        sample_ra_att_result_msg_t *p_att_result =
            (sample_ra_att_result_msg_t *)(response->body);
        fprintf(file, "ATTESTATION RESULT MSG platform_info_blob - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->platform_info_blob),
                        sizeof(p_att_result->platform_info_blob));

        fprintf(file, "ATTESTATION RESULT MSG mac - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->mac), sizeof(p_att_result->mac));

        fprintf(file, "ATTESTATION RESULT MSG secret.payload_tag - %u bytes\n",
                p_att_result->secret.payload_size);

        fprintf(file, "ATTESTATION RESULT MSG secret.payload - ");
        PRINT_BYTE_ARRAY(file, p_att_result->secret.payload,
                        p_att_result->secret.payload_size);
    }
    else
    {
        fprintf(file, "\nERROR in printing out the response. "
                       "Response of type not supported %d\n",
                response->type);
    }
}

extern char sendbuf[BUFSIZ]; //数据传送的缓冲区
extern char recvbuf[BUFSIZ]; //数据接受的缓冲区

int myaesencrypt(const ra_samp_request_header_t *p_msgenc,
                 uint32_t msg_size,
                 sgx_enclave_id_t id,
                 sgx_status_t *status,
                 sgx_ra_context_t context)
{
    if (!p_msgenc||
        (msg_size != LENOFMSE))
    {
        return -1;
    }
    int ret = 0;

    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE] = {0};
    uint8_t out_data[LENOFMSE] = {0};
    uint8_t testdata[LENOFMSE] = {0};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
```

```
    memcpy_s(p_data, LENOFMSE, p_msgenc, msg_size);
    do
    {
        ret = enclave_encrypt(
            id,
            status,
            p_data,
            LENOFMSE,
            out_data);
        fprintf(stdout, "\nD %d %d",id, *status);
        ret = enclave_encrypt(
            id,
            status,
            out_data,
            LENOFMSE,
            testdata);
        fprintf(stdout, "\nD %d %d",id, *status);

    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    fprintf(stdout, "\nData of Encrypt is\n");
    PRINT_BYTE_ARRAY(stdout, p_data, 16);
    fprintf(stdout, "\nData of Encrypted is\n");
    PRINT_BYTE_ARRAY(stdout, out_data, 16);
    PRINT_BYTE_ARRAY(stdout, testdata, 16);
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGENC;
    p_msg2_full->size = msg2_size;
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
```

```
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);

    return ret;
}

//原本设计为32字节的消息长度，前16个字节是token，但是由于时间关系直接解密了
int myaesdecrypt(const ra_samp_request_header_t *p_msgenc,
                 uint32_t msg_size,
                 sgx_enclave_id_t id,
                 sgx_status_t *status,
                 sgx_ra_context_t context)
{
    if (!p_msgenc ||
        (msg_size != LENOFMSE))
    {
        return -1;
    }
    int ret = 0;
    fprintf(stdout, "\nD %d %d",id, *status);
    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE] = {0};
    uint8_t out_data[LENOFMSE] = {0};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
    memcpy_s(p_data, LENOFMSE, p_msgenc, msg_size);
    do
    {
        ret = enclave_decrypt(
            id,
            status,
            p_data,
            LENOFMSE,
            out_data);
    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    if(ret != SGX_SUCCESS)
        return ret;
    fprintf(stdout, "\nData of Decrypt is\n");
    PRINT_BYTE_ARRAY(stdout, p_data, 16);
    fprintf(stdout, "\nData of Decrypted is\n");
    PRINT_BYTE_ARRAY(stdout, out_data, 16);
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGDEC;
    p_msg2_full->size = msg2_size;
    // The simulated message2 always passes.  This would need to be set
    // accordingly in a real service provider implementation.
```

```
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);
    fprintf(stdout, "\nSend Decrypt Data Done.");
    return ret;
}

int myaessetkey(const ra_samp_request_header_t *p_msgdec,
                uint32_t msg_size,
                sgx_enclave_id_t id,
                sgx_status_t *status,
                sgx_ra_context_t context)
{
    if (!p_msgdec  ||
        (msg_size != LENOFMSE * 2))
    {
        return -1;
    }
    int ret = 0;
    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE * 2] = {0};
    uint8_t out_data[LENOFMSE] =
{'K','E','Y','S','E','T','S','U','C','C','E','S'};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
    memcpy_s(p_data, msg_size, p_msgdec, msg_size);
    //应该调用isv_enclave_u.h中生成的函数
    do
    {
        ret = enclave_generate_key(
            id,
            status,
            p_data,
```

```
            msg_size);
    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    if(ret != SGX_SUCCESS)
        return ret;
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGSETKEY;
    p_msg2_full->size = msg2_size;
    // The simulated message2 always passes.  This would need to be set
    // accordingly in a real service provider implementation.
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);
    return ret;
}
// This sample code doesn't have any recovery/retry mechanisms for the remote
// attestation. Since the enclave can be lost due S3 transitions, apps
// susceptible to S3 transitions should have logic to restart attestation in
// these scenarios.
#define _T(x) x
int main(int argc, char *argv[])
{
    int ret = 0;
    ra_samp_request_header_t *p_msg0_full = NULL;
    ra_samp_response_header_t *p_msg0_resp_full = NULL;
    ra_samp_request_header_t *p_msg1_full = NULL;
```

```
    ra_samp_response_header_t *p_msg2_full = NULL;
    sgx_ra_msg3_t *p_msg3 = NULL;
    ra_samp_response_header_t *p_att_result_msg_full = NULL;
    sgx_enclave_id_t enclave_id = 0;
    int enclave_lost_retry_time = 1;
    int busy_retry_time = 4;
    sgx_ra_context_t context = INT_MAX;
    sgx_status_t status = SGX_SUCCESS;
    ra_samp_request_header_t *p_msg3_full = NULL;
    ra_samp_request_header_t *p_msgaes_full = NULL;

    int32_t verify_index = -1;
    int32_t verification_samples = sizeof(msg1_samples) /
sizeof(msg1_samples[0]);

    FILE *OUTPUT = stdout;
    ra_samp_request_header_t *p_req;
    ra_samp_response_header_t **p_resp;
    ra_samp_response_header_t *p_resp_msg;
    int server_port = 12333;
    int buflen = 0;
    uint32_t extended_epid_group_id = 0;
    { // creates the cryptserver enclave.

        ret = sgx_get_extended_epid_group_id(&extended_epid_group_id);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_get_extended_epid_group_id fail
[%s].",
                    __FUNCTION__);
            return ret;
        }
        fprintf(OUTPUT, "\nCall sgx_get_extended_epid_group_id success.");

        int launch_token_update = 0;
        sgx_launch_token_t launch_token = {0};
        memset(&launch_token, 0, sizeof(sgx_launch_token_t));
        do
        {
            ret = sgx_create_enclave(_T(ENCLAVE_PATH),
                                     SGX_DEBUG_FLAG,
                                     &launch_token,
                                     &launch_token_update,
                                     &enclave_id, NULL);
            if (SGX_SUCCESS != ret)
            {
                ret = -1;
                fprintf(OUTPUT, "\nError, call sgx_create_enclave fail [%s].",
                        __FUNCTION__);
                goto CLEANUP;
            }
            fprintf(OUTPUT, "\nCall sgx_create_enclave success.");

            ret = enclave_init_ra(enclave_id,
                                  &status,
                                  false,
                                  &context);
```

```
            //Ideally, this check would be around the full attestation flow.
        } while (SGX_ERROR_ENCLAVE_LOST == ret && enclave_lost_retry_time--);

        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_init_ra fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nCall enclave_init_ra success.");
    }

    //服务进程，对接受的数据进行响应
    fprintf(OUTPUT, "\nstart socket....\n");
    server(server_port);

    //如果接受的信息类型为服务类型，就解析
    do
    {
        //阻塞调用socket
        buflen = RecvfromCient();
        if (buflen > 0 && buflen < BUFSIZ)
        {
            p_req = (ra_samp_request_header_t *)malloc(buflen+2);

            fprintf(OUTPUT, "\nPrepare receive struct");
            if (NULL == p_req)
            {
                ret = -1;
                goto CLEANUP;
            }
            if (memcpy_s(p_req, buflen+ 2, recvbuf, buflen))
            {
                fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in
[%s].",
                        __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            //todo: 添加一个检查p_req的函数，由于时间紧张，就先放一放
            fprintf(OUTPUT, "\nrequest type is %d",p_req->type);
            switch (p_req->type)
            {
            //收取msg0，进行验证
            case TYPE_RA_MSG0:
                fprintf(OUTPUT, "\nProcess Message 0");
                ret = sp_ra_proc_msg0_req((const sample_ra_msg0_t *)((uint8_t
*)p_req + sizeof(ra_samp_request_header_t)),
                                          p_req->size);
                fprintf(OUTPUT, "\nProcess Message 0 Done");
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail
[%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
```

```
                break;
            //收取msg1，进行验证并返回msg2
            case TYPE_RA_MSG1:
                fprintf(OUTPUT, "\nBuffer length is %d\n", buflen);
                p_resp_msg = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t)+170);//简化处理
                memset(p_resp_msg, 0, sizeof(ra_samp_response_header_t)+170);
                fprintf(OUTPUT, "\nProcess Message 1\n");
                ret = sp_ra_proc_msg1_req((const sample_ra_msg1_t *)((uint8_t
*)p_req + sizeof(ra_samp_request_header_t)),
                                          p_req->size,
                                          &p_resp_msg);
                fprintf(OUTPUT, "\nProcess Message 1 Done");
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail
[%s].",
                            __FUNCTION__);
                }
                else
                {
                    memset(sendbuf, 0, BUFSIZ);
                    if (memcpy_s(sendbuf, BUFSIZ, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size))
                    {
                        fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed
in [%s].",
                                __FUNCTION__);
                        ret = -1;
                        goto CLEANUP;
                    }
                    fprintf(OUTPUT, "\nSend Message 2\n");
                    PRINT_BYTE_ARRAY(OUTPUT, p_resp_msg, 176);
                    int buflen = SendToClient(sizeof(ra_samp_response_header_t) +
p_resp_msg->size);
                    fprintf(OUTPUT, "\nSend Message 2 Done,send length = %d",
buflen);
                }
                SAFE_FREE(p_req);
                SAFE_FREE(p_resp_msg);
                break;
            //收取msg3，返回attestation result
            case TYPE_RA_MSG3:
                fprintf(OUTPUT, "\nProcess Message 3");
                p_resp_msg = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t)+200);//简化处理
                memset(p_resp_msg, 0, sizeof(ra_samp_response_header_t)+200);
                ret = sp_ra_proc_msg3_req((const sample_ra_msg3_t *)((uint8_t
*)p_req +

sizeof(ra_samp_request_header_t)),
                                          p_req->size,
                                          &p_resp_msg);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg3_req fail
[%s].",
                            __FUNCTION__);
```

```
                }
                else
                {
                    memset(sendbuf, 0, BUFSIZ);
                    if (memcpy_s(sendbuf, BUFSIZ, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size))
                    {
                        fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed
in [%s].",
                                __FUNCTION__);
                        ret = -1;
                        goto CLEANUP;
                    }
                    fprintf(OUTPUT, "\nSend attestation data\n");
                    PRINT_BYTE_ARRAY(OUTPUT, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size);
                    int buflen = SendToClient(sizeof(ra_samp_response_header_t) +
p_resp_msg->size);
                    fprintf(OUTPUT, "\nSend attestation data Done,send length =
%d", buflen);
                }
                SAFE_FREE(p_req);
                SAFE_FREE(p_resp_msg);
                break;

            //进行解密
            case TYPE_RA_MSGDEC:
                fprintf(OUTPUT, "\nProcess Decrypt");
                fprintf(OUTPUT, "\nDecrypt 1 %d %x",enclave_id, status);
                /*SGX_ERROR_MAC_MISMATCH 0x3001 Indicates verification error for
reports, sealed datas, etc */
                ret = myaesdecrypt((const ra_samp_request_header_t *)((uint8_t
*)p_req +

 sizeof(ra_samp_request_header_t)),
                                   p_req->size,
                                   enclave_id,
                                   &status,
                                   context);
                fprintf(OUTPUT, "\nDecrypt Done %d %d",enclave_id, status);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call decrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                goto CLEANUP;
            //进行加密
            case TYPE_RA_MSGENC:
                fprintf(OUTPUT, "\nProcess Encrypt");
                ret = myaesencrypt((const ra_samp_request_header_t *)((uint8_t
*)p_req +

 sizeof(ra_samp_request_header_t)),
                                   p_req->size,
                                   enclave_id,
                                   &status,
                                   context);
```

```
                fprintf(OUTPUT, "\nEncrypt Done %d %d",enclave_id, status);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call encrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                break;

            case TYPE_RA_MSGSETKEY:
                //本来的逻辑是验证数据是不是enclave传过来的, token
                fprintf(OUTPUT, "\nSet Key");
                ret = myaessetkey((const ra_samp_request_header_t *)((uint8_t
*)p_req +

sizeof(ra_samp_request_header_t)),
                                  p_req->size,
                                  enclave_id,
                                  &status,
                                  context);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call encrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                break;
            default:
                ret = -1;
                fprintf(stderr, "\nError, unknown ra message type. Type = %d
[%s].",
                        p_req->type, __FUNCTION__);
                break;
            }
        }
    } while (true);

CLEANUP:
    // Clean-up
    // Need to close the RA key state.
    if (INT_MAX != context)
    {
        int ret_save = ret;
        ret = enclave_ra_close(enclave_id, &status, context);
        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_ra_close fail [%s].",
                    __FUNCTION__);
        }
        else
        {
            // enclave_ra_close was successful, let's restore the value that
            // led us to this point in the code.
            ret = ret_save;
        }
        fprintf(OUTPUT, "\nCall enclave_ra_close success.");
    }
```

```
    sgx_destroy_enclave(enclave_id);

    ra_free_network_response_buffer(p_msg0_resp_full);
    ra_free_network_response_buffer(p_msg2_full);
    ra_free_network_response_buffer(p_att_result_msg_full);

    // p_msg3 is malloc'd by the untrusted KE library. App needs to free.
    SAFE_FREE(p_msg3);
    SAFE_FREE(p_msg3_full);
    SAFE_FREE(p_msg1_full);
    printf("\nExit ...\n");
    return ret;
}
```

Enclave

enclave.edl

```
enclave {
    from "sgx_tkey_exchange.edl" import *;

    include "sgx_key_exchange.h"
    include "sgx_trts.h"

    trusted {
        public sgx_status_t enclave_init_ra(int b_pse,
                                            [out] sgx_ra_context_t *p_context);
        public sgx_status_t enclave_ra_close(sgx_ra_context_t context);
        public sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                                  [in,size=message_size] uint8_t*
message,
                                                  size_t message_size,
                                                  [in,size=mac_size] uint8_t*
mac,
                                                  size_t mac_size);
        public sgx_status_t put_secret_data(sgx_ra_context_t context,
                                            [in,size=secret_size] uint8_t*
p_secret,
                                            uint32_t secret_size,
                                            [in,count=16] uint8_t* gcm_mac);
        public sgx_status_t enclave_generate_key([in,size=secret_size]uint8_t
*p_data,
                                            uint32_t secret_size);
        public sgx_status_t enclave_encrypt([in,size=secret_size] uint8_t*
p_data,
                                            uint32_t secret_size,
                                            [out,size=secret_size] uint8_t*
out_data);
        public sgx_status_t enclave_decrypt([in,size=secret_size] uint8_t*
p_data,
                                            uint32_t secret_size,
```

```
                                        [out,size=secret_size] uint8_t*
out_data);
    };

};
```

enclave.cpp

```
#include <assert.h>
#include "isv_enclave_t.h"
#include "sgx_tkey_exchange.h"
#include "sgx_tcrypto.h"
#include "string.h"
// This is the public EC key of the SP. The corresponding private EC key is
// used by the SP to sign data used in the remote attestation SIGMA protocol
// to sign channel binding data in MSG2. A successful verification of the
// signature confirms the identity of the SP to the ISV app in remote
// attestation secure channel binding. The public EC key should be hardcoded in
// the enclave or delivered in a trustworthy manner. The use of a spoofed public
// EC key in the remote attestation with secure channel binding session may lead
// to a security compromise. Every different SP the enlcave communicates to
// must have a unique SP public key. Delivery of the SP public key is
// determined by the ISV. The TKE SIGMA protocl expects an Elliptical Curve key
// based on NIST P-256
static const sgx_ec256_public_t g_sp_pub_key = {
    {
        0x72, 0x12, 0x8a, 0x7a, 0x17, 0x52, 0x6e, 0xbf,
        0x85, 0xd0, 0x3a, 0x62, 0x37, 0x30, 0xae, 0xad,
        0x3e, 0x3d, 0xaa, 0xee, 0x9c, 0x60, 0x73, 0x1d,
        0xb0, 0x5b, 0xe8, 0x62, 0x1c, 0x4b, 0xeb, 0x38
    },
    {
        0xd4, 0x81, 0x40, 0xd9, 0x50, 0xe2, 0x57, 0x7b,
        0x26, 0xee, 0xb7, 0x41, 0xe7, 0xc6, 0x14, 0xe2,
        0x24, 0xb7, 0xbd, 0xc9, 0x03, 0xf2, 0x9a, 0x28,
        0xa8, 0x3c, 0xc8, 0x10, 0x11, 0x14, 0x5e, 0x06
    }

};

// Used to store the secret passed by the SP in the sample code. The
// size is forced to be 8 bytes. Expected value is
// 0x01,0x02,0x03,0x04,0x0x5,0x0x6,0x0x7
uint8_t g_secret[8] = {0};
sgx_ec_key_128bit_t sk_key;
//lhadd
sgx_ec_key_128bit_t aes_key;
sgx_ec_key_128bit_t aes2_key;

#ifdef SUPPLIED_KEY_DERIVATION

#pragma message ("Supplied key derivation function is used.")

typedef struct _hash_buffer_t
```

```
{
    uint8_t counter[4];
    sgx_ec256_dh_shared_t shared_secret;
    uint8_t algorithm_id[4];
} hash_buffer_t;

const char ID_U[] = "SGXRAENCLAVE";
const char ID_V[] = "SGXRASERVER";

// Derive two keys from shared key and key id.
bool derive_key(
    const sgx_ec256_dh_shared_t *p_shared_key,
    uint8_t key_id,
    sgx_ec_key_128bit_t *first_derived_key,
    sgx_ec_key_128bit_t *second_derived_key)
{
    sgx_status_t sgx_ret = SGX_SUCCESS;
    hash_buffer_t hash_buffer;
    sgx_sha_state_handle_t sha_context;
    sgx_sha256_hash_t key_material;

    memset(&hash_buffer, 0, sizeof(hash_buffer_t));
    /* counter in big endian  */
    hash_buffer.counter[3] = key_id;

    /*convert from little endian to big endian */
    for (size_t i = 0; i < sizeof(sgx_ec256_dh_shared_t); i++)
    {
        hash_buffer.shared_secret.s[i] = p_shared_key->s[sizeof(p_shared_key-
>s)-1 - i];
    }

    sgx_ret = sgx_sha256_init(&sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&hash_buffer, sizeof(hash_buffer_t),
sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_U, sizeof(ID_U), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_V, sizeof(ID_V), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_get_hash(sha_context, &key_material);
    if (sgx_ret != SGX_SUCCESS)
```

```
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_close(sha_context);

    assert(sizeof(sgx_ec_key_128bit_t)* 2 == sizeof(sgx_sha256_hash_t));
    memcpy(first_derived_key, &key_material, sizeof(sgx_ec_key_128bit_t));
    memcpy(second_derived_key, (uint8_t*)&key_material +
sizeof(sgx_ec_key_128bit_t), sizeof(sgx_ec_key_128bit_t));

    // memset here can be optimized away by compiler, so please use memset_s on
    // windows for production code and similar functions on other OSes.
    memset(&key_material, 0, sizeof(sgx_sha256_hash_t));

    return true;
}

//isv defined key derivation function id
#define ISV_KDF_ID 2

typedef enum _derive_key_type_t
{
    DERIVE_KEY_SMK_SK = 0,
    DERIVE_KEY_MK_VK,
} derive_key_type_t;

sgx_status_t key_derivation(const sgx_ec256_dh_shared_t* shared_key,
    uint16_t kdf_id,
    sgx_ec_key_128bit_t* smk_key,
    sgx_ec_key_128bit_t* sk_key,
    sgx_ec_key_128bit_t* mk_key,
    sgx_ec_key_128bit_t* vk_key)
{
    bool derive_ret = false;

    if (NULL == shared_key)
    {
        return SGX_ERROR_INVALID_PARAMETER;
    }

    if (ISV_KDF_ID != kdf_id)
    {
        //fprintf(stderr, "\nError, key derivation id mismatch in [%s].",
__FUNCTION__);
        return SGX_ERROR_KDF_MISMATCH;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_SMK_SK,
        smk_key, sk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_MK_VK,
        mk_key, vk_key);
```

```
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }
    return SGX_SUCCESS;
}
#else
#pragma message ("Default key derivation function is used.")
#endif

// This ecall is a wrapper of sgx_ra_init to create the trusted
// KE exchange key context needed for the remote attestation
// SIGMA API's. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
// @param b_pse Indicates whether the ISV app is using the
//              platform services.
// @param p_context Pointer to the location where the returned
//                  key context is to be copied.
//
// @return Any error return from the create PSE session if b_pse
//         is true.
// @return Any error returned from the trusted key exchange API
//         for creating a key context.

sgx_status_t enclave_init_ra(
        int b_pse,
        sgx_ra_context_t *p_context) {
    // isv enclave call to trusted key exchange library.
    sgx_status_t ret;
#ifdef SUPPLIED_KEY_DERIVATION
    ret = sgx_ra_init_ex(&g_sp_pub_key, b_pse, key_derivation, p_context);
#else
    ret = sgx_ra_init(&g_sp_pub_key, b_pse, p_context);
#endif
    return ret;
}


// Closes the tKE key context used during the SIGMA key
// exchange.
//
// @param context The trusted KE library key context.
//
// @return Return value from the key context close API

sgx_status_t SGXAPI enclave_ra_close(
        sgx_ra_context_t context) {
sgx_status_t ret;
ret = sgx_ra_close(context);
return ret;
}


// Verify the mac sent in att_result_msg from the SP using the
// MK key. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
```

```
//
//
// @param context The trusted KE library key context.
// @param p_message Pointer to the message used to produce MAC
// @param message_size Size in bytes of the message.
// @param p_mac Pointer to the MAC to compare to.
// @param mac_size Size in bytes of the MAC
//
// @return SGX_ERROR_INVALID_PARAMETER - MAC size is incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESCMAC function.
// @return SGX_ERROR_MAC_MISMATCH - MAC compare fails.

sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                   uint8_t* p_message,
                                   size_t message_size,
                                   uint8_t* p_mac,
                                   size_t mac_size)
{
    sgx_status_t ret;
    sgx_ec_key_128bit_t mk_key;

    if(mac_size != sizeof(sgx_mac_t))
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }
    if(message_size > UINT32_MAX)
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }

    do {
        uint8_t mac[SGX_CMAC_MAC_SIZE] = {0};

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_MK, &mk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        ret = sgx_rijndael128_cmac_msg(&mk_key,
                                       p_message,
                                       (uint32_t)message_size,
                                       &mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        if(0 == consttime_memequal(p_mac, mac, sizeof(mac)))
        {
            ret = SGX_ERROR_MAC_MISMATCH;
            break;
        }

    }
    while(0);
```

```
    return ret;
}


// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t put_secret_data(
    sgx_ra_context_t context,
    uint8_t *p_secret,
    uint32_t secret_size,
    uint8_t *p_gcm_mac)
{
    sgx_status_t ret = SGX_SUCCESS;

    do {
        if(secret_size != 8)
        {
            ret = SGX_ERROR_INVALID_PARAMETER;
            break;
        }

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_SK, &sk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }

        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_secret,
                                         secret_size,
                                         &g_secret[0],
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
                                         0,
                                         (const sgx_aes_gcm_128bit_tag_t *)
                                            (p_gcm_mac));

        uint32_t i;
        bool secret_match = true;
        for(i=0;i<secret_size;i++)
        {
```

```
            if(g_secret[i] != i)
            {
                secret_match = false;
            }
        }

        if(!secret_match)
        {
            ret = SGX_ERROR_UNEXPECTED;
        }

        // Once the server has the shared secret, it should be sealed to
        // persistent storage for future use. This will prevents having to
        // perform remote attestation until the secret goes stale. Once the
        // enclave is created again, the secret can be unsealed.
    } while(0);
    return ret;
}

// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t enclave_generate_key(
    uint8_t *p_data,
    uint32_t secret_size)
{
    sgx_status_t ret = SGX_SUCCESS;

    if(secret_size != 32)
    return SGX_ERROR_INVALID_METADATA;

    uint8_t aes_gcm_iv[12] = {0};
    uint8_t out_data[32] = {0};
    int i = 0;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    do {
        //首先验证16个字节是不是token
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_data,
                                         32,
                                         &out_data[0],
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
```

```
                                        0,
                                        &c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
    } while(0);
    //token这个硬编码成5到20
    bool secret_match = true;
    for(i=0;i<16;i++)
    {
        if(out_data[i] != i+5)
        {
            secret_match = false;
        }
    }
    if(secret_match == true)
    {//设定后16字节为key
        memcpy((void* )&aes_key, &out_data[16], 16);
        memcpy((void* )&aes2_key, &out_data[16], 16);
    }

    return ret;
}

sgx_status_t enclave_encrypt(
    uint8_t *p_data,
    uint32_t secret_size,
    uint8_t *out_data)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    do {
        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_encrypt(&aes_key,
                                        p_data,
                                        secret_size,
                                        out_data,
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
                                        &c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
    } while(0);
    return ret;
}

sgx_status_t enclave_decrypt(
    uint8_t *p_data,
    uint32_t secret_size,
    uint8_t *out_data)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
```

```
    memcpy(out_data, &aes2_key,16);
    do {
        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&aes2_key,
                                         p_data,
                                         secret_size,
                                         out_data,
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
                                         0,
                                         (const sgx_aes_gcm_128bit_tag_t
*)&c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {

            break;
        }
    } while(0);
    return ret;
}
```

Service Provider

network_ra_server.cpp

提供socket通信服务

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
//add
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#include "network_ra.h"
#include "service_provider.h"


// Used to send requests to the service provider sample.  It
// simulates network communication between the ISV app and the
// ISV service provider.  This would be modified in a real
// product to use the proper IP communication.
//
// @param server_url String name of the server URL
// @param p_req Pointer to the message to be sent.
// @param p_resp Pointer to a pointer of the response message.

// @return int
char sendbuf[BUFSIZ];  //数据传送的缓冲区
char recvbuf[BUFSIZ];  //数据接受的缓冲区
```

```
int server_sockfd;//服务器端套接字
int client_sockfd;//客户端套接字

int ra_network_send_receive(const char *server_url,
    const ra_samp_request_header_t *p_req,
    ra_samp_response_header_t **p_resp)
{
    int ret = 0;
    ra_samp_response_header_t* p_resp_msg;

    if((NULL == server_url) ||
        (NULL == p_req) ||
        (NULL == p_resp))
    {
        return -1;
    }

    switch(p_req->type)
    {

    case TYPE_RA_MSG0:
        ret = sp_ra_proc_msg0_req((const sample_ra_msg0_t*)((uint8_t*)p_req
            + sizeof(ra_samp_request_header_t)),
            p_req->size);
        if (0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail [%s].",
                __FUNCTION__);
        }
        break;

    case TYPE_RA_MSG1:
        ret = sp_ra_proc_msg1_req((const sample_ra_msg1_t*)((uint8_t*)p_req
            + sizeof(ra_samp_request_header_t)),
            p_req->size,
            &p_resp_msg);
        if(0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail [%s].",
                __FUNCTION__);
        }
        else
        {
            *p_resp = p_resp_msg;
        }
        break;

    case TYPE_RA_MSG3:
        ret =sp_ra_proc_msg3_req((const sample_ra_msg3_t*)((uint8_t*)p_req +
            sizeof(ra_samp_request_header_t)),
            p_req->size,
            &p_resp_msg);
        if(0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg3_req fail [%s].",
                __FUNCTION__);
        }
        else
```

```
        {
            *p_resp = p_resp_msg;
        }
        break;

    default:
        ret = -1;
        fprintf(stderr, "\nError, unknown ra message type. Type = %d [%s].",
            p_req->type, __FUNCTION__);
        break;
    }

    return ret;
}

int server(int port)
{
    FILE *OUTPUT = stdout;
    int len;
    struct sockaddr_in my_addr;   //服务器网络地址结构体
    struct sockaddr_in remote_addr; //客户端网络地址结构体
    socklen_t sin_size;

    memset(&my_addr,0,sizeof(my_addr)); //数据初始化--清零
    my_addr.sin_family=AF_INET; //设置为IP通信
    my_addr.sin_addr.s_addr=INADDR_ANY;//服务器IP地址--允许连接到所有本地地址上
    my_addr.sin_port=htons(port); //服务器端口号

    /*创建服务器端套接字--IPv4协议，面向连接通信，TCP协议*/
    if((server_sockfd=socket(PF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket");
        return 1;
    }

        /*将套接字绑定到服务器的网络地址上*/
    if (bind(server_sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr))
<0)
    {
        perror("bind");
        return 1;
    }

    /*监听连接请求--监听队列长度为5*/
    listen(server_sockfd,5);

    sin_size=sizeof(struct sockaddr_in);

    /*等待客户端连接请求到达*/
    if((client_sockfd=accept(server_sockfd,(struct sockaddr
*)&remote_addr,&sin_size))<0)
    {
        perror("accept");
        return 1;
    }
    fprintf(OUTPUT, "\naccepted\n");
    return 0;
}
```

```
int SendToClient(int len)
{
    len=send(client_sockfd,sendbuf,len,0);//发送欢迎信息
}

int RecvfromCient()
{
    /*接收客户端的数据*/
    int len = 0;
    memset(recvbuf, 0, BUFSIZ);
    len=recv(client_sockfd,recvbuf,BUFSIZ,0);
    if (len > 0 || len < BUFSIZ)
        recvbuf[len] = 0;
    return len;
}

int Cleanupsocket()
{
    close(client_sockfd);
    close(server_sockfd);
    return 0;
}
// Used to free the response messages.  In the sample code, the
// response messages are allocated by the SP code.
//
//
// @param resp Pointer to the response buffer to be freed.

void ra_free_network_response_buffer(ra_samp_response_header_t *resp)
{
    if(resp!=NULL)
    {
        free(resp);
    }
}
```

###