System Security
Writing an Intel SGX Enclave Application

Graded

Distribution: 25.11.2021
Submission: 09.12.2021, 14:15

After months of research, Alice has finally made a huge discovery. She found a way to to add two integers[1]! To her dismay, when she excitedly tells her friend Bob about this discovery, he replies: "Pfft, I've known a method for adding two integers together for years now!" Alice demands proof, but Bob does not want to give up his secret method, nor does he want to disclose the addition results to Alice[2]. Therefore, they agree on the following approach:

Each of them will implement an enclave, enclave A and enclave B, respectively. Enclave A will create challenges, i.e. a pair of Integers, and send them to enclave B. Enclave B will add the Integers and send the result back to enclave A. Enclave A verifies and outputs whether the result is correct. To ensure that the result is not correct only by chance, multiple rounds of challenge-response are performed between the enclaves. To ensure at least some form of authentication, the enclaves will exchange pre-shared secrets before they start the challenge-response protocol[3].

In this exercise, you will implement enclaves A and B as well as their accompanying untrusted apps A and B.

1 Environment Setup

We will run the Intel SGX enclaves in simulation mode (`SIM`), so you do not need SGX-enabled hardware. To compile, execute and test SGX applications, you will need to install the SGX SDK provided by Intel [1].

To install the SDK on the VM provided by us, one needs to download the installation binary provided by Intel, install it in the directory of your choice, and update the terminal's environment variables so that it can find the SDK. We provide the script `install.sh` which takes care of these steps. For this exercise, we use the SDK version 2.15.

[1] For a realistic scenario, one could replace adding two integers with, for example, reversing a cryptographic hash function.

[2] If this were about reversing a hash function, Alice could claim to have found her own method so that Bob computes pre-images for her.

[3] In a realistic deployment, this authentication mechanism is clearly not sufficient, as it is vulnerable to a number of attacks. Instead, one would typically rely on attestation. However, this is out of scope for this exercise.

2 Developer Reference

It is part of this exercise that you familiarize yourself with Intel SGX and the steps needed to write, compile, and run enclave applications. The Developer Reference [2] is an excellent starting point, both for understanding the general development flow and as a documentation of library functions and types.

Note that you do not need to read the whole development reference for this exercise. Here are some pointers to important sections:

- *Enclave Development Basics*, p. 29
- *Enclave Definition Language Syntax*, p.41. Important: you need to add external libraries in the edl file, e.g. `include "sgx_tcrypto.h"`. Pointer handling can be found on page 45
- *Untrusted Library Functions*, such as enclave creation and destruction, p.113
- *Trusted libraries*, p. 115, in particular:
  - Helper functions, p. 116
  - Cryptography Library, p.128

3 Example Code and Project Structure

The folder `sgxsdk` that you have installed contains sample code and all the libraries that you need to write an enclave application. Inside it, you find a directory called `SampleCode` that contains multiple sample projects to give you plenty of clues on how to implement an SGX enclave application. If you want, you can find more information on these samples in [2] starting from page 89. Let us look into the sample project `SealUnseal` that demonstrates sealing and unsealing of data between two enclaves. Its structure depicted in Figure 1.

1. The App folder contains the source code for the application that creates the enclave. Note that the App is an untrusted entity. All the IO operations are done in the App.
2. Enclave_seal contains all code for the enclave that seals data and hands it over to the App. This also contains an enclave definition language (EDL) file that contains the OCALL and ECALL specification.
3. Enclave_Unseal contains all code for the enclave that gets the sealed data from the App (via filesystem operations) and unseals the data.
4. Compiled binary and libraries (only available after running `make`, see below). “app” is the App executable.
5. Makefile contains the compilation instructions.
6. The sealed data (only available after running the app).

To compile the project, execute `make SGX_MODE=SIM` inside the SealUnseal directory that will create the executable `app`.

To execute the compiled enclave application, run: `./app`.

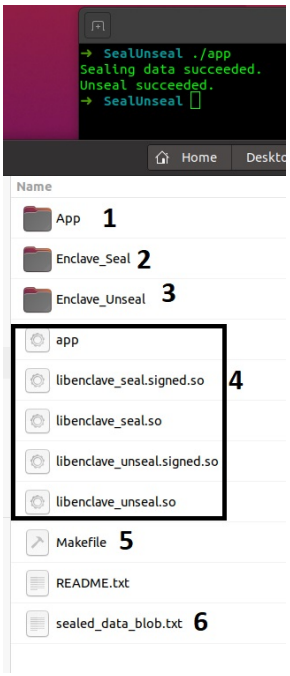

Figure 1: Structure of the SealUnseal sample project.

4 Write your Intel SGX Enclaves

For this exercise, write two enclaves (E_A and E_B) along with two untrusted apps A_A and A_B where E_A and E_B are managed by A_A and A_B respectively.

Figure 2 depicts the protocol between these two pairs of enclaves and applications. Implement this protocol. The apps A_A and A_B act as the untrusted transport between the two enclaves E_A and E_B. A_A communicates the the outcome of the challenges to the user. The communication between A_A and A_B could, for example, be implemented using a filesystem, sockets, or shared memory. Note that some parts of the protocol are underspecified by purpose. We expect you to make reasonable design decisions for these parts.

Tip 1: The key exchange (`sgx_ecc256_compute_shared_dhkey`) produces a 256 bit shared secret. However, the AES-CTR in SGX (`sgx_aes_ctr_encrypt` and `sgx_aes_ctr_decrypt`) only supports 128bit keys. Use the first 128 bits from the 256 bit shared key.

Tip 2: Any changes just in the EDL files are not picked up by the incremental compilation. In such a case, do a `make clean` before `make SGX_MODE=SIM`. Make sure to include external libraries in case you use some of their data types in the OCALL or ECALL arguments.

Tip 3: `sgx_aes_ctr_encrypt` requires an initialization vector (`*p_ctr`) and the parameter (`ctr_inc_bits`) be set to the number of bits to increment the counter. You can initialize the IV as a zero byte array at both sides (E_1 and E_2). However, if you want a more secure communication, you should initialize the IV to a random byte array (`sgx_read_rand`) and send it across with every ciphertext. However, our grading scheme does not consider if you use this or not.

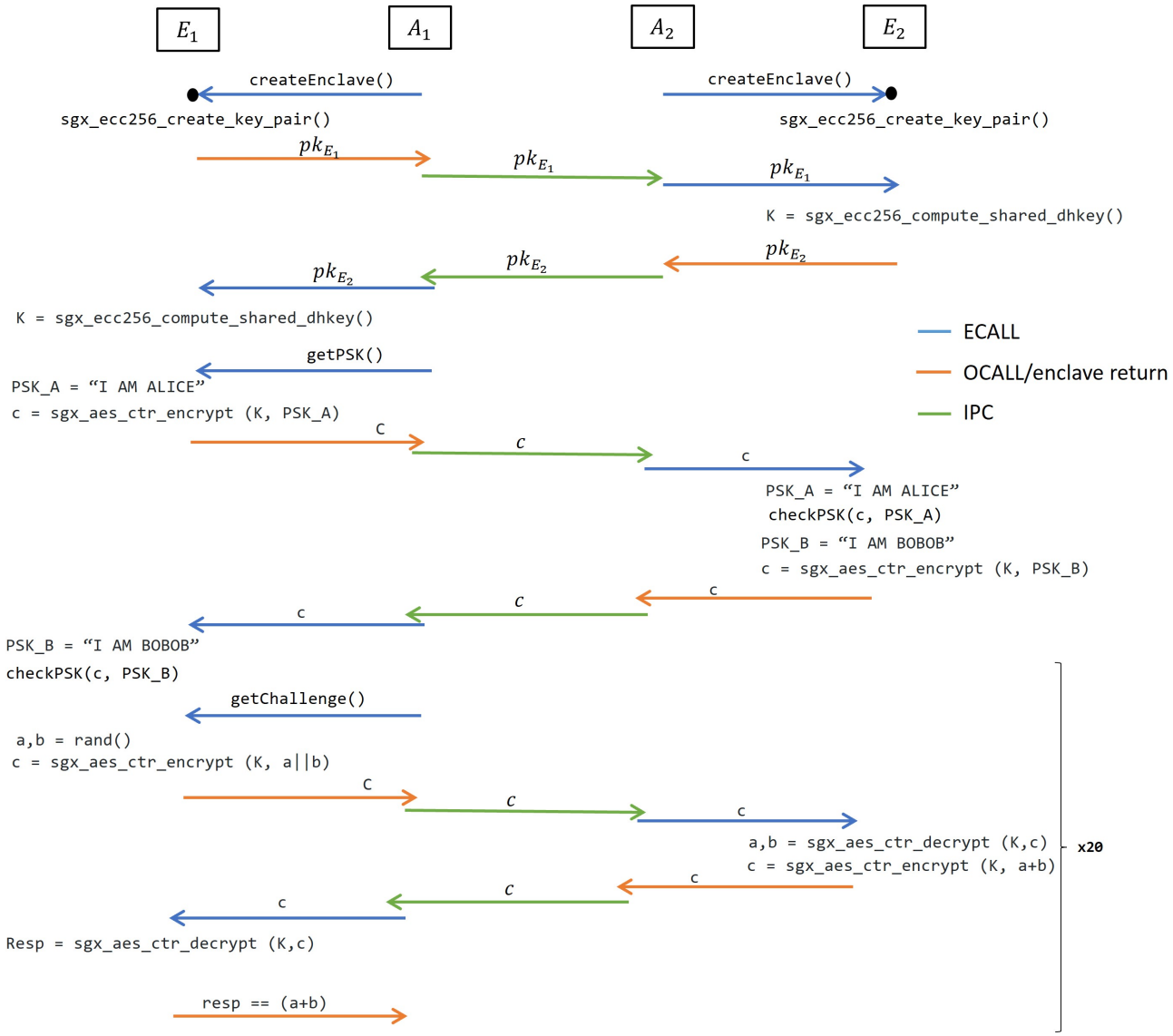

Figure 2: Protocol

Tip 4: `sgx_ecc256_compute_shared_dhkey` produces the DH shared secret that is passed as one of the arguments to the function. The shared secret data type is `sgx_ec256_dh_shared_t`. It is a structure unlike the other data types used in the SGX crypto library (typically byte arrays). Access the shared secret by accessing the member `s` of the struct. For more information, look at [3] to understand the data types of the SGX crypto API.

Tip 5: Some of the code files that you will see in the directory, such as Enclave_u.c or Enclave_u.h, are auto-generated. Do not modify these files. You can run `make clean` to remove all files generated by `make`.

4.1 Project template

We have prepared a template for you to facilitate starting with the enclave programming. The template consists of two identical enclaves (E_A and E_B) along with their untrusted applications (A_A and A_B). These are placed in directories `Enclave_A` and `Enclave_B`, respectively. Note that you do not need to modify the make files unless you are using additional external libraries or add more files, which is not a requirement to successfully complete the assignment.

4.2 How to annotate the code

Please mark your code at the following points:

1. Points of communication between A_A and A_B: Sending and receiving the public keys, the encrypted PSK, and the encrypted challenges/responses.
2. The points where the enclaves E_A and E_B generate their key pairs.
3. The points where the enclaves E_A and E_B calculate the shared secret.
4. The point where enclave E_A generates and encrypts the challenge.
5. The point where enclave E_A decrypts and verifies the response.
6. The point where enclave E_B decrypts the challenge.
7. The point where enclave E_B computes and encrypts the response.

For this, use the following format:

```
/*************************
 * BEGIN [part that you're marking, e.g. 6. E_B decrypt challenge]
 *************************/
<your code here>
/*************************
 * END [part that you're marking, e.g. 6. E_B decrypt challenge]
 *************************/
```

Furthermore, if parts your code are inspired by external sources, please annotate accordingly, e.g. "`// Based on https://stackoverflow.com/how_to_initialize_SGX_enclave`"

4.3 Submission process

Please submit the zipped directory of the project in Moodle. In case you use external libraries (which is not necessary to successfully complete this project), make sure that your `Makefile` is self-sufficient to compile your project. We cannot grade you if the compilation fails. Add a short description in the included `README.txt` file that provides clear instructions on how to compile and run your code.

References

[1] Intel. *sgx-linux version 2.15.* https://download.01.org/intel-sgx/sgx-linux/2.15/. https://download.01.org/intel-sgx/sgx-linux/2.15/

[2] Intel. *Intel Software Guard Extensions SDK for Linux OS Developer Reference.* https://download.01.org/intel-sgx/sgx-linux/2.15/docs/Intel_SGX_Developer_Reference_Linux_2.15_Open_Source.pdf.

[3] Intel. *linux-sgx/sgx_tcrypto.h at master . intel/linux-sgx . GitHub.* https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_tcrypto.h.