Lab 02 - Hardware Security

Objectives

- Hardware Security Basics
- Intel SGX API

Intel SGX

Description

SGX is an Intel technology aiming secure code execution inside enclaves, available starting with Intel Skylake CPUs. An enclave is a protected region of memory that can only be accessed by the owning application (not even the kernel or other peripherals - e.g. DMA - is allowed to read it). It can provide remote attestation to services that require code to be executed on untrusted devices.

SGX applications are divided into two logical components: trusted and untrusted. Intel advises that the trusted component (the enclave) should be as small as possible (small = lesser chance of bugs), enforced by a hardware limit of 128 MB for the entire protected memory space.

The enclave code runs in user space, and thus can access the entire memory of the process within which it runs. In this way, data can be transmitted between the trusted and untrusted application regions, e.g. for passing function parameters and results. An aspect worth mentioning is that, inside the enclave, one can not directly execute priviledeged operations (such as system calls). For this, one needs to execute an untrusted function.

For the interaction between those two components, the SDK exposes two important concepts:

- ECALL (Enclave Call) - entering the enclave and calling a function inside the enclave.
- OCALL (Outside Call) - calling an untrusted function from the enclave mode. It implies leaving the protected mode, executing the untrusted function and then reentering the enclave.

These functions represent the way the untrusted application part and the enclave interact. They are defined in an EDL file (Enclave Definition Language), in a specific format. A simple EDL file looks like this:

```
enclave {
    trusted {
        /* define ECALLs here. */
        public int get_sum(int a, int b);
    };

    untrusted {
        /* define OCALLs here. */
        void ocall_print([in, string]const char* str);
    };
};
```

ECALL and OCALL functions parameters that are pointers should be decorated with either a pointer direction attribute in, out or a user_check attribute explicitly.

- [in] – the parameter is passed from the calling procedure to the called procedure. For an ECALL the in parameter is passed from the application to the enclave, for an OCALL the parameter is passed from the enclave to the application.
- [out] – the parameter is returned from the called procedure to the calling procedure. In an ECALL function an out parameter is passed from the enclave to the application and an OCALL function passes it from the application to the enclave.
- [user_check] - Pointer is not checked. Users must perform the check and/or copy.

Observation: Using direction attributes ([in], [out]) implies copying buffers from the application to enclave or the other way around, so usually these are followed by a size attributes that indicates how much data needs to be copied, e.g.:

```
void foo([in, size=buf_len]const char* buf, size_t buf_len);
```

Preparation

Instance Creation

Ideally, you should solve this laboratory on openstack [https://cloud-controller.grid.pub.ro]. As last week, create an **m1.small** instance from the **ISC 2020** image. If you are outside the campus local network, you will need to set up a 2-hop SSH connection via fep. If you have configured your **localhost**'s keypair on openstack (not the one you generated on fep), you can connect directly this way:

```
$ ssh -J ${LDAP_ID}@fep.grid.pub.ro student@${VM_IP}
```

NOTE: you can have more than one keypair configured (both for your localhost, and for fep). Select the one you want from the *Key Pair* tab during the instance creation.

There's currently a problem with the VM. See **Working Locally** below.

Recording

Please record your screen using asciinema [https://www.mankier.com/1/asciinema] while working.

```
# install asciinema
$ sudo apt update -y && sudo apt install -y asciinema

# start recording
# NOTE: use --append if you pause your work and resume it later
#       it helps us if you upload a single .cast
$ asciinema rec [--append] lab02_${LDAP_ID}.cast

# IMPORTANT: before you start, echo your name in the terminal
$ echo "Andrei Popescu"

# work...

# stop recording
$ exit

# upload recording
$ ASCIINEMA_API_URL=https://asciinema.cs.pub.ro asciinema upload lab02_${LDAP_ID}.cast
```

When you finish your work, submit the details on the form [https://forms.office.com/r/GZzRJVqQuy]. Double check the form responses [https://ctipub-my.sharepoint.com/:x:/g/personal/mihai_chiroiu_upb_ro/Ee4pZApRKA5Iq9JgR2r652QB0FVL4J9EFtTBtva3jX-1Lw?e=ylQ7lu] so there aren't any surprises.

SDK and Code Skeleton

You can find the skeleton here: isc-enclave-2020.zip. The provided code archive contains TODOs that will guide you through solving the tasks. Over the course of the lab, make sure to consult the documentation [https://download.01.org/intel-sgx/sgx-linux/2.8/docs/Intel_SGX_Developer_Reference_Linux_2.8_Open_Source.pdf].

Since you'll be working inside a headless machine, you will need some vim skills to effortlessly edit the code; don't worry! there are plenty of docs on the Internet [https://vim.rtorr.com/] ;)

Normally, to benefit from *most* SGX security features, you need two things:

1. **SGX driver & BIOS support**: needed to lock memory and limit access to Enclaves.
2. **SGX SDK (Software Development Kit)**: implements most Enclave interactions, generates required code from EDL files, signs resulting objects, etc.

SGX projects can be compiled in both **hardware** and **simulation** mode. Since we will be working in simulation mode, we won't require the SGX driver, the BIOS support, or even the ENCLU [https://www.felixcloutier.com/x86/enclu] instructions that implement the SGX functionality. The SDK is already installed on your VM. In order to be able to use it, you will need to import/update some environment variables:

```
# do this in every shell you open (maybe add it to .bashrc)
source /opt/intel/sgxsdk/environment
```

Use the following Makefile targets for compiling / cleaning the workspace:

```
$ make SGX_MODE=SIM
$ make clean
```

Working Locally

If you encounter problems creating an *openstack* instance, you may be able to solve the lab locally. You will have to install the SGX SDK from here [https://github.com/intel/linux-sgx]. The code for this laboratory will be compiled in **simulation** mode, so you won't need hardware support. Here are the steps for Ubuntu 18.04. Make the necessary modifications based on the SDK repo's README.md if you are using something else:

The SDK build process for **Ubuntu 16** is broken. This is 100% the fault of the Intel developers working on the SDK. Unless you want to try and update your *glibc* from 2.23 to 2.27, work in a docker container (how to install [https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04]):

```
# use sudo if you are not in the docker group
$ docker run -ti --entrypoint bash ubuntu:20.04
```

```
# IMPORTANT: find out what distro you are using
#            especially if you are working on WSL
#            use this information where appropriate in the build process
$ cat /etc/lsb-release
    DISTRIB_ID=Ubuntu
    DISTRIB_RELEASE=18.04
    DISTRIB_CODENAME=bionic
    DISTRIB_DESCRIPTION="Ubuntu 18.04.5 LTS"

# install deps for Ubuntu 18
$ sudo apt update -y && sudo apt install -y          \
  build-essential ocaml ocamlbuild automake autoconf \
  libtool wget python libssl-dev git cmake perl      \
  libssl-dev libcurl4-openssl-dev protobuf-compiler  \
  libprotobuf-dev debhelper cmake reprepro unzip

# install deps for Ubuntu 20
$ sudo apt update -y && sudo apt install -y              \
  build-essential ocaml ocamlbuild automake autoconf     \
  libtool wget python-is-python3 libssl-dev git cmake    \
  perl libssl-dev libcurl4-openssl-dev protobuf-compiler \
  libprotobuf-dev debhelper cmake reprepro unzip


# clone repo and init submodules
$ git clone https://github.com/intel/linux-sgx.git
$ cd linux-sgx
$ make preparation -j $(nproc)

# make sure to use the provided mitigation binaries or the
# next step will fail; change "ubuntu18.04" to your distro
$ export PATH="$(realpath external/toolset/ubuntu18.04):${PATH}"
$ which as ld ld.gold objdump
    ${HOME}/linux-sgx/external/toolset/ubuntu18.04/as
    ${HOME}/linux-sgx/external/toolset/ubuntu18.04/ld
    ${HOME}/linux-sgx/external/toolset/ubuntu18.04/ld.gold
    ${HOME}/linux-sgx/external/toolset/ubuntu18.04/objdump

# compile SDK with default settings; create installer
$ make sdk_install_pkg -j $(nproc)

# create installation path and install SDK
$ mkdir /opt/intel
$ cd linux/installer/bin
$ ./build-installpkg.sh sdk
$ ./sgx_linux_x64_sdk_2.13.100.4.bin
    Do you want to install in current directory? [yes/no] : no
    Please input the directory which you want to install in : /opt/intel

# import env variables specific to the installed SDK
$ source /opt/intel/sgxsdk/environment

# test that everything is working
$ cd ../../../SampleCode/SampleEnclave
$ make SGX_MODE=SIM
$ ./app
    Checksum(0x0x7fffffffd5f0, 100) = 0xfffd4143
    Info: executing thread synchronization, please wait...
    Info: SampleEnclave successfully returned.
    Enter a character before exit ...
```

If you installed the latest SDK yourselves, use the updated skeleton from here.

If you want to check whether you do have hardware support, there are two easy ways:

1. Check if bit 2 is set in the EBX register as returned by the EAX=7,ECX=0 CPUID [https://en.wikipedia.org/wiki/CPUID#EAX=7,_ECX=0:_Extended_Features] leaf instruction. You can use the tool with the same name: cpuid [https://linux.die.net/man/1/cpuid].
2. Compile and run *test-sgx.c* from this repo [https://github.com/ayeks/SGX-hardware]. Note that this user also maintains a list of hardware that supports Intel SGX (if you may ever need it).

Tasks

[20p] 1. Implement an ECALL function that generates a random number (unsigned int) inside the enclave, between 3 and 42.

The function should have the **prototype**:

```
unsigned int generate_random_number(void)
```

Steps

1. Add the function prototype to **Enclave/Enclave.edl**
2. Implement the function in **Enclave/Enclave.cpp**. The function must make use of the **sgx_read_rand** function in the Intel SGX SDK.

   ```
   sgx_status_t sgx_read_rand(unsigned char *rand, size_t length_in_bytes);
   ```

   Where:

  - rand - pointer to the memory area where the random bytes will be generated
  - length_in_bytes - number of bytes to generate
  - **sgx_trts.h** must be included
3. Call the function in **App/App.cpp**. First parameter of the function must be **global_eid**. The result is returned in the **second parameter**. The following parameters (if any) represent the arguments of the function defined in **Enclave/Enclave.cpp**

- **Hint**: See the already implemented get_sum ECALL.

[20p] 2. System calls can not be called directly from an enclave.

- Implement 2 OCALLS, one for reading from a file, the other for writing to a file.
- The functions should have the prototypes:

```
void ocall_write_file(const char* filename, const char* buf, size_t buf_len);
void ocall_read_file(const char* filename, char* buf, size_t buf_len);
```

- The functions must be implemented in **App/App.cpp** and their prototypes must be added to **Enclave/Enclave.edl**. The parameters which are pointers must be decorated as described in the beginning of the lab (similar to **ocall_print**). Please note that **filename** is a string, but **buf** might contain non-ASCII characters, so **buf_len** must be used to describe its size.
- **Obs**: The methods are also responsible for opening/closing the corresponding file descriptors.
- **Hint**: I/O system calls [https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/] Don't forget to include **unistd.h** and **fcntl.h**

[20p] 3. Implement a function, inside the enclave, that does the following operations:

- Seals the string “SGX_RULLZ” using **sgx_seal_data**.

```
sgx_status_t sgx_seal_data(
    const uint32_t additional_MACtext_length,  ---> no additional MAC, this parameter should be 0
    const uint8_t * p_additional_MACtext,      ---> no additional MAC, this parameter should be NULL
    const uint32_t text2encrypt_length,        ---> length of the text to be encrypted
    const uint8_t * p_text2encrypt,            ---> the text to be encrypted (make sure the type of this parameter is uint8_t* - you can use casts)
    const uint32_t sealed_data_size,           ---> length of the output (encrypted text), must be computed using sgx_calc_sealed_data_size (see below)
    sgx_sealed_data_t * p_sealed_data          ---> output of the function; you must dynamically allocate this prior to this function call
);
```

. Use **sgx_calc_sealed_data_size** to calculate the number of bytes to allocate for the **sgx_sealed_data_t** structure and to use for the **sealed_data_size** parameter.

```
uint32_t sgx_calc_sealed_data_size(
    const uint32_t add_mac_txt_size,           ---> no additional MAC, this parameter should be 0
    const uint32_t txt_encrypt_size            ---> length of the text to be encrypted
);
```

- Writes the sealed data into a file (filename is given by the macro SECRET_ENCLAVE)
- More on SGX sealing [https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing].

- The function prototype should be:

```
void seal_secret(void);
```

- Inspect the file contents. Can you guess the secret?

- **Obs**: Uncomment the function call from main and don't forget about **Enclave/Enclave.edl**

[20p] 4. Similar to the sealing function, implement a function, inside the enclave, that unseals the enclave secret.

- Reads the content of SECRET_ENCLAVE file (it contains sealed data)
- Uses **sgx_get_encrypt_txt_len()** (see documentation [https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf]) to determine the buffer size for the decrypted data and allocate said buffer.
- Unseals the data using **sgx_unseal_data**

```
sgx_status_t sgx_unseal_data(
    const sgx_sealed_data_t * p_sealed_data, ---> encrypted data (read from the file); you must cast the buffer read from the file to sgx_sealed_data_t *
    uint8_t * p_additional_MACtext,          ---> no additional MAC, this parameter should be NULL
    uint32_t * p_additional_MACtext_length,  ---> no additional MAC, this parameter should be 0
    uint8_t * p_decrypted_text,              ---> buffer where the decrypted data will be written; you can allocate it
                                                  statically, just make sure you provide a large enough buffer
    uint32_t * p_decrypted_text_length       ---> this parameter is both input and output;
                                                  as input, it represents the length of p_decrypted_text;
                                                  as output, represents the number of bytes that were decrypted
);
```

- Prints the unsealed string.
- The function prototype should be:

```
void unseal_secret(void);
```

- **Obs**: Uncomment the function call from main and don't forget about **Enclave/Enclave.edl**

[10p] 5. Modify the seal_secret ECALL such that it seals a random generated string, instead of “SGX_RULLZ”.

- Use **sgx_read_rand** for generating the string.

[10p] 6. Feedback

Please take a minute to fill in the feedback form [https://forms.gle/5Lu1mFa63zptk2ox9] for this lab.

isc/labs/02.txt · Last modified: 2021/03/16 14:54 by radu.mantu