MCS: Secure Execution Environments 2022-23

Practical Exercises: SGX code examples

March 24, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

1 Introduction

The goal of this laboratory guide is to modify or extend some simple SGX enclave code. These exercises must be compiled and executed on a GNU/Linux system running on a Intel processor with the Security Guard Extensions (SGX) enabled and its corresponding software development toolkit installed (do it in `sgx.ieeta.pt`).

2 The first exercise

The code of the first example is already complete. We will only

- change the private key used to sign the enclave, and
- change the signed copy of the enclave code, to confirm that any changes to that code results in a failure to launch the enclave.

Work to be done by the student:

1. Get the HelloEnclaveV1 code from the course web site (it's in the `projects/HelloEnclaveV1` directory inside the `sgx_examples.tgz` archive).
2. Study the code. The code is subdivided into application code (in the `App` directory), and the enclave code (in the `Enclave1` directory).
3. Generate a new private key. Just run

   ```
   bash new_private_key.bash
   ```

   Study the `bash` script to see how this is done. Be aware that the RSA modulus must have 3072 bits and that the public exponent must be 3 (these limitations are imposed by the SGX SDK).
4. Compile the entire project:

   ```
   make
   ```

5. Run the program:

   ```
   ./app
   ```

6. Edit the signed enclave code using the command

   ```
   vi -b enclave1.signed.so
   ```

   As an alternative to this command, you may also use any hexadecimal editor (for example, `ghex`). Replace "Hello" with "Hallo". Keep in mind that we are editing a shared library, so we cannot change the size of the file, as that would mess with the file format of this kind of files.

7. Run the program again and confirm that the enclave was not launched.

8. Clean up after finishing this exercise:

   ```
   make clean
   ```

3 The second exercise

In the second exercise we will add a new enclave function that can be called from outside of the enclave (an **ecall**), and we will call it in our new application.

Work to be done by the student:

1. Make a copy of the original HelloEnclaveV1 code:

   ```
   cp -ar HelloEnclaveV1 HelloEnclaveV2
   ```

   (The `sgx_examples.tgz` archive contains the final code in the `projects/HelloEnclaveV2_done` directory, so do not use it at first. That is cheating!)

2. Add the function

   ```
   void e1_sum_array(int *ptr,size_t n,int *sum)
   {
     size_t i;
     int s = 0;
     for(i = 0;i < n;i++)
       s += ptr[i];
     *sum = s;
   }
   ```

   to the `Enclave1/Enclave1.cpp` file. This function has two input arguments (the pointer `ptr` to the beginning of an array of integers and the integer argument `n` that specifies the number of elements of the array) and an output argument (the pointer `sum` to an integer). Note that the output argument is passed back via a pointer. That is so because the ecall returns a status value, and so our ecalls must be declared as void functions.

3. Add the prototype of the new function to the `Enclave1/Enclave1.h` file.

4. Add the specification of the new ecall to the trusted section of the `Enclave1/Enclave1.edl` file. To do this properly you need to read the "Enclave Definition Language Syntax" section of the "Intel Software Guard Extensions SDK for Linux OS" manual. In particular, you need to add the following text to that file:

   ```
   public void e1_sum_array([in, count=n] int *ptr,size_t n,[out] int *sum);
   ```

   Note that the `ptr` pointer was classified as being an "in" argument having `n` elements, and so the `edger8r` tool will make sure that whenever this ecall is called the memory used by the integer array of `n` elements pointed to by `ptr` is copied to a private copy inside the enclave. **Be aware**

that it will not be copied back! If you need that, you will also need to put "out" inside the square brackets. Note also that for pointer variables it is the memory pointed to that is passed and not the pointer itself; when the count value is omitted it is assumed to be equal to 1.

5. Add the following code to the main function in the `App/App.cpp` file:

```
  int n = 10;
  int sum,a[n];
  for(int i = 0;i < n;i++)
    a[i] = i;
  if((ret = e1_sum_array(global_eid1,&a[0],n,&sum)) != SGX_SUCCESS)
  {
    print_error_message(ret);
    return -1;
  }
  printf("n = %d\nsum = %d\n",n,sum);
```

6. Compile and run the program.

7. Study the contents of the four files produced by the `edger8r` tool: `App/Enclave1_u.[ch]` and `Enclave1/Enclave1_t.[ch]`.

8. Add code to measure the time it takes to compute the sum of the elements of the array, doing so either inside or outside of the enclave. Try different values of `n`. You should do several measurements in each case, and you should perform the time measurements by reading the time stamp counter:

```
static unsigned long rdtsc(void)
{
  unsigned long rax,rcx,rdx;

  asm volatile
  (
    "rdtscp"
    : "=a" (rax),"=c" (rcx),"=d" (rdx)
  );
  return (rdx << 32) + rax;
}
```

The output of your program should be like this (first for `n`=1000 and, on a different run, for `n`=10000):

```
Hello, world (from enclave 1)
App: n=1000 sum=499500 t=[1379,708,699,694,687,693,694,689,25623,716]
Enclave1: n=1000 sum=499500
    t=[60748,16860,37494,16616,15971,16265,15971,15998,15963,15949]

Hello, world (from enclave 1)
App: n=10000 sum=49995000 t=[9003,6867,6828,6845,6832,6836,6790,32893,7006,7025]
Enclave1: n=10000 sum=49995000
    t=[110158,28447,27911,27728,28542,27814,27686,27942,54108,45487]
```

The execution time variability is due to the activity of other programs and to external events (interrupts). For large values of `n` running the code inside the enclave is roughly twice as slow as running it outside of the enclave (ito run it inside of the enclave it is necessary to make a memory copy).

4 The third exercise

In the third exercise we will use two enclaves in the same program and we will use the Diffie-Hellman key exchange protocol to establish a 128-bit secret key that can be used by a symmetric cipher (not done) to exchange messages in a secure way between them.

The teachers cloned the enclave code of the first example and they stored the cloned copy in the `Enclave2` directory. In the cloned enclave all symbols containing the number 1 – that refer to the first enclave – were modified to contain the number 2 – that refer to the second enclave. The `Makefile` was also modified so that the code of the two enclaves is compiled and used by the application.

You can find the initial code for the third exercise in the `HelloEnclaveV3` directory; the final code (no peaking, please) can be found in the `HelloEnclaveV3_done` directory.

Initial work to be done by the student:

1. Study the code. In particular, compare the `App.cpp` code of the first exercise with the code for this one (the `meld` program may be useful for this). Also, compare the code of the two enclaves of this exercise (apart for 1's having been replaced by 2's, they are identical).
2. Compile and run the program.
3. In the file `App.cpp` replace the line

   ```
   if((ret = e1_printf_hello_world(global_eid1)) != SGX_SUCCESS)
   ```

   by the line

   ```
   if((ret = e1_printf_hello_world(global_eid2)) != SGX_SUCCESS)
   ```

   (That is, call the ecall function of the first enclave using the identifier of the second.)

   Compile and run the modified program. Were you expecting an error message[1]? After this experiment, undo the change.

At this point you have tested the program that was provided by the teachers.

It is now time for you to modify it to establish the 128-bit secret key (stored separately in each enclave). Fortunately for you, the SGX SDK already provides code to perform the Diffie-Hellman key exchange. Study the "Diffie-Hellman Key Exchange Library and Local Attestation Flow" of the "Intel Software Guard Extensions SDK for Linux OS" manual. Also, take a good look at the `/opt/intel/sgxsdk/include/sgx_dh.h` include file and study, compile and run the `/opt/intel/sgxsdk/SampleCode/LocalAttestation` code sample.

The required flow of information to establish the 128-bit secret key is the following (this is also stored, with some more details, in the `dh_work_flow.txt` file.)

Step 1 performed in enclave 1, `e1_init_session` – this has to create the initiator session;

Step 2 performed in enclave 2, `e2_init_session` – this has to create the responder session;

Step 3 performed in enclave 2, `e2_create_message1` – this has to create the first message;

Step 4 send the first message from enclave 2 to enclave 1;

[1]The "glue" code produced by the `edger8r` tool does not care about function names. All that matters is the internal function number of the ecall (check the documentation of the `sgx_ecall` function). As long as the function number and argument list are correct, the glue code will not crash when given the wrong enclave identifier. As the ecall functions of the two enclaves are similar, in this case it calls successfully the ecall of the other enclave! In general, all hell may break loose when this is done.

Step 5 performed in enclave 1, `e1_process_message1` – this has to process the first message (the end result is the second message);

Step 6 send the second message from enclave 1 to enclave 2;

Step 7 performed in enclave 2, `e2_process_message2` – this has to process the second message (the end result is the third message, the secret key, and the initiator identity);

Step 8 send the third message from enclave 2 to enclave 1;

Step 9 performed in enclave 1, `e1_process_message3` – this has to process the third message (the end result is the secret key, and the responder identity).

You only need to create ecalls to do the necessary work. At the end, print the secret key stored in each enclave (to verify that they are equal). The communication channel between the two enclaves is the (untrusted) application ifself, so steps 4, 6 and 8 do nothing if the ecalls return to the application the messages that they generate. In a more realistic scenario, when the two enclaves are running on different machines, it would be necessary to provide the means to transfer the messages between them (using TCP or UDP sockets). One possible output of the program is the following:

```
Hello, world (from enclave 1)
Hello, world (from enclave 2)
Enclave 1 AEK: 54 B0 16 56 07 E0 8D 7F 5B 2F 28 CD 7E E2 CA 93
Enclave 2 AEK: 54 B0 16 56 07 E0 8D 7F 5B 2F 28 CD 7E E2 CA 93
```