CONTENTS

Software Guard Extensions Part 7: Refine the Enclave with Proxy Functions

By John M., published on November 28, 2016

Part 7 of the Intel® Software Guard Extensions (Intel® SGX) tutorial series revisits the enclave interface and adds a small refinement to make it simpler and more efficient. We'll discuss how the proxy functions marshal data between unprotected memory space and the enclave, and we'll also discuss one of the advanced features of the Enclave Definition Language (EDL) syntax.

You can find a list of all of the published tutorials in the article Introducing the Intel® Software Guard Extensions Tutorial Series.

Source code is provided with this installment of the series. With this release we have migrated the application to the 1.7 release of the Intel SGX SDK and also moved our development environment to Microsoft Visual Studio® Professional 2015.

The Proxy Functions

When building an enclave using the Intel SGX SDK you define the interface to the enclave in the EDL. The EDL specifies which functions are ECALLs ("enclave calls," the functions that enter the enclave) and which ones are OCALLs ("outside calls," the calls to untrusted functions from within the enclave).

When the project is built, the Edger8r tool that is included with the Intel SGX SDK parses the EDL file and generates a series of proxy functions. These proxy functions are essentially wrappers around the *real* functions that are prototyped in the EDL. Each ECALL and OCALL gets a pair of proxy functions: a trusted half and an untrusted half. The trusted functions go into *EnclaveProject_t.h* and *EnclaveProject_t.c* and are included in the Autogenerated Files folder of your enclave project. The untrusted proxies go into *EnclaveProject_u.h* and *EnclaveProject_u.c* and are placed in the Autogenerated Files folder of the project that will be interfacing with your enclave.

Your program does not call the ECALL and OCALL functions directly: it calls the proxy functions. When you make an ECALL, you call the untrusted proxy function for the ECALL, which in turn calls the trusted proxy function inside the enclave. That proxy then calls the "real" ECALL and the return value propagates back to the untrusted function. This sequence is shown in Figure 1. When you make an OCALL, the sequence is reversed: you call the trusted proxy function for the OCALL, which calls an untrusted proxy function outside the enclave that, in turn, invokes the "real" OCALL.

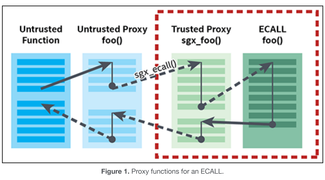

Figure 1. Proxy functions for an ECALL.

The proxy functions are responsible for:

- Marshaling data into and out of the enclave
- Placing the return value of the *real* ECALL or OCALL in an address referenced by a pointer parameter
- Returning the success or failure of the ECALL or OCALL *itself* as an **sgx_status_t** value

Note that this means that each ECALL or OCALL has potentially two return values. There's the success of the ECALL or OCALL *itself*, meaning, were we able to successfully enter or exit the enclave, and then the return value of the function being called in the ECALL or OCALL.

The EDL syntax for the ECALL functions *ve_lock()* and *ve_unlock()* in our Tutorial Password Manager's enclave is shown below:

```
1 enclave {
2   trusted {
3     public void ve_lock ();
4     public int ve_unlock ([in, string] char *password);
5   };
6 }
```

And here are the untrusted proxy function prototypes that are generated by the Edger8r tool:

```
1 sgx_status_t ve_lock(sgx_enclave_id_t eid);
2 sgx_status_t ve_unlock(sgx_enclave_id_t eid, int* retval, char*
  password);
```

Note the additional arguments that have been added to the parameter list for each function and that the functions now return a type of **sgx_status_t**.

Both proxy functions need the enclave identifier, which is passed in the first parameter, eid. The *ve_lock()* function has no parameters and does not return a value so no further changes are necessary. The *ve_unlock()* function, however, does both. The second argument to the proxy function is a pointer to an address that will store the return value from the *real* ve_unlock() function in the enclave, in this case a return value of type **int**. The actual function parameter, **char** **password*, is included after that.

Data Marshaling

The untrusted portion of an application does not have access to enclave memory. It cannot read from or write to these protected memory pages. This presents some difficulties when the function parameters include pointers. OCALLs are especially problematic, because a memory allocated inside the enclave is *not* accessible to the OCALL, but even ECALLs can have issues. Enclave memory is mapped into the application's memory space, so enclave pages can be adjacent to unprotected memory pages. If you pass a pointer to untrusted memory into an enclave, and then fail to do appropriate bounds checking in your enclave, you may inadvertently cross the enclave boundary when reading or writing to that memory in your ECALL.

The Intel SGX SDK's solution to this problem is to copy the contents of data buffers into and out of enclaves, and have the ECALLs and OCALLs operate on these copies of the original memory buffer. When you pass a pointer into an enclave, you specify in the EDL whether the buffer referenced by the pointer is being pass into the call, out of the call, or in both directions, and then you specify the size of the buffer. The proxy functions generated by the Edger8r tool use this information to check that the address range does not cross the enclave boundary, copy the data into or out of the enclave as indicated, and then substitute a pointer to the copy of the buffer in place of the original pointer.

This is the slow-and-safe approach to marshaling data and pointers between unprotected memory and enclave memory. However, this approach has drawbacks that may make it undesirable in some cases:

- It's slow, since each memory buffer is checked and copied.
- It requires additional heap space in your enclave to store the copies of the data buffers.
- The EDL syntax is a little verbose.

There are also cases where you just need to pass a raw pointer into an ECALL and out to an OCALL without it ever being used inside the enclave, such as when passing a function pointer for a callback function straight through to an OCALL. In this case, there is no data buffer per se, just the pointer address itself, and the marshaling functions generated by Edger8r actually get in the way.

The Solution: user_check

Fortunately, the EDL language does support passing a raw pointer address into an ECALL or an OCALL, skipping both the boundary checks and the data buffer copy. The `user_check` parameter tells the Edger8r tool to pass a pointer as it is and assume that the developer has done the proper bounds checking on the address. When you specify `user_check` you are essentially trading safety for performance.

A pointer marked with the `user_check` does not have a direction (`in` or `out`) associated with it, because there is no buffer copy taking place. Mixing `user_check` with `in` or `out` will result in an error at compile time. Similarly, you don't supply a `count` or `size` parameter, either.

In the Tutorial Password Manager, the most appropriate place to use the `user_check` parameter is in the ECALLs that load and store the encrypted password vault. While our design constraints put a practical limit on the size of the vault itself, generally speaking these sorts of bulk reads and writes benefit from allowing the enclave to directly operate on untrusted memory.

The original EDL for *ve_load_vault()* and *ve_get_vault()* looks like this:

```
1 public int ve_load_vault ([in, count=len] unsigned char *edata,
  uint32_t len);
2
3 public int ve_get_vault ([out, count=len] unsigned char *edata,
  uint32_t len);
```

Rewriting these to specify `user_check` results in the following:

```
1 public int ve_load_vault ([user_check] unsigned char *edata);
2
3 public int ve_get_vault ([user_check] unsigned char *edata, uint32_t
  len);
```

Notice that we were able to drop the *len* parameter from *ve_load_vault()*. As you might recall from Part 4, the issue we had with this function was that although the length of the vault is stored as a variable in the enclave, the proxy functions don't have access to it. In order for the ECALL's proxy functions to copy the incoming data buffer, we had to supply the length in the EDL so that the Edger8r tool would know the size of the buffer. With the `user_check` option, there is no buffer copy operation, so this problem goes away. The enclave can read directly from untrusted memory, and it can use its internal variable to determine how many bytes to read.

However, we still send the length as a parameter to *ve_get_vault()*. This is a safety check to ensure that we don't accidentally overflow a buffer when fetching the encrypted vault from the enclave.

Summary

The EDL provides three options for passing pointers into an ECALL or an OCALL: `in`, `out`, and `user_check`. These options are summarized in Table 1.

Specifier/Direction	ECALL	OCALL
in	The buffer is copied from the application into the enclave. Changes will only affect the buffer inside the enclave.	The buffer is copied from the enclave to the application. Changes will only affect the buffer outside the enclave.
out	A buffer will be allocated inside the enclave and initialized with zeros. It will be copied to the original buffer when the ECALL exits.	A buffer will be allocated outside the enclave and initialized with zeros. This untrusted buffer will be copied to the original buffer in the enclave when the OCALL exits.
in, out	Data is copied back and forth.	Data is copied back and forth.
user_check	The pointer is not checked. The raw address is passed.	The pointer is not checked. The raw address is passed.

Table 1. Pointer specifiers and their meanings in ECALLs and OCALLs.

If you use the direction indicators, the data buffer referenced by your pointer gets copied and you must supply a count so that the Edger8r can determine how many bytes are in the buffer. If you specify `user_check`, the raw pointer is passed to the ECALL or OCALL unaltered.

Sample Code

The code sample for this part of the series has been updated to build against the Intel SGX SDK version 1.7 using Microsoft Visual Studio 2015. It should still work with the Intel SGX SDK version 1.6 and Visual Studio 2013, but we encourage you to update to the newer release of the Intel SGX SDK.

Coming Up Next

In Part 8 of the series, we'll integrate the GUI with our application. Stay tuned!