Using the Intel® Software Guard Extensions (Intel® SGX) SSL Library

Scope

This paper describes how to use the Intel® Software Guard Extensions (Intel® SGX) SSL Library for Intel SGX enabled applications. The Intel SGX SSL enclave libraries are derived from OpenSSL* and can be used for secure communications (TLS) across the network and for cryptographic operations within an enclave. This paper covers applications targeting Microsoft* Windows*. The content assumes an understanding of Intel SGX. General information on Intel SGX can be found on the Intel SGX portal at: https://software.intel.com/sgx.

Introduction

OpenSSL is arguably the most widely used TLS and cryptographic toolkit in the world. It's used to provide secure communications in web browsers, IOT devices, server applications, etc. The Intel SGX SSL library is a derivative work that augments OpenSSL to support both cryptographic operations and TLS termination from within an Intel SGX enclave.

The goal of this paper is to describe the steps needed to enable TLS termination from your Intel SGX enclave. Use of TLS termination from the enclave protects the communication session key and keeps the data being exchanged with the enclave over the network confidential.

SgxSSL communication basics

The following subsections describe SSL TLS communication basics. (Jump to the next section if you already know this.)

Setting up the SSL context and the BIO object

SgxSSL uses an abstraction library called `BIO` to handle communication of various kinds, including files and sockets, both secure and unsecure. Prior to setting up a connection, whether secure or unsecure, create a pointer for a `BIO` object in the application.

```
        BIO bio;
```

Setting up a TLS connection

TLS protocol requires a handshake after the connection is established. During the handshake, the server sends a certificate to the client and the client verifies it against the set of trusted certificates. It also make sure that the certificate hasn't expired.

The client sends a certificate to the server only if the server requests one. This is known as client authentication. Using the certificate(s), cipher parameters are passed between the client and server to set up the secure connection. Even though the handshake is performed after the connection is established, the client or server can request a new handshake at any point in time.

To establish a secure TLS connection, execute the following instructions:

1. Create an `SSL_CTX` pointer for holding SSL information and another pointer named `SSL` is to store the SSL connection structure, as follows:

   ```
   SSL_CTX * ctx = SSL_CTX_new(TLSv1_2_client_method());
   SSL * ssl;
   ```

2. Next load a trust certificate store. This is absolutely necessary for the verification of peer certificate to succeed.

   Note: One choice is to access the Windows Certificate Store, enumerate the certificates, and add them to OpenSSL. For details, see: https://stackoverflow.com/questions/9507184/can-openssl-on-windows-use-the-system-certificate-store.

3. Call `SSL_CTX_load_verify_locations` to load the trust store file, as follows:

   ```
   if (! SSL_CTX_load_verify_locations (ctx, "/path/to/TrustStore.pem", NULL))
       {
       /* handle failed to load here */
       }
   ```

4. Then the `BIO` object is created using `BIO_new_ssl_connect ()` and the previously created `ctx` as an argument. After creating the `BIO` object, the SSL structure is retrieved, as follows:

   ```
   bio = BIO_new_ssl_connect (ctx);
   BIO_get_ssl (bio, &ssl);
   ```

5. With the `SSL` context structure set up, the connection can be created. The hostname is set using the `BIO_set_conn_hostname` function. This function also opens the connection to the host.

Communicating with the Server

Reading and writing to the `BIO` object, regardless of whether it is a socket or file, is always be performed using the following two functions:

- *BIO_read:*

  `BIO_read` attempts to read a specified number of bytes from the server. It returns the number of bytes read or 0 or -1.

  - On a blocking a connection, a return of 0 means that connection was closed, while -1 indicates an error.
  - On a non-blocking connection, a return of 0 means no data was available and -1 indicates an error.

    Data can be read using the *bio* connection object as follows:

```
int x = BIO_read (bio, buf, len);

if (x == 0)
    {
    /* Handle closed connection */
    }
else if (x<0)
    {
    if (!bio_should_retry (bio))
        {
        /* Handle failed read here*/
        }
    }
```

- *BIO_write:*

  Similarly `BIO_write` attempts to write bytes to the socket. It returns the number of bytes actually written, or 0 or -1.

    Data can be written to socket as shown below:

```
if ((BIO_write(bio, buf, len) <= 0)

    {
    if (bio_should_retry(bio)
         {
         /* Handle failed write here */
         }
    }
```

Closing the connection

Closing the connection is done using one of the following two `BIO` API functions:

- *BIO_reset:*

  If the connection object needs to be reused another time, use `BIO_reset`. The `BIO_reset` command, shown below, closes the connection and resets the internal state of the `BIO` object so that the connection can be reused.

  ```
  BIO_reset(bio);
  ```

- *BIO_free_all:*

  The `BIO_free_all` command, shown below, frees the internal structure and releases all the associated memory, which includes closing the associated sockets.

  ```
  BIO_free_all (bio);
  ```

Integrating the SGXSSL enclave library

The SgxSSL Library can be included in an enclave by adding it as a static library and linking it with the enclave code. The following software is needed for enclaving the SgxSSL:

- Microsoft Visual Studio* 2015
- Intel SGX SDK for Windows 1.8
- Intel SGX Platform SW Installer 1.8
- Full set of Intel Management Engine (ME) software components
- Intel SgxSSL Library source

The APIs exposed by the Intel SgxSSL library are fully compliant with unmodified OpenSSL APIs. However, only a specific subset of APIs available in OpenSSL is supported by the Intel SgxSSL library, namely the APIs needed for enclaving.

Note: OpenSSL APIs that are *not* supported are still included in the Intel SgxSSL library, but since they are not validated, you should *not* use them.

The Intel SgxSSL library consists of the following components, as shown in Figure 1:

- Intel SgxSSL cryptographic and TLS libraries (derived from OpenSSL libraries) built to run inside an enclave.
- A trusted library providing implementation for system APIs inside an enclave (not derived from OpenSSL).
- An untrusted library providing implementation of system APIs outside an enclave (not derived from OpenSSL).

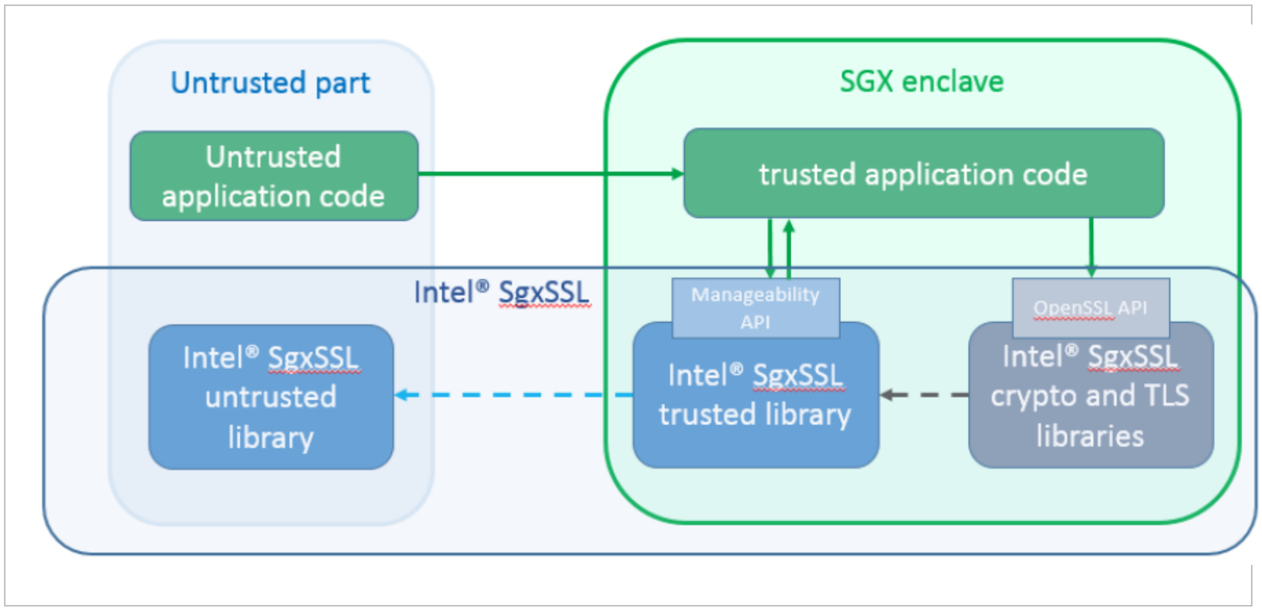
Untrusted part
SGX enclave
Untrusted application code
trusted application code
Intel® SgxSSL
Manageability API
OpenSSL API
Intel® SgxSSL untrusted library
Intel® SgxSSL trusted library
Intel® SgxSSL crypto and TLS libraries

Figure 1: Intel SGXSSL library usage

Intel SGX applications add SgxSSL-specific functions in the form of trusted and untrusted libraries. The Intel SgxSSL trusted library provides the functionality for handling the trusted system specific calls like crypto and TLS. The Intel SgxSSL-specific untrusted library provides the functions for handling untrusted system-specific OCalls from enclaved SgxSSL.

The SgxSSL-specific functionality execution control flow in Intel SGX is as follows:

1. The untrusted application code calls the trusted code with a function declared in the EDL (Enclave Description Language) file.
2. The trusted application code may use manageability APIs for purposes such as getting the Intel SgxSSL library version, registering a callback function to intercept the Intel SgxSSL cryptographic or TLS libraries printout messages, etc.
3. The trusted code continues execution and at a certain point calls an API supported by the Intel SgxSSL cryptographic or TLS libraries.
4. The call is passed to the Intel SgxSSL cryptographic or TLS library. Some functions are internal and don't rely on system APIs (for example, SHA256) so the functions complete and return.
5. Other functions require system APIs, so the execution passes to the Intel SgxSSL trusted library code that implements them. If the system API is simple and can be implemented internally (for example, `__gmtime64`), it returns after completion without leaving the enclave.
6. Other APIs must leave the trusted code and are executed in the untrusted area (for example, `send`, `recv`, etc.)

Building Intel SGX application with Intel SgxSSL

To implement the Intel SGX application with enclaved SgxSSL, the trusted and untrusted libraries listed in Table 1 are required.

Table 1: Intel SgxSSL library names and usage

Library Name	Description
sgx_tfipscanister.lib	FIPS object module library built for Intel SgxSSL cryptographic library.
sgx_tlibeayfips32.lib	FIPS capable Intel SgxSSL cryptographic library (derived from OpenSSL crypto library).
sgx_tssleay32.lib	Intel SgxSSL SSL/TLS library (derived from OpenSSL SSL/TLS library).
sgx_tssl.lib	Trusted library, providing implementation for system APIs required by Intel.
sgx_ussl.lib	Untrusted library, providing implementation for system calls outside an enclave required to resolve external dependencies on Intel SgxSSL cryptographic and TLS libraries.

The following discussion assumes that an Intel SGX application is already built with the necessary trusted and untrusted libraries. To enclave SgxSSL in this Intel SGX application, the Intel SgxSSL libraries should be added in the Visual Studio project settings as follows:

1. Add the following statement in the application enclave EDL file:

   ```
   "sgx_tssl.edl" import *;
   ```

   This file contains the Ocall targeting the untrusted system call implemented outside an enclave that is required to resolve external dependencies of Intel SgxSSL cryptographic and TLS libraries.
2. Add `#include <windows.h>` before the `#include <openssl\xxx>` statements. The `windows.h` file is found in the Intel SGX SSL `include` directory, and contains several definitions required by many of the SgxSSL headers.
3. In the Visual Studio enclave project, perform the following steps:
   a. Select **Properties->Linker->Input->Additional Dependencies**: Add `sgx_tlibeayfips32.lib; sgx_tssleay32.lib; sgx_tssl.lib sgx_tfipscanister.lib`
   b. Select **Properties->Linker->General->Additional Library Directories:** Add the folder where you placed the libraries, (Hint: Use the built-in macros like `$(SolutionDir)$(Platform)\$(Configuration)\` etc., so you can control the different builds)
   c. Also add the path to `sgx_tfipscanister.lib`, for instance: `<path to the package>\lib\fipsopenssl\$(Platform);`
   d. To add the folder where you placed the EDL file, right click your EDL file, then select **Properties->Custom Build Tool->Command Line**: Add the EDL file path to the `--search path` separated with `;`

   e. Select **Properties**->**C/C++**->**General**->**Additional Include Directories** and add the folder where Intel SgxSSL header files are located. `<installed_package>\include)`
   f. Select **Properties->Linker->All Options->Image Has Save Exception Handlers**: Choose `No (/SAFESEH:NO`)
4. In the **Application** project, perform the following steps to set up the environment for the Intel SgxSSL library:
   a. Select **Properties->Linker->Input->Additional Dependencies**: Add `sgx_ussl.lib;Ws2_32.lib`
   b. Select **Properties->Linker->General->Additional Library Directories**: Add the folder where you placed the libraries (Hint: Use the built in macros like `$(SolutionDir)$(Platform)\$(Configuration)\` etc., so you can control the different builds)
   c. To add the folder where you placed the EDL file, right click your EDL file, then select **Properties->Custom Build Tool->Command Line**: Add the EDL file path to the `--search path` separated with `;`
   d. If your project does not use the Intel compiler, add the path to the Intel compiler libraries through **Properties->Linker->General->Additional Library Directories**

Build with FIPS capable SgxSSL

To configure your Visual Studio project with the FIPS capable SgxSSL Library, perform these steps for all your build environments:

1. In the **Enclave** project:
   - In 32-bit configuration, select **Properties->Linker->Advanced**
     - Set **Randomized Base Address** to `No(/DYNAMICBASE:NO)`
     - Set **Fixed Base Address** to `Yes(/Fixed)`
       - Set the **Base Address**
         Note: The SDK requires the Base Address to be enclave-size aligned.

   Note: This step is mandatory for the FIPS runtime integrity check in Windows 32-bit configuration only to ensure the digest calculated during build is the same as the digest calculated at runtime. It is *not* mandatory in a 64-bit configuration, as 64-bit binaries do not contain any reallocations.

2. Copy `<path_to_the_package>\lib\fipsopenssl\$(Platform)\fips_premain.c` into your enclave project and add the copied file into the enclave project files.
3. Build your project.
4. Calculate the FIPS module fingerprint and incorporate it in the enclave DLL:
   a. Open VS Win32/X64 CMD:
      - Change directory to the `<path to the package>\util`

- Calculate the hash value by running `perl <release_package>\util\msincore -dso <output_path>\<enclave_name>.dll`

  Note: The hash value is calculated from FIPS module text and data areas. The calculated value is not impacted by the enclave signing.

Note: Instead of opening VS Win32/X64 CMD, you can calculate the HASH as Post-Build Event by adding an additional command to the **Properties->BuildEvent->CommandLine**. In this case the command will look like:

```
C:\Perl64\bin\perl.exe <path_to_the_package>\util\msincore -dso
$(OutDir)\<enclave_name>.dll
```

b. Select **Properties->C/C++->Preprocessor**
Add `HMAC_SHA1_SIG="<calculated_hash_value>"`
For instance, if calculated hash value is `8fe5425f0c5b1d944b89c27fb4a9d1d19a2cb9cd`, then you should put it in "" to define HMAC_SHA1_SIG, like:
`HMAC_SHA1_SIG="8fe5425f0c5b1d944b89c27fb4a9d1d19a2cb9cd"`

c. Rebuild your project.

Note: In the current SGX SDK, *Release* mode does not generate the `enclave.signed.dll` file, but rather prepares signing material (since it must signed in a secure machine that protects the private key, etc.). Enclaves signed with single-step signing method using ISV's test private key can only be launched in *Debug* or *Prerelease* modes.

SgxSSL API

The Intel SgxSSL library exposes two sets of APIs:

- *SgxSSL APIs* — derived from OpenSSL APIs. These APIs are fully complaint with the unmodified OpenSSL APIs. Table 2 lists these SgxSSL APIs derived from OpenSSL.
- *Manageability APIs* — exposed by our trusted library to provide manageability services needed for use in a system. Table 3 lists these Manageability APIs.

Table 2: Supported OpenSSL APIs

Purpose	Type	API
Digest	SHA1, SHA256	SHA1_Init SHA1_Update SHA1_Final SHA256_Init
Key Generation	RSA 2048	RSA_new RSA_free RSA_generate_key PEM_read_PrivateKey

Purpose	Type	API
Asymmetric Encryption	RSA PKCS #1	RSA_generate_key_ex RSA_public_encrypt
Symmetric Encryption	3DES, AES128 ECB	EVP_CIPHER_CTX_init EVP_CipherInit_ex EVP_CipherFinal_ex
Sign and Verification	X.509 certificate(PKCS#7)	EVP_MD_CTX_init EVP_MD_CTX_create EVP_MD_CTX_destroy EVP_SignUpdate Ecp_SIgnFinal
Certificate signing request	X.509 Certificate	X509_free X509_NAME_add_entry_by_txt X509_REQ_set_pubkey X509_REQ_new X509_REQ_sign
Certificate Revocation List (CRL)		X509_CRL_new X509_STORE_add_cr1 X509_CRL_sign X509_CRL_verify X509_CRL_free
Random Number Generator	Add entropy to random number generator	RAND_add RAND_seed
FIPS mode	FIPS mode	FIPS_mode_set FIPS_mode

Table 3: Manageability APIs

API	Description
setPrintToStdoutStderrCB	Set Callback function to intercept printouts sent by IntelSgxSSL cryptographic and TLS libraries to stdout/stderr. If not used, printout is silently omitted.
setProxyCertsPolicy	Set proxy certificate policy. For regular OpenSSL this policy is provided via OPENSSL_ALLOW_PROXY_CERTS environment variable. Intel SGX OpenSSL library will ignore the value provided by this environmental variable.
getSgxSSLVersion	Get the Intel SgxSSL library version.
setUnreachableCodePolicy	Set unreachable code policy. Unreachable code consists of functions and flows that our implementation should never reach. That is why by default, reaching unreachable code will cause an enclave to be aborted.

Security challenges and recommendations

Intel SgxSSL provides support for OpenSSL inside an enclave. Security assets, like cryptographic keys, client certificate private keys, plain data (both network traffic and cryptographic payload) don't need to leave an enclave and thus are protected by Intel SGX technology. Some security challenges, however, still exist.

- **Side Channel Attack**: The Intel SgxSSL library relies on an implementation of OpenSSL and Intel SGX to help mitigate side channel attacks. Our architecture is designed to not leak additional information through the OCALLS, but it does *not* protect against side channel attacks. In case of side channel attacks, it is as secure as OpenSSL without Intel SGX.
- **Time-based Attack**: Current system time is *not* supported inside an enclave. So it is implemented by Intel SgxSSL library as OCALL. This approach allows an attacker to manipulate the time values coming from an untrusted component. Time values are used by Intel SgxSSL library for time-related certificate verification checks as well as TLS session expiration checks. So there is a chance for time-based attack. If the ISV's enclave application utilizing Intel SgxSSL library uses server certificate pinning, the risk for a time-based attack is low. Therefore, SGX enclave applications should implement server certificate pinning.

Intel SGX Application security expectations and recommendations

- The ISV's application is responsible for building a production enclave as a non-debug enclave.
- The ISV's application should use the Intel SGX architecture and Intel SGX SW to protect security assets. For instance, the ISV's application should use certificate pinning. Server certificates should be securely provisioned into an enclave and protected by enclave sealing capabilities.
- The ISV's application should not expose security assets via trusted-to-untrusted transitions.
- The ISV's application should sanitize input data coming from untrusted components.
- The ISV's application should use only the SSL API versions that are included in the Intel SgxSSL library, as OpenSSL has not been verified with Intel SGX.
- The ISV's application should use SgxSSL API correctly to verify server certificates.

Summary

In situations where two communicating Intel SGX applications reside across the network, SSL can be used to secure that communication. The SSL implementation, however, must be included in the enclave to protect this communication.

The Intel SgxSSL library supports a subset of OpenSSL library APIs to provide the services like Sign and Verification, Certificate Signing, Certificate Revocation, Random Number generation, etc. The SgxSSL APIs are fully complaint with unmodified OpenSSL APIs. Intel SgxSSL also supports Manageability APIs, which are exposed by the trusted library to provide basic manageability services.

To integration the Intel SgxSSL library into an enclave, add the recommended trusted and untrusted SgxSSL libraries to the Intel SGX application. Security assets, like cryptographic keys, client certificate private keys, plain data (both network traffic and cryptographic payload) can then be protected by Intel SGX technology that implements SgxSSL.

Even with Intel SGX and the Intel SgxSSL Library, there are some security challenges that remain, like Side Channel Attacks and Time-based Attacks. Intel SGX application developers should consider these security challenges and the recommendations provided as they develop their secured application.

References

1. Intel® Software Guard Extensions SDK Users Guide for Windows* OS — 2017 Intel Corporation. https://software.intel.com/sgx-sdk/documentation.
2. Secure Programming with OpenSSL API — 2004 Kenneth Ballard, Software Engineer, MediNotes Corp.
3. Intel® SgxSSL Library User Guide — 2016 Intel Corporation. https://software.intel.com/sgx-sdk/download.