Intel® SGX Web Training

Hands-on Lab

Manual
Version 1.0

Overview

In this lab, you will explore features of Intel® Software Guard Extensions (Intel® SGX). The Hands-on Lab web course is designed to be developer-centric, and it focuses more on using Intel SGX than on discussing its theoretical underpinnings.

Topics in this lab will be presented through a combination video presentations and self-directed, hands-on work directly with Intel SGX. Time-wise, you will spend the majority of your time working directly with Intel SGX, either coding or following along with examples.

Task 0: Lab Setup

Because this lab uses the Intel SGX software-development kit (SDK), the development environment needs to be set up accordingly. This preparatory step is added for reference and is not part of the actual hands-on lab.

Task 0 Steps

1. Install Microsoft® Visual Studio® 2015 Professional. Community Edition is not sufficient because it does not support plugins.
2. Install the Intel SGX SDK.

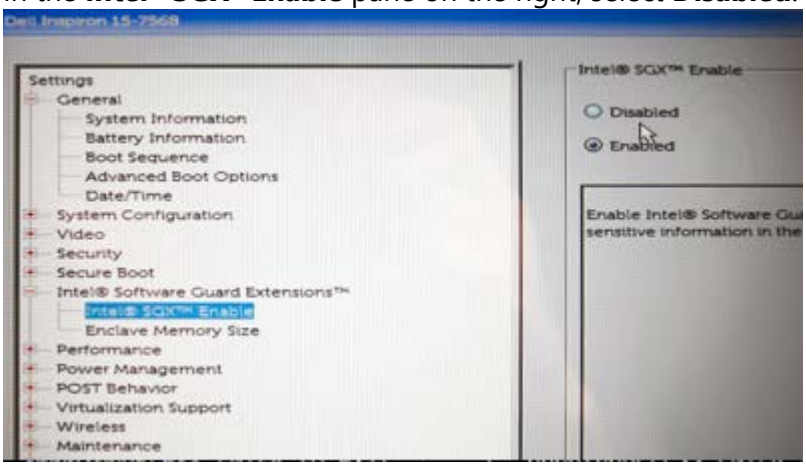
   (Optional) Install the Intel SGX platform software (PSW).

Task 1: Intel SGX Overview

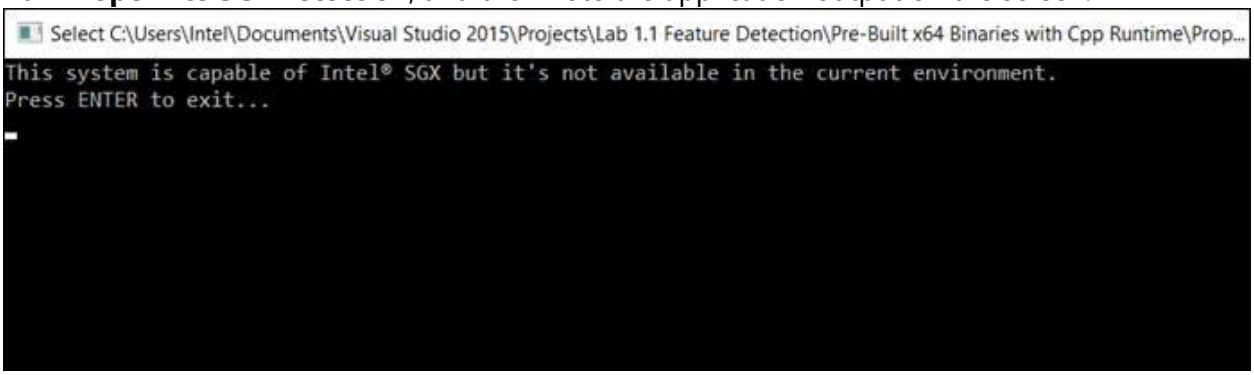
Task 1.1: Intel SGX Settings in BIOS

Lab Objective	In this lab, we will view Intel SGX BIOS settings; if supported, the BIOS might include settings for Enabled, Software Controlled, and Disabled.

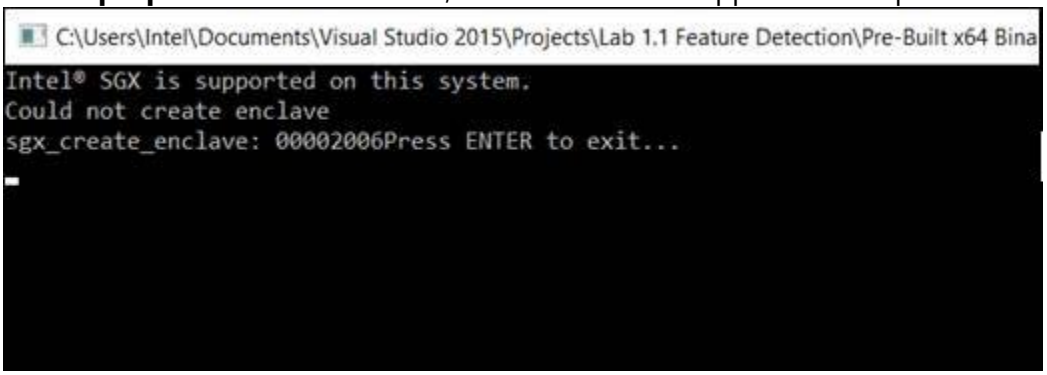
1. Boot an **Intel® SGX™** capable system. If you do not have access to such a system, follow along by viewing the screen captures.
2. Enter the BIOS configuration screen. (The method for this varies by manufacturer.)
3. Find the setting that controls **Intel® Software Guard Extensions™**. Note that not all **Intel® SGX™** systems provide BIOS options to enable and disable **Intel® SGX™**. If your system does not have these options, follow along by viewing the screen captures.
4. Select **Intel® SGX™ Enable**.
5. In the **Intel® SGX™ Enable** pane on the right, select **Disabled**.



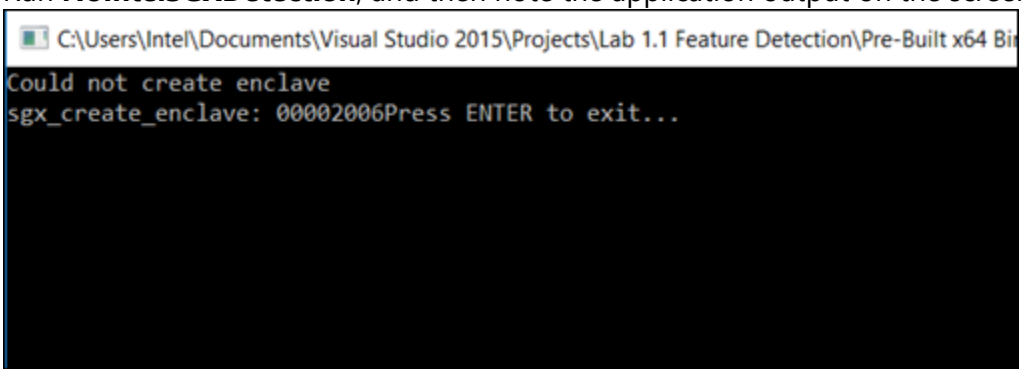
6. Exit the BIOS configuration screen and let the system reboot.
7. Once fully booted, download and unzip the Lab 1.1 package.
8. Run **ProperIntelSGXDetection**, and then note the application output on the screen.



9. Run **ImproperIntelSGXDetection**, and then note the application output on the screen.



10. Run **NoIntelSGXDetection**, and then note the application output on the screen.



11. Restart the system.
12. Re-enter the BIOS configuration screen.
13. In the **Intel® SGX™ Enable** pane on the right, select **Enabled**.

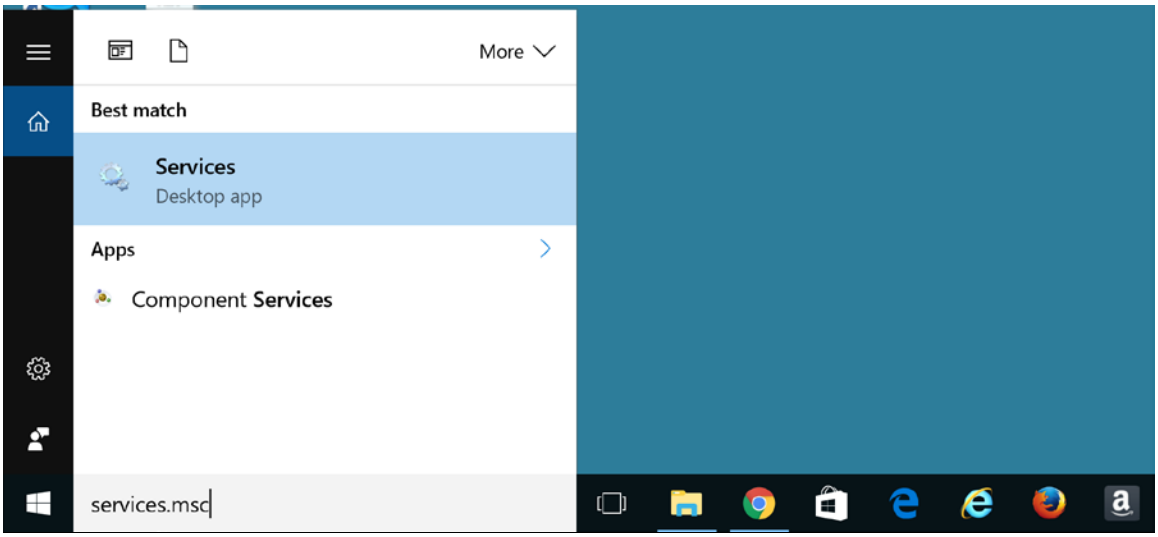
Exit the BIOS configuration screen and let the system fully boot to continue.

Task 1.2: Intel SGX AESM Service

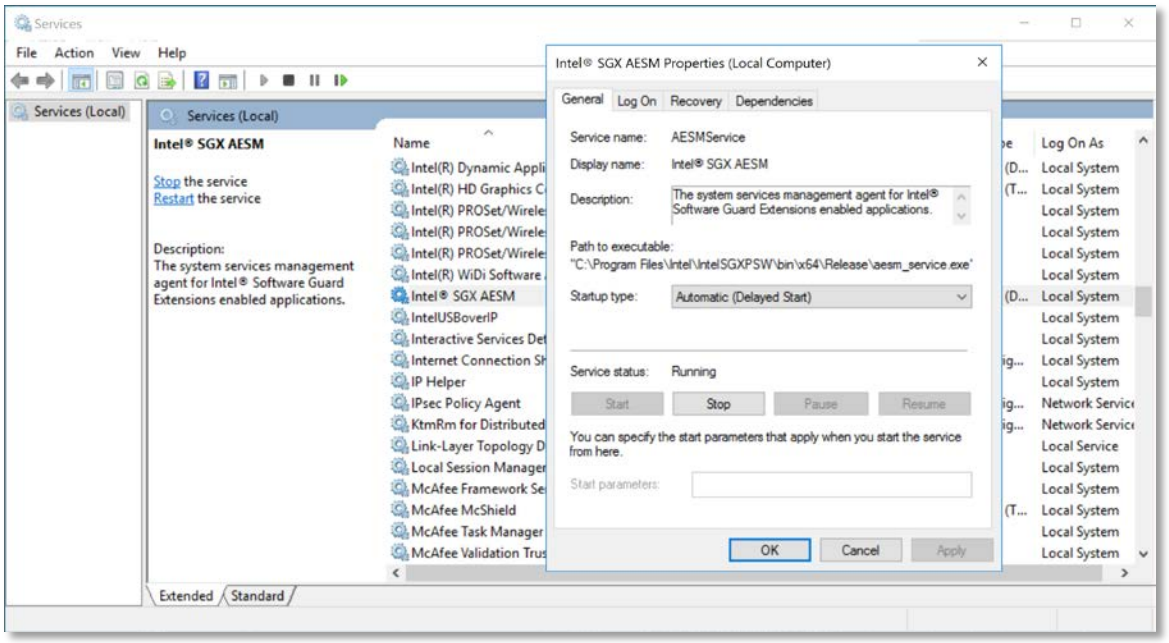
Lab Objective	In this lab, we will determine if the Intel SGX Application Enclave Services Manager (AESM) service is running from within the service-management console.

DEVELOPER TIP	Successful installation of the Intel SGX PSW on Intel SGX–enabled hardware should result in the Intel SGX AESM service running. Refer to the Intel SGX Installation Guide for additional information regarding prerequisites.

1. Launch the Start menu.
2. Type **services.msc**, and then press **Enter** to launch the service-management console.



3. Scroll down to **Intel® SGX AESM**.
4. Double-click to view the service's settings.

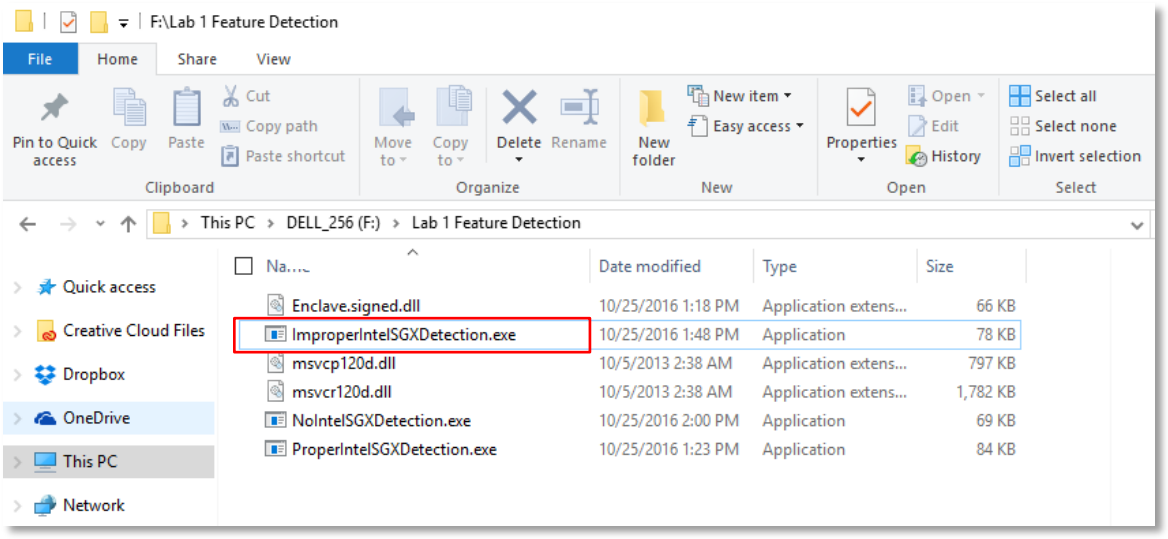


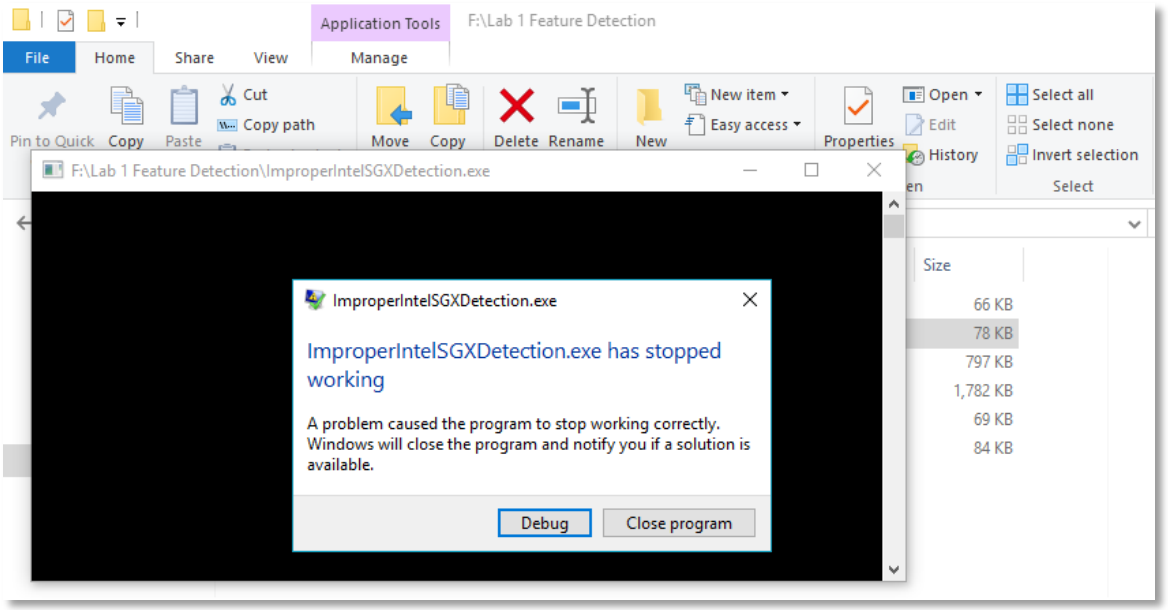
Task 1.3: Testing for Intel SGX Support

Lab Objective	In this lab, we will run precompiled examples that demonstrate why Intel SGX applications must be written to properly detect Intel SGX hardware and fail gracefully.

Run the following tasks from a laptop or desktop that *does not support* Intel SGX.

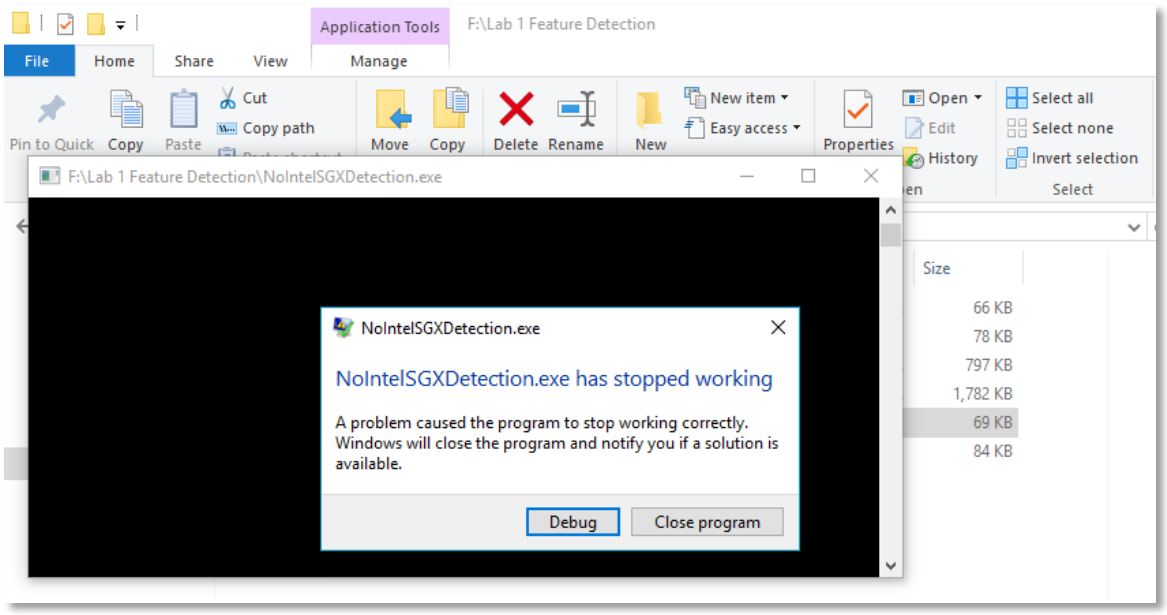
1. From the lab package, open **Labs\Lab 1.1 Feature Detection\Pre-Build x64 Binaries with Cpp Runtime** into Visual Studio. Copy the executables to a computer that does not support Intel SGX (these binaries will require the Microsoft Visual C++ Redistributable package, included in the directory).
2. Run **ImproperIntelSGXDetection.exe**.





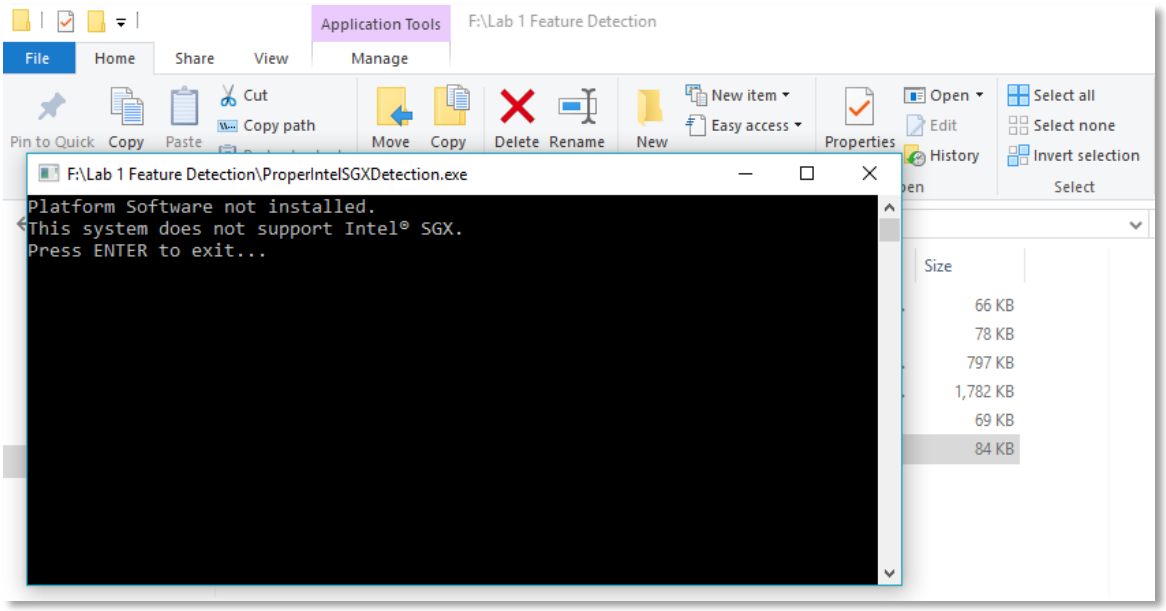
Result	Because Intel SGX was not detected on the machine, the application will crash. Intel SGX detection code is in the application; however, it was written in a way in which it does not exit gracefully.

3. Run **NoIntelSGXDetection.exe**.



Result	Because Intel SGX detection was not implemented, the application executed Intel SGX function calls without Intel SGX support. As a result, the application will crash without exiting gracefully.

4. Run **ProperIntelSGXDetection.exe**.

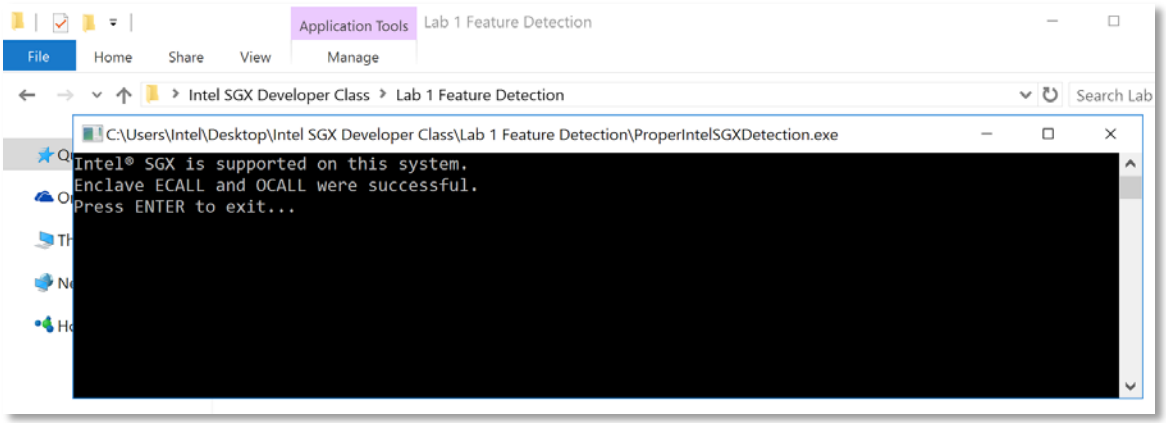


Result

By providing multiple code paths, the application was able to fail gracefully when it was unable to detect Intel SGX. As a result, the application is able to handle the error and report the issue without a crash.

Run the following from a laptop or desktop *that supports* Intel SGX.

1. Browse to **Labs\Lab 1.1 Feature Detection\Pre-Build x64 Binaries with Cpp Runtime**.
2. Run **ProperIntelSGXDetection.exe**.



Result

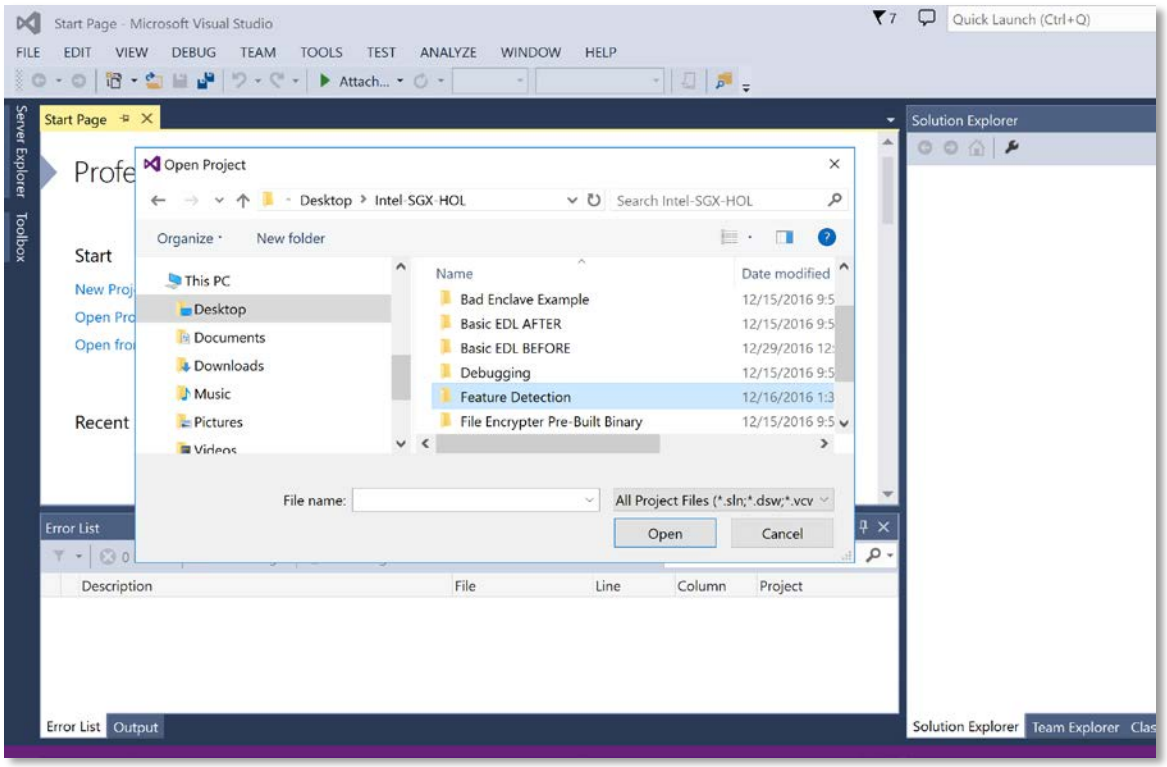
Notice that Intel SGX support was detected, Intel SGX ECALL and OCALL functions were executed successfully, and the functions were reported to the end user.

MORE INFO	Because we're now running the Intel SGX detection examples on an Intel SGX–capable and enabled device, there is no need to run **ImproperIntelSGXDetection.exe** or **NoIntelSGXDetection.exe**—they will execute successfully regardless of proper detection.
DEVELOPER TIP	Intel provides a detailed article and code samples for Intel SGX detection. Refer to the "Properly Detecting Intel® Software Guard Extensions in Your Applications" article for details.

Task 1.4: Review of Intel SGX Support

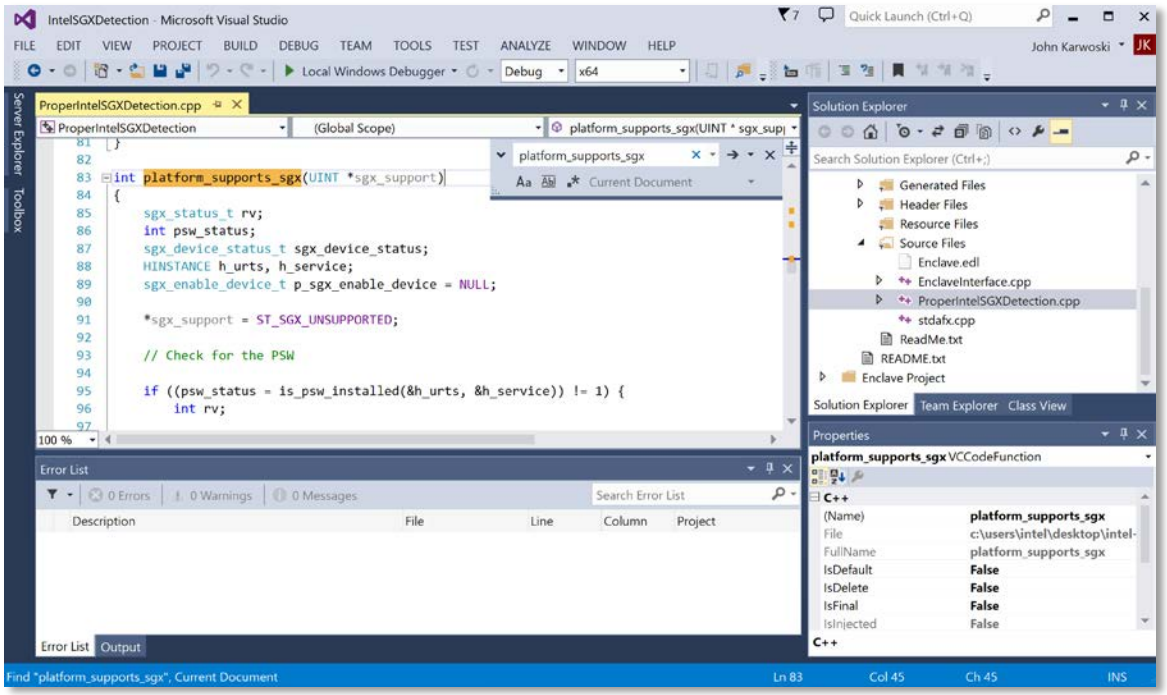
Lab Objective	In this lab, we will review and learn about key Intel SGX–detection function calls used in the **ProperIntelSGXDetection** project.

1. Launch **Visual Studio 2013**.
2. Click **Open Project**.
3. Browse to **Labs\Lab 1.1 Feature Detection**.

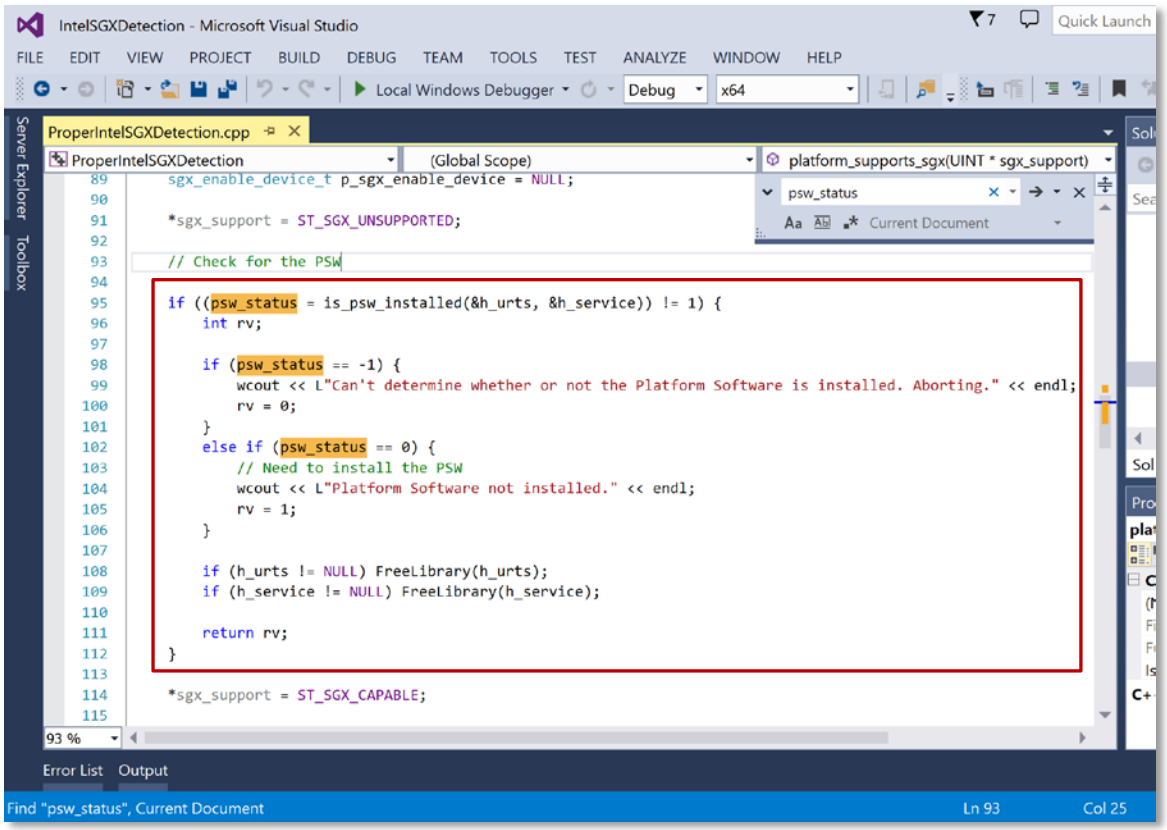


4. Select **IntelSGXDetection.sln**, and then click **Open**.
5. Expand **Application Project** > **ProperIntelSGXDetection** > **Source Files**.
6. Select **ProperIntelSGXDetection.cpp**.

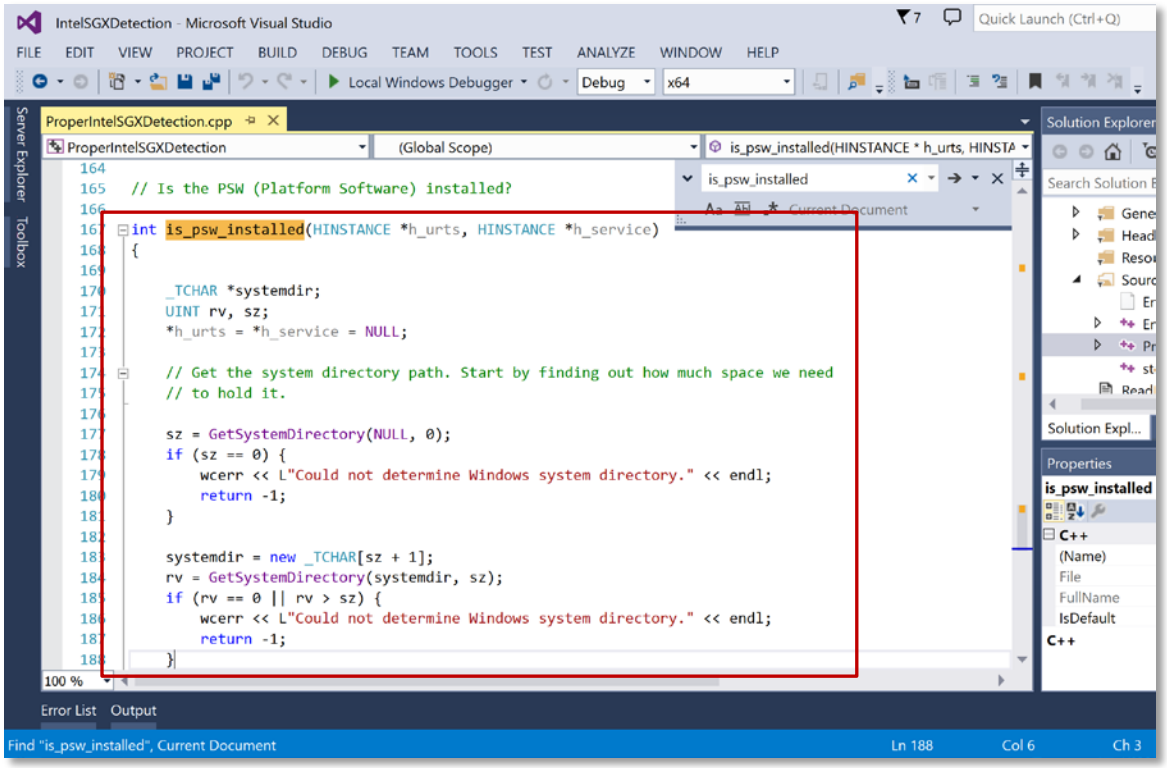
7. Go to the **platform_supports_sgx** function. This function checks if the hardware supports Intel SGX.



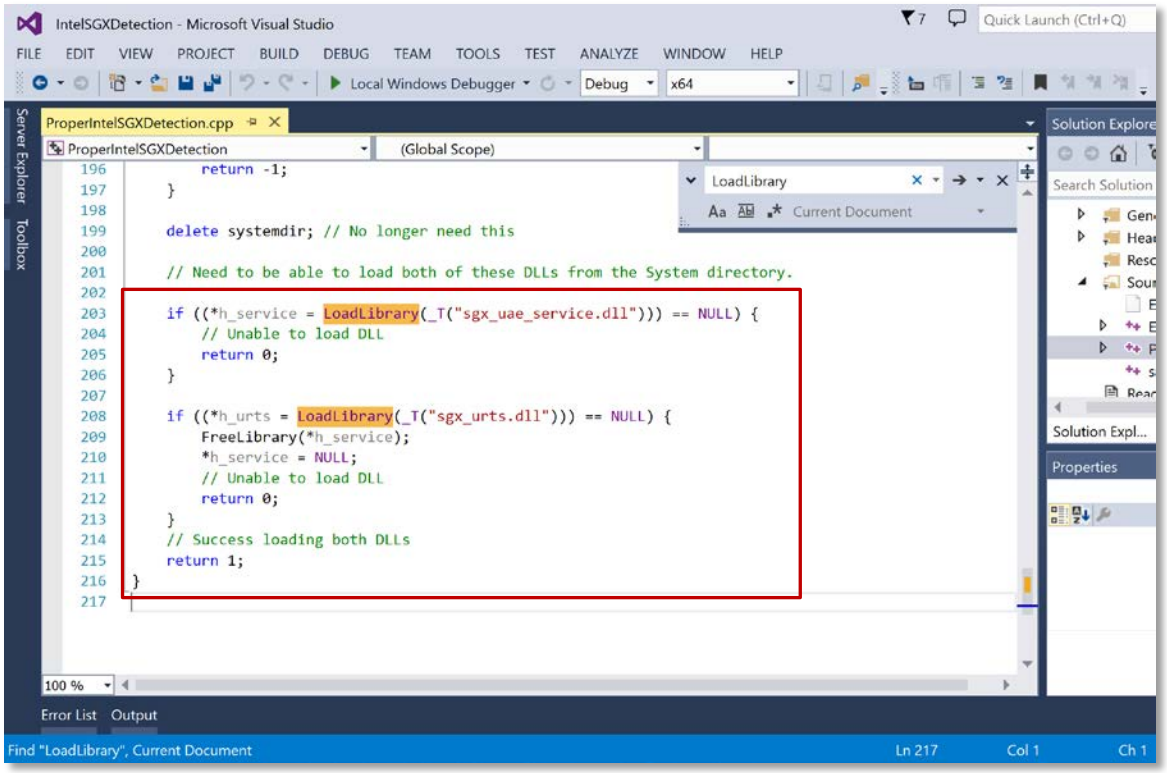
8. Go to **psw_status**. This function checks if the PSW exists and returns a value of True or False.



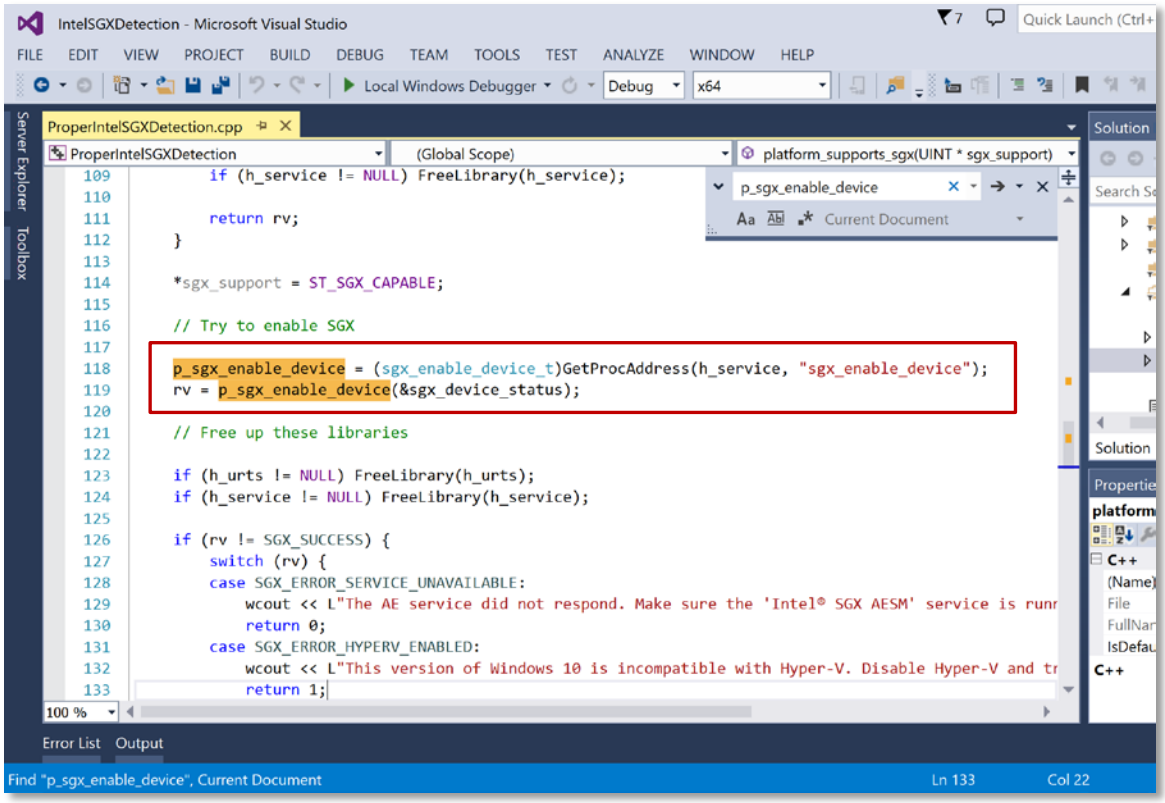
9. Go to **is_psw_installed**. This function attempts to load the PSW-required dynamic-link libraries (DLLs).



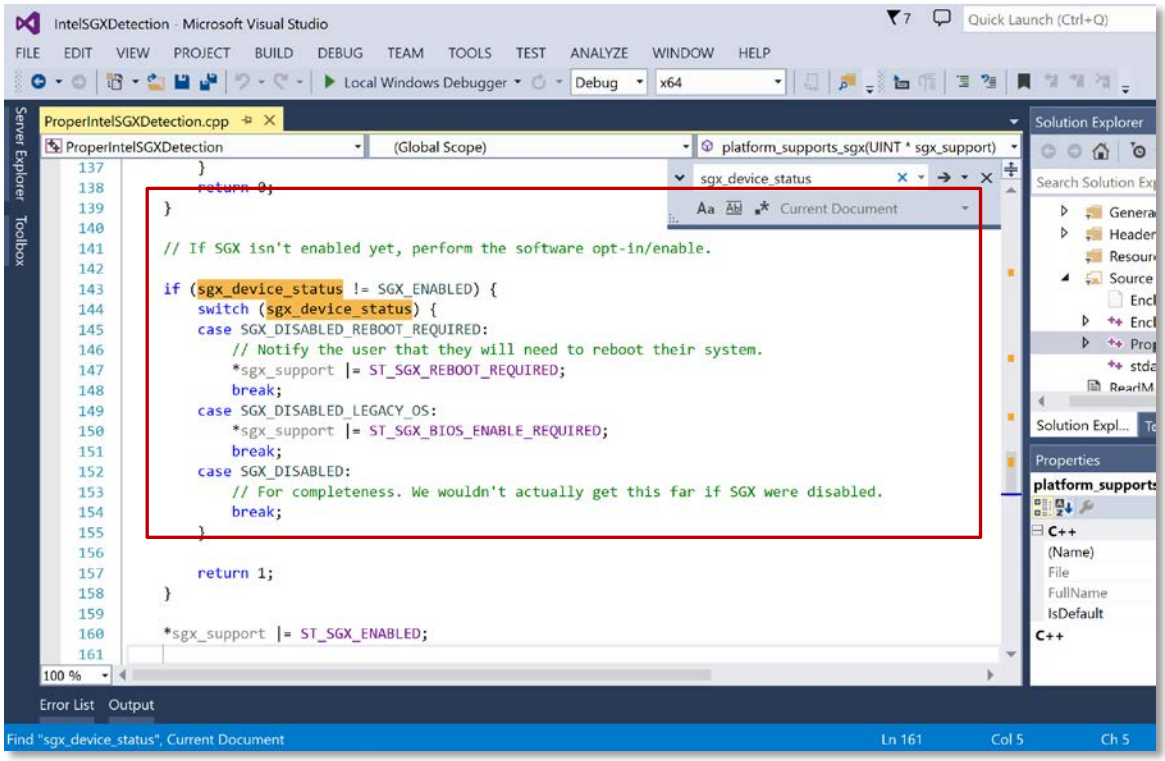
10. Go to the **LoadLibrary** functions. These functions load the DLLs. If they return NULL, the PSW is not installed and enclaves cannot run. The sgx_uae_service.dll and sgx_urts.dll DLLs are required in order to run enclaves.



11. Go to the **p_sgx_enable_device** function. This function attempts to enable Intel SGX if it's currently disabled.



12. Go to the **sgx_device_status** function. This function will check if Intel SGX is enabled on the device.

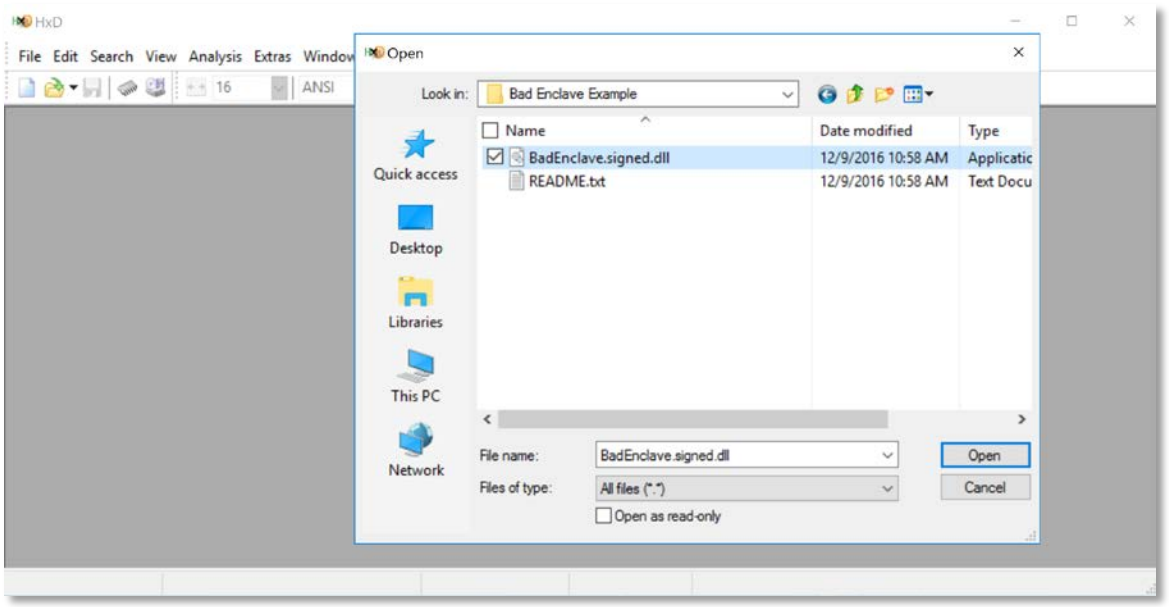


MORE INFO	Refer to the proper Intel SGX detection and multiple code paths within your Intel SGX application articles for more information.

Task 1.5: Don't Compile Secrets into Enclave DLLs

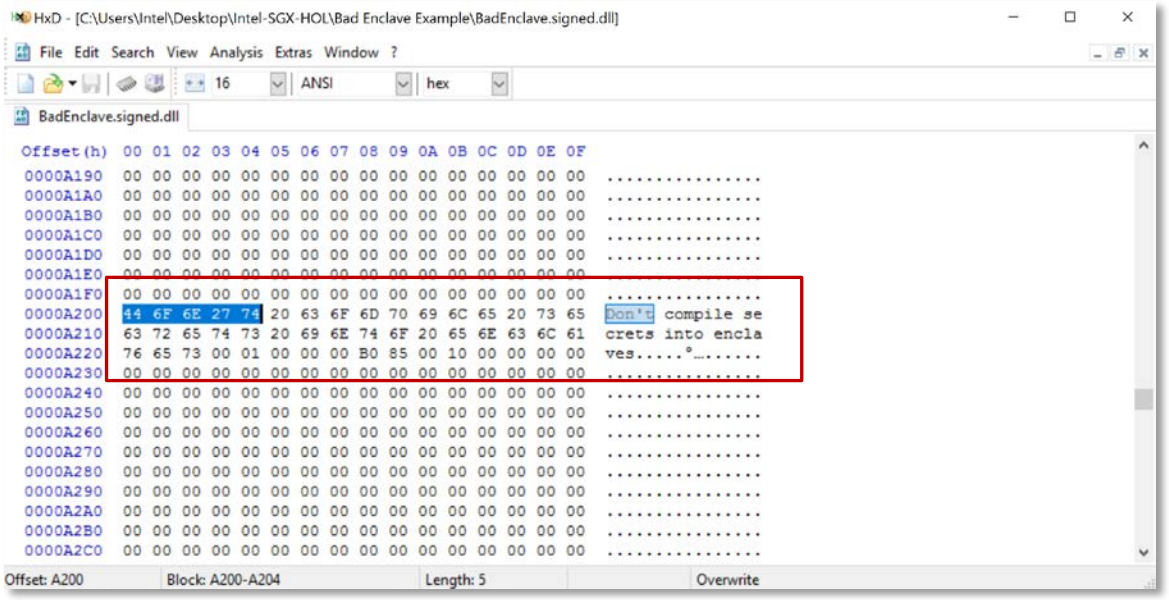
Lab Objective	In this lab, we will learn why it's critical that you don't compile secrets into your enclaves.

1. Launch a hex editor application (HxD is shown).
2. Select **File** > **Open**.
3. Browse to **Labs\Lab 1.5 Bad Enclave Example** in the lab package.
4. Open **BadEnclave.signed.dll**.



More Info	An enclave is built and loaded as a Windows® DLL or a shared-object (SO) library in Unix®. The enclave file can be disassembled; as a result, the algorithms used by the enclave developer will not remain secret.

1. Search for **Don't**. This shows the secret that was compiled in the DLL.



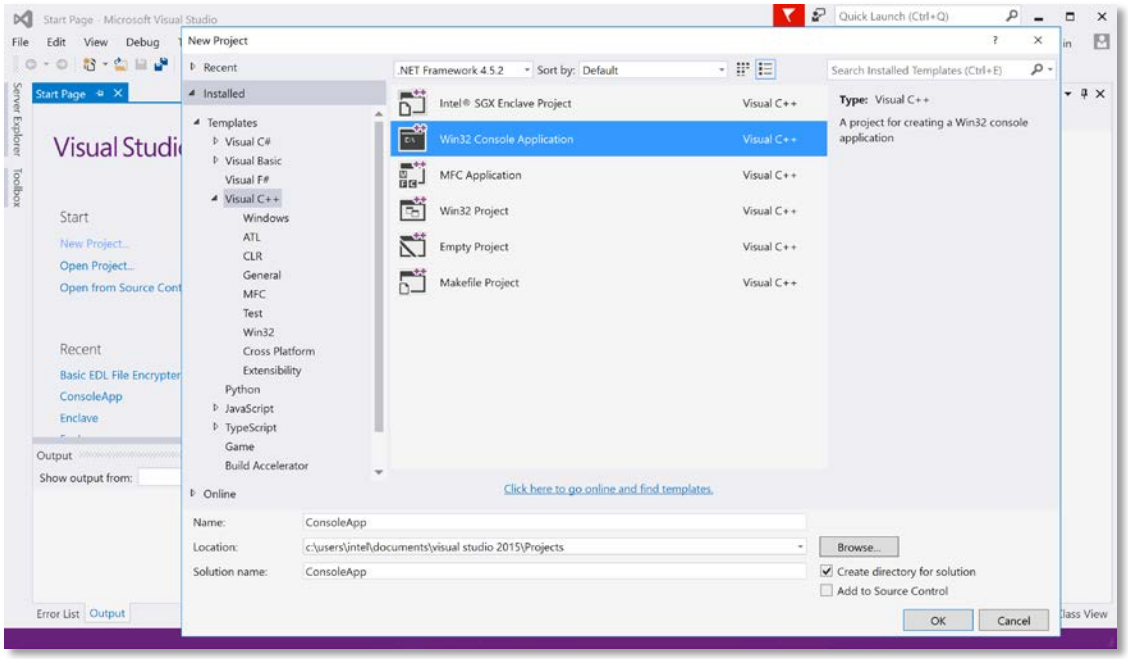
Result	Because the secret was compiled into the enclave DLL file, it's easily found. This is unsecure, and the secret is *not* safe.
DEVELOPER TIP	To secure your secret, remember these steps: 1. Write the enclave application without secrets. 2. Get the enclave application signed and bound to a CPU. 3. Provision secrets from a trusted remote client. 4. Run the enclave program in the CPU for trusted execution.

Task 2: Introduction to Intel SGX Applications

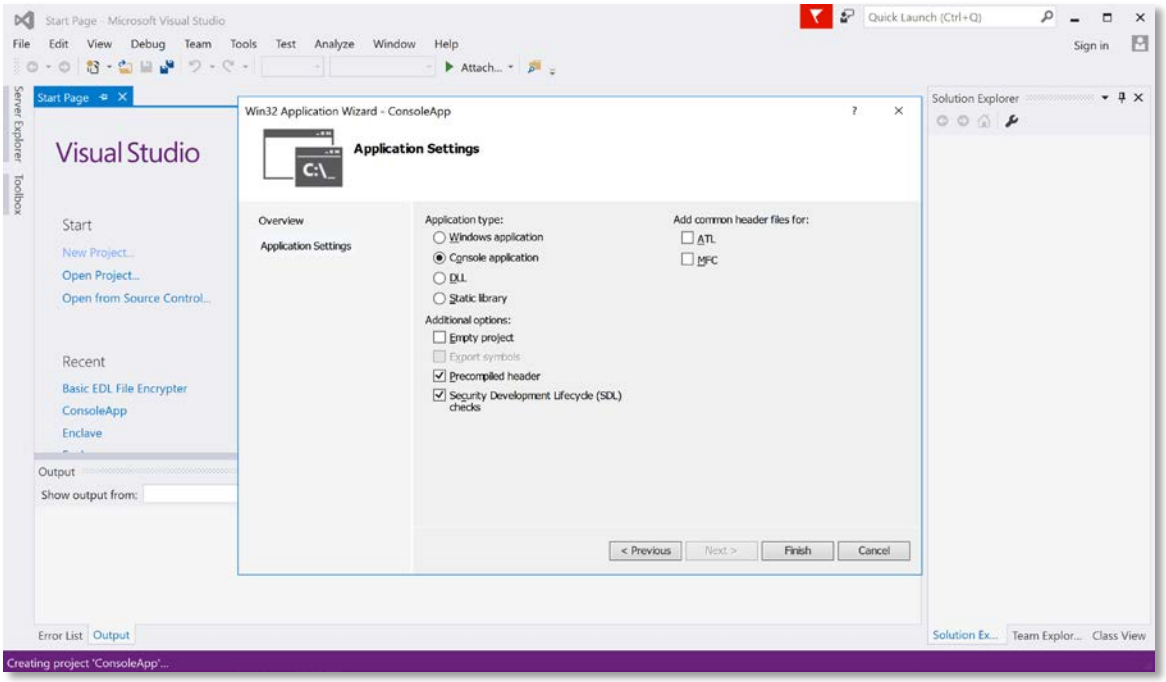
Task 2.1: Create a Console-Application Project

Lab Objective	In this lab, we will spend just a moment to create a simple Win32 console application. We can make sure Visual Studio is installed and configured before we continue on to more complex activities.

1. Launch Visual Studio.
2. Click **New Project**.
3. Select **Visual C++**.
4. Select **Win32 Console Application**.



5. Name the application **ConsoleApp**.
6. Accept the Win32 Application Wizard default settings.



7. Click **Finish**.
8. Under **#include "stdafx.h"**, enter the following lines to reference the class and namespace:

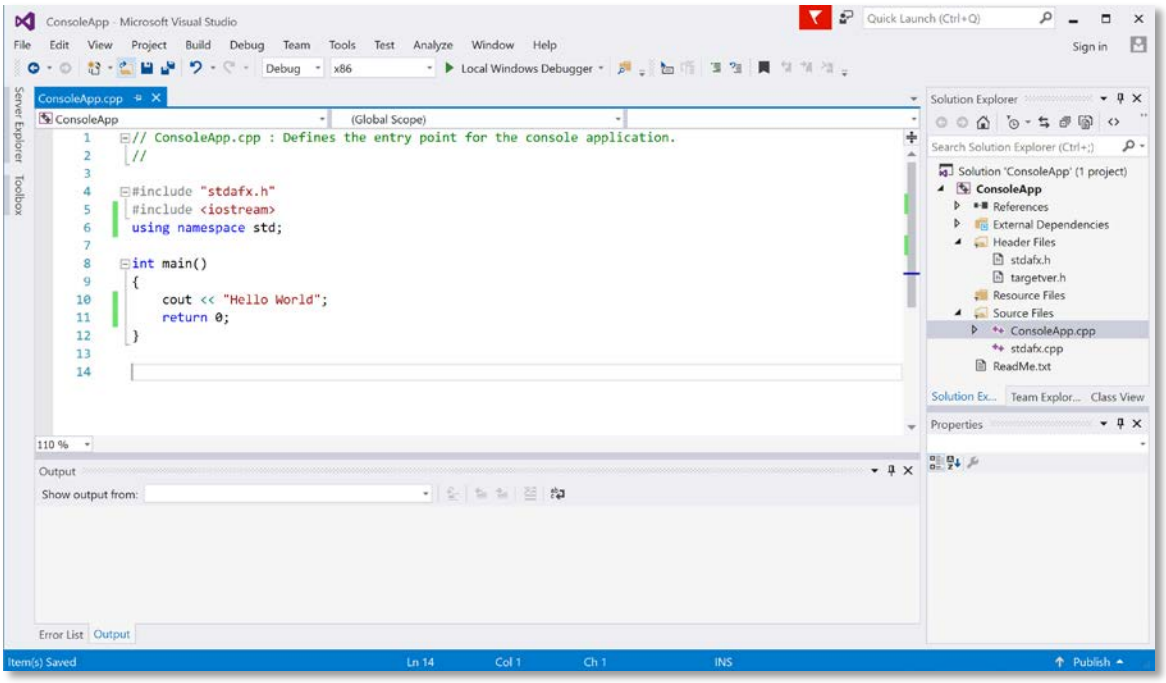
Enter Code

```
#include <iostream>
using namespace std;
```

9. Above **return 0;**, enter the following:

Enter Code

```
cout << "Hello World";
```

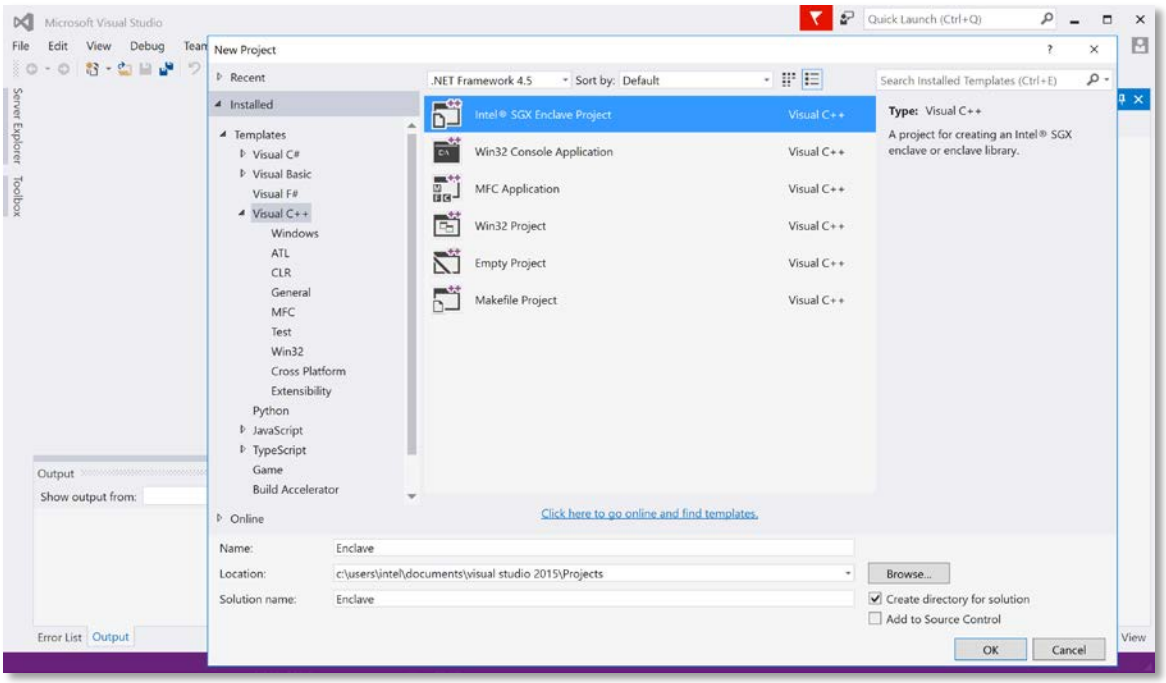


10. Save the application.

Task 2.2: Create an Enclave-Application Project

Lab Objective	In this lab, we will quickly demonstrate how to create an Intel SGX enclave project.

1. Launch Visual Studio, if it is not already running.
2. Click **File** > **New** > **Project**.
3. Select **Visual C++**.
4. Select **Intel® SGX Enclave Project**.



5. Name the application **Enclave**.
6. From the **Welcome to the Intel® SGX Enclave Project Wizard** screen, click **Next**.
7. Select **Enclave** as the project type, and then click **Finish**.
8. Save the project.

MORE INFO	When an enclave project is created for the first time, you have to choose either using an already existing signing key or having Visual Studio automatically generate a key for you. Currently, the evaluation SDK allows the developer to create and run enclaves using the Debug and Pre-release profiles. Enclaves compiled under the Release profile will not work until the developer completes the production licensing process. If you would like to deliver a production-quality application using Intel SGX, please contact the Intel SGX Program at sgx_program@intel.com for more information about a production license. After you receive a production license, use the provided key to sign the enclave. Follow this link for more information regarding signing key files.
DEVELOPER TIP	Avoid using spaces in enclave naming conventions; Intel SGX tools and functions do not support spaces in enclave names. **Yes**: my_enclave.edl **No**: my enclave.edl

Task 2.3: Add an Enclave Project to the Console-Application Project

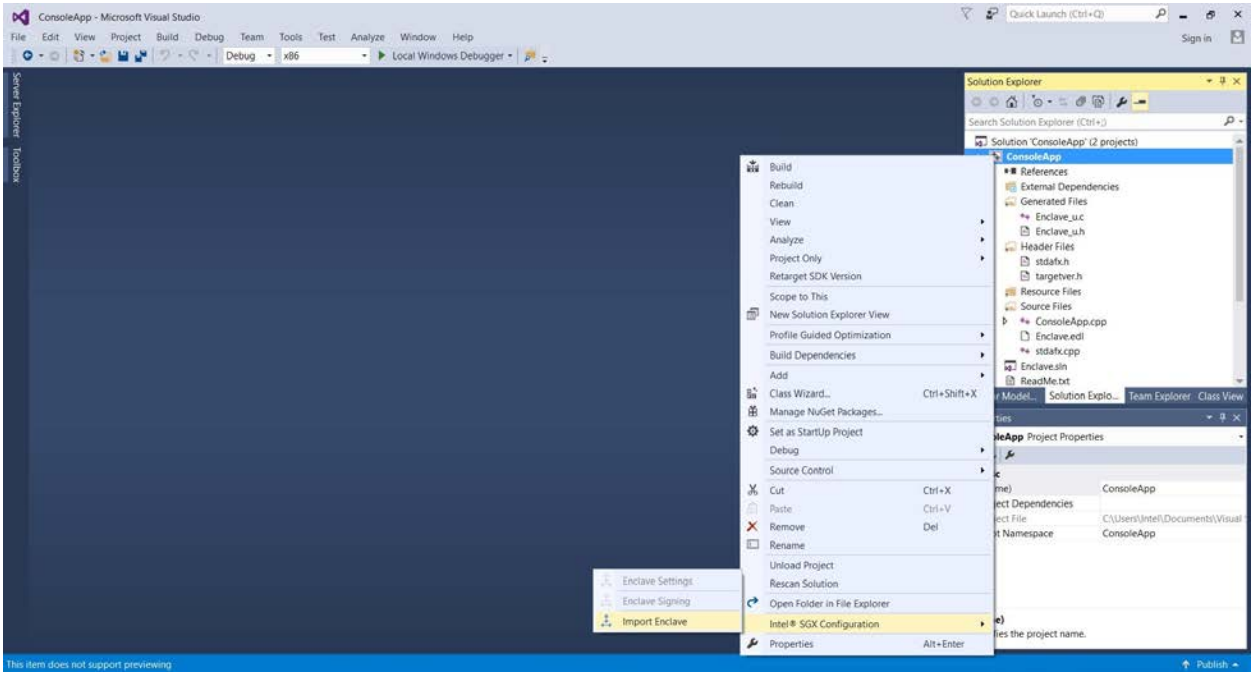
Lab Objective	In this lab, we will add multiple projects under the same solution, making our examples closer to real-world development projects.

1. Right-click the ConsoleApp solution.
2. Click **Add** > **Existing Project**.
3. Search for **Enclave.vcxproj**.
4. Click **Open**.
5. Now both the console application and enclave project are in the same solution file.
6. Click **Save**.

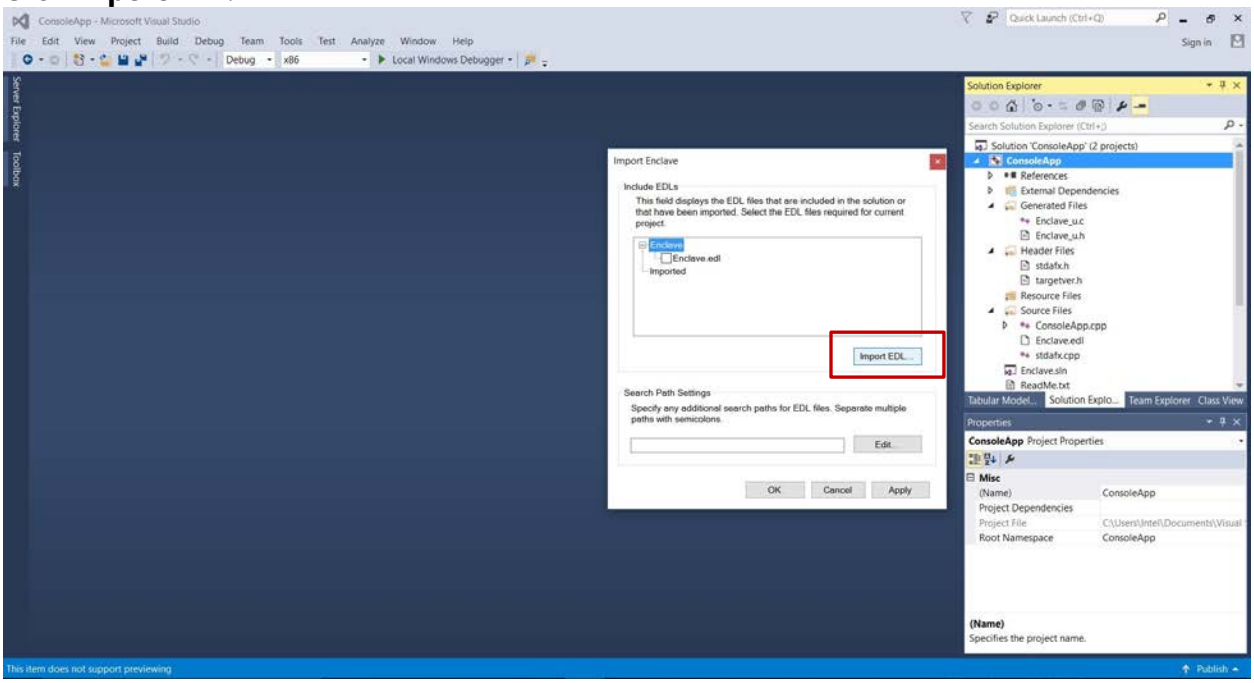
Task 2.4: Import an Enclave into the Console-Application Project

Lab Objective	In this lab, we will demonstrate how to import an enclave (.edl file) from one project to another.

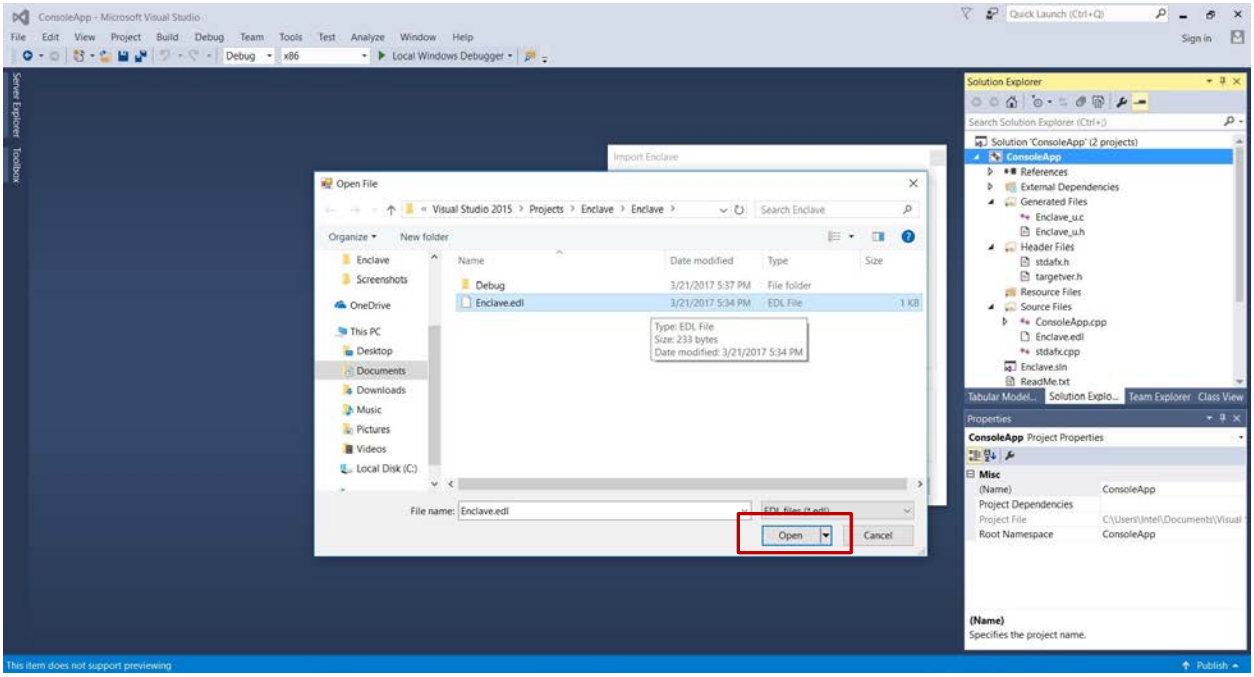
1. From Visual Studio, click **File** > **Open** > **Project/Solution**.
2. Open the **ConsoleApp** project.
3. Right-click the **ConsoleApp** application project.
4. Select **Intel® SGX Configuration**.
5. Select **Import Enclave**.



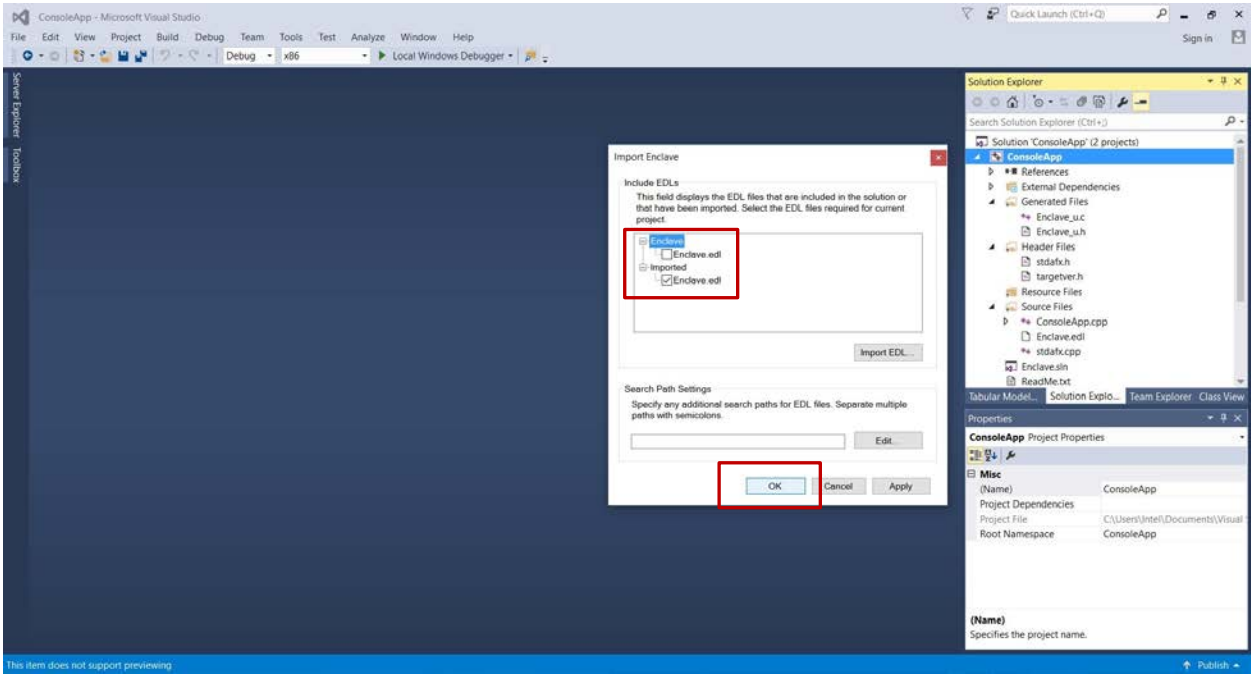
6. Click **Import EDL**.



7. Locate the **Enclave.edl** file in the EDL project, and then click **Open**.



8. Expand **Imported**, select **Enclave.edl**, and then click **OK**.

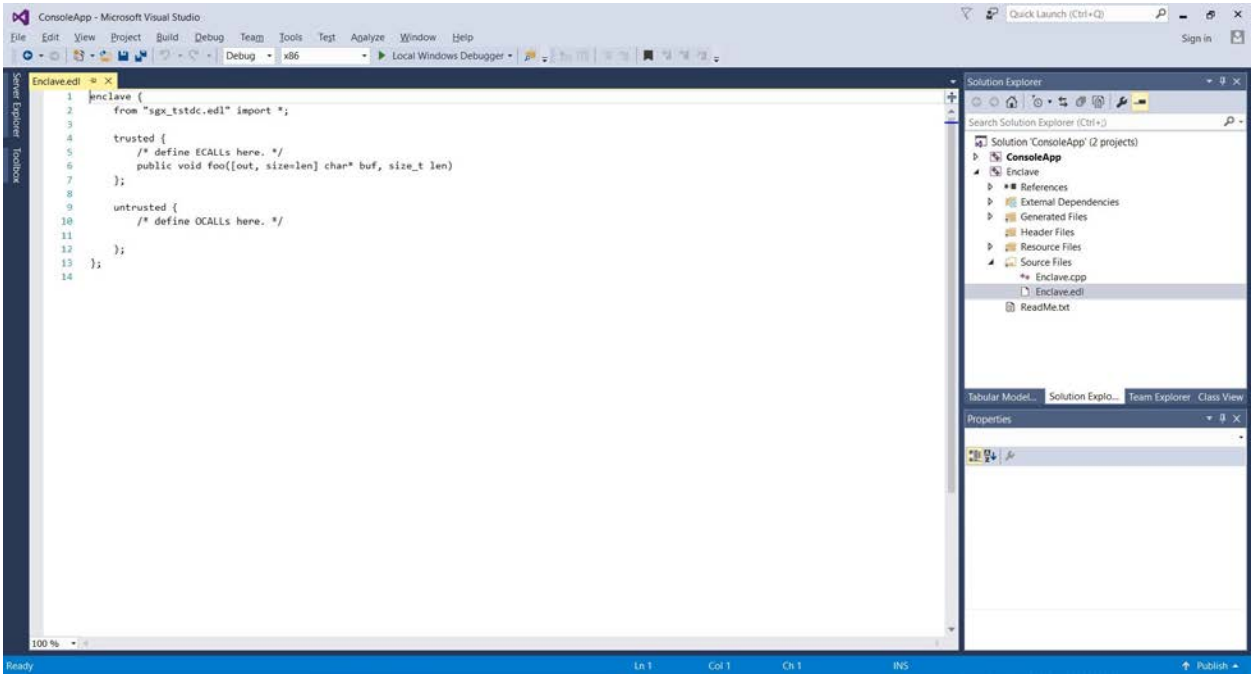


MORE INFO	Additional information regarding importing enclaves into your application can be found in this article.

Task 2.5: Add a Trusted Call to an Enclave Project

Lab Objective	In this lab, we will add a trusted call to our enclave project, compile the Enclave.edl, and review the auto-generated wrapper and header files.

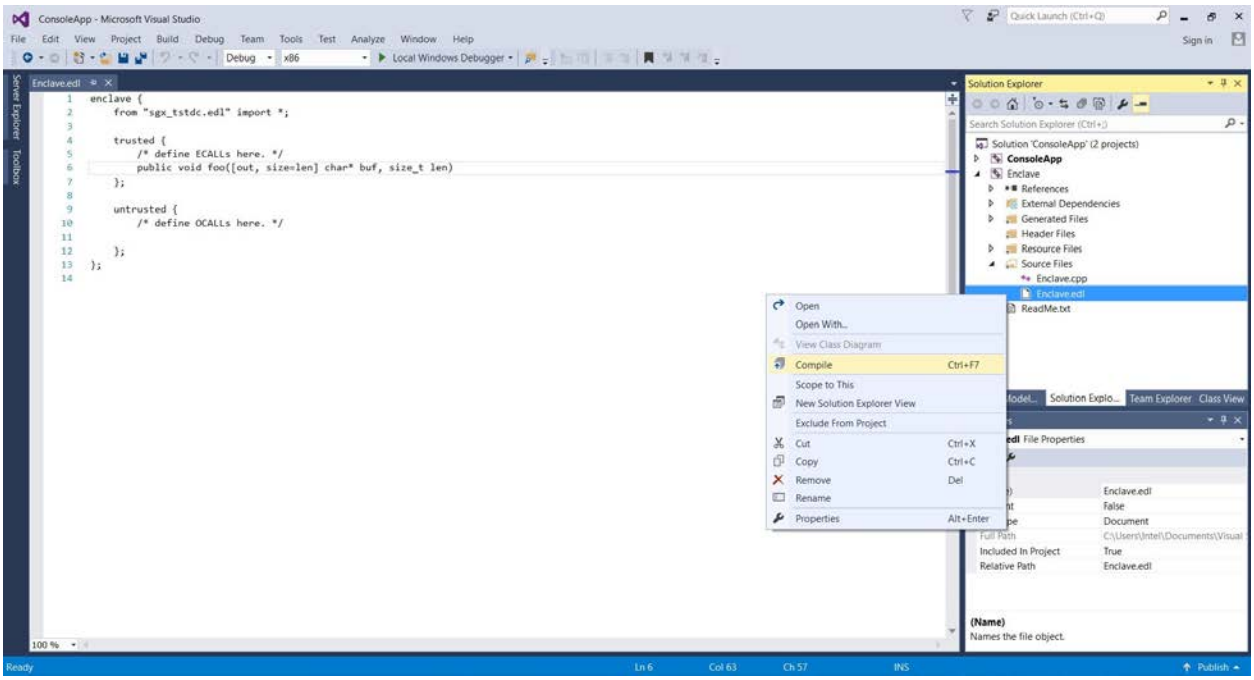
1. From the **Enclave** project, open **Enclave.edl**.



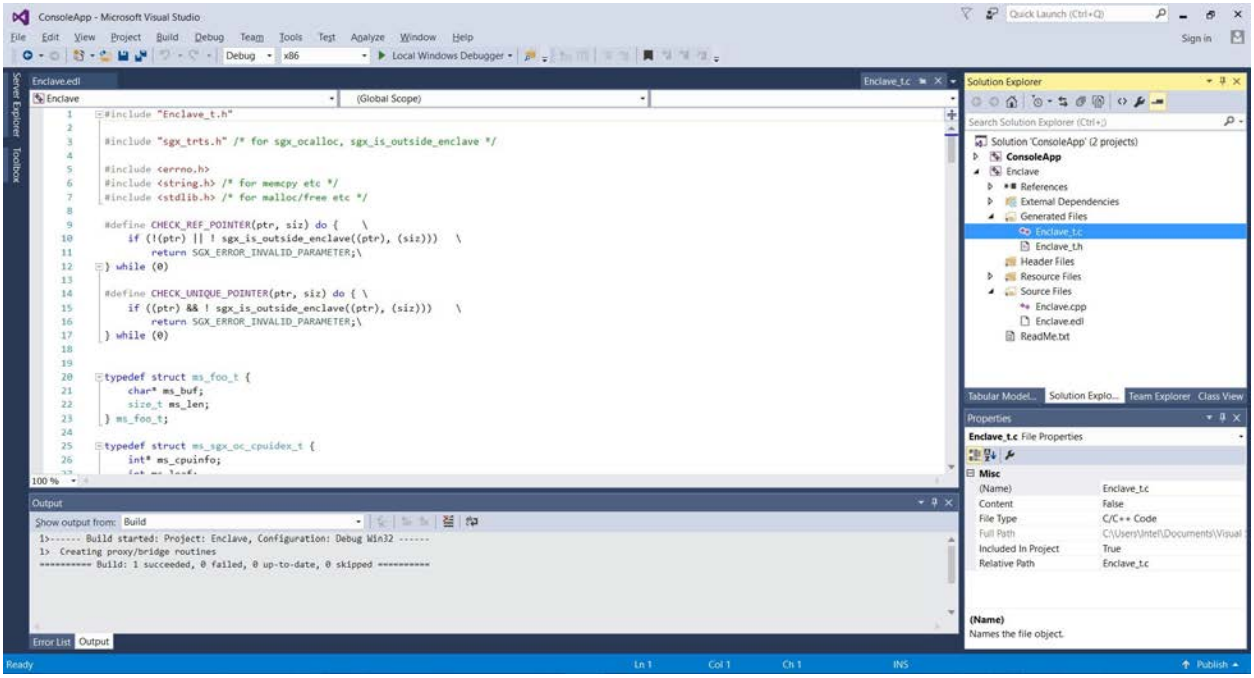
2. Enter the following sample trusted call:

Enter Code	`public void foo([out, size=len] char* buf, size_t len);`

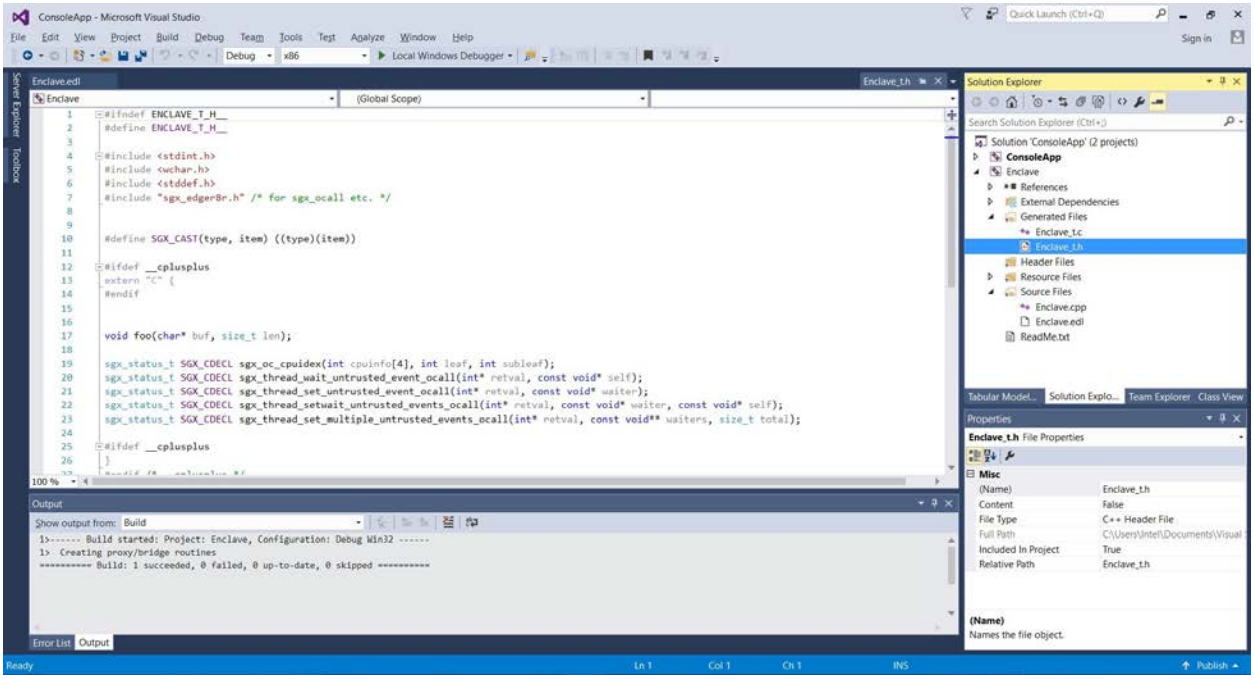
3. Right-click **Enclave.edl**, and then select **Compile**.



4. Go to **Generated Files > Enclave_t.c**. This is the *auto-generated wrapper function file*. Do not modify this file by hand because it is recreated with each build. This also contains the function definitions for the trusted proxies and bridges.



5. Select **Enclave_t.h**. This is an *auto-generated header file* for the wrapper functions. This also contains the prototype declarations for the trusted proxies and bridges.



6. Save the enclave project.

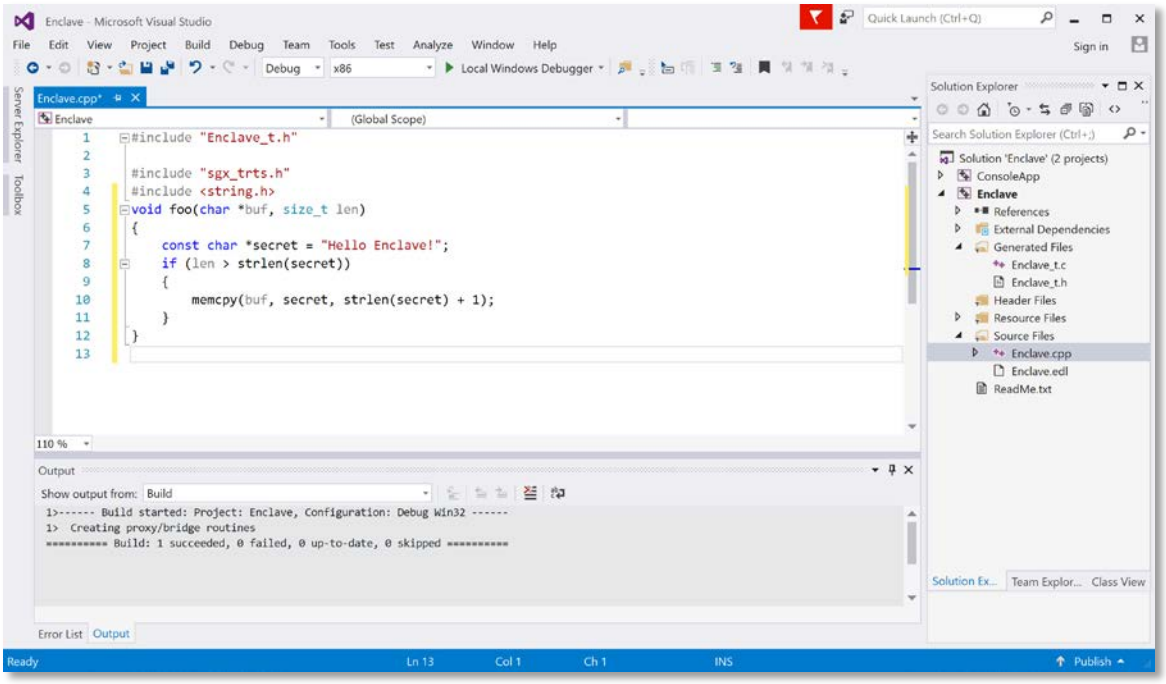
Task 2.6: Implement Enclave Functions

Lab Objective	In this lab, we will create an enclave and, if successful, provision a secret to the enclave.

1. Open **Enclave.cpp**.
2. Add the following function code to call the trusted ECALL:

Enter Code

```
#include <string.h>
void foo(char *buf, size_t len)
{
const char *secret = "Hello Enclave!";
if (len > strlen(secret))
      {
      memcpy(buf, secret, strlen(secret) + 1);
      }
}
```



3. Expand **ConsoleApp** > **Source Files**.
4. Open **ConsoleApp.cpp**.
5. Add the following *before* the main() function:

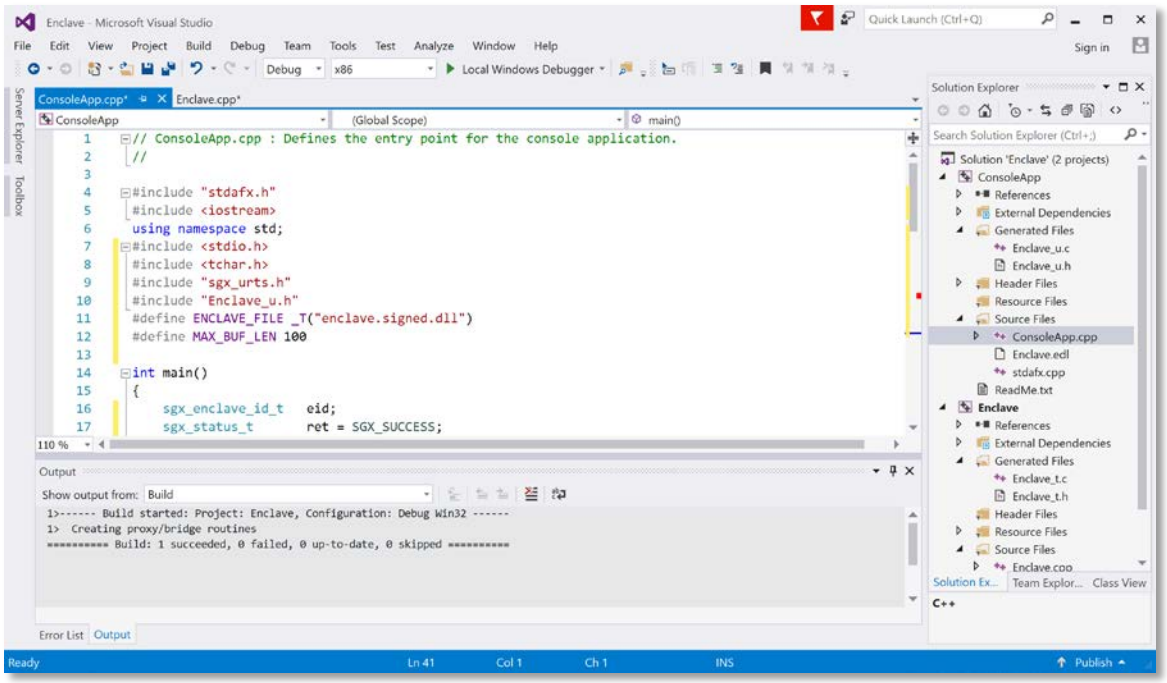
Enter Code

```
#include <stdio.h>
#include <tchar.h>
#include "sgx_urts.h"
#include "Enclave_u.h"
#define ENCLAVE_FILE _T("enclave.signed.dll")
#define MAX_BUF_LEN 100
```

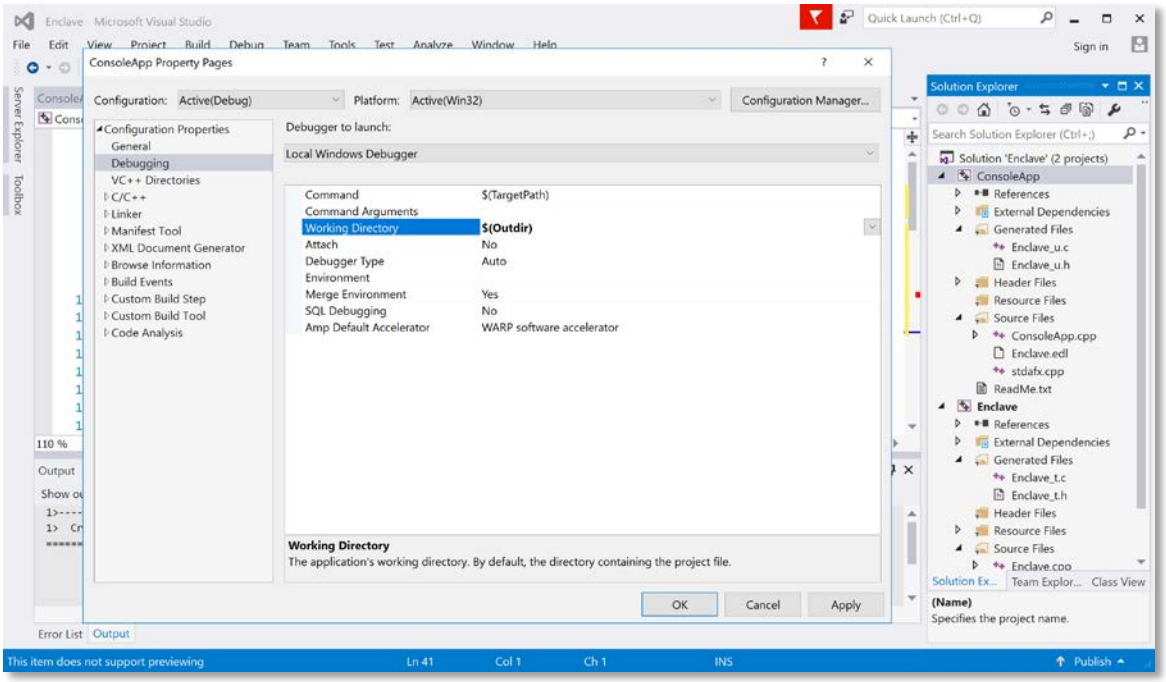
6. Add the following *in* the main() function:

Enter Code

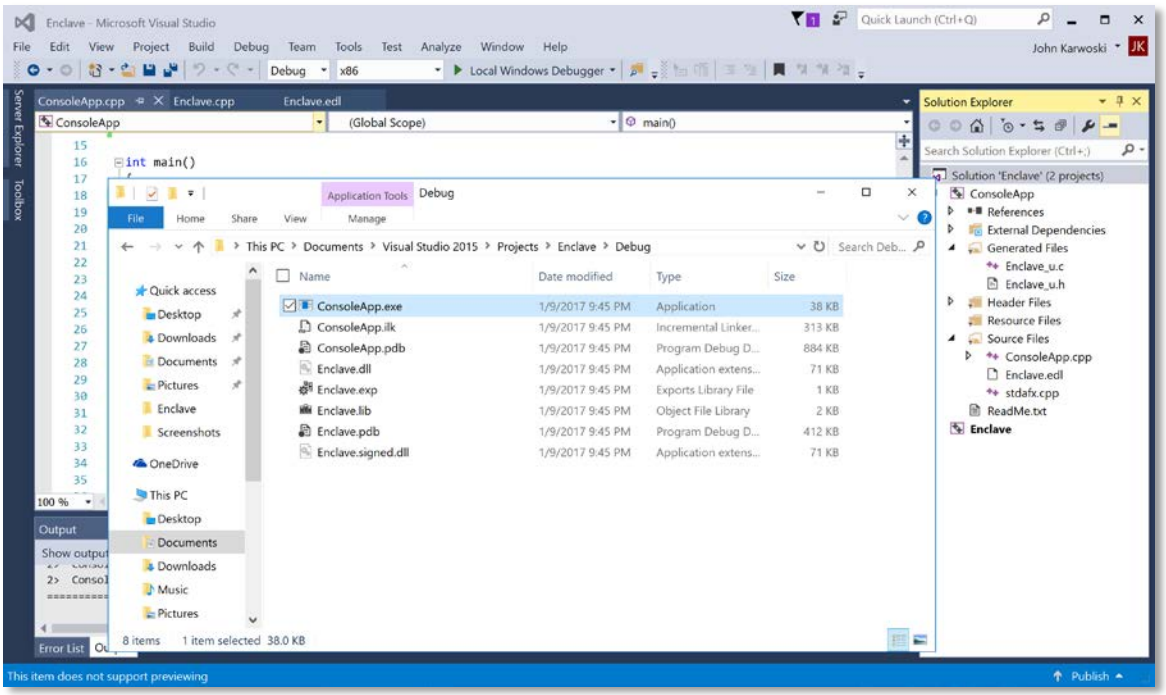
```
sgx_enclave_id_t   eid;
sgx_status_t       ret   = SGX_SUCCESS;
sgx_launch_token_t token = {0};
int updated = 0;
char buffer[MAX_BUF_LEN] = "Hello World!";

// Create the Enclave with above launch token.
ret = sgx_create_enclave(ENCLAVE_FILE, SGX_DEBUG_FLAG,
&token, &updated,
                         &eid, NULL);
if (ret != SGX_SUCCESS) {
printf("App: error %#x, failed to create enclave.\n",
ret);
return -1;
}


// A bunch of Enclave calls (ECALL) will happen here.
foo(eid, buffer, MAX_BUF_LEN);
printf("%s", buffer);

// Destroy the enclave when all Enclave calls finished.
if(SGX_SUCCESS != sgx_destroy_enclave(eid))
return -1;
cin.get();

return 0;
```



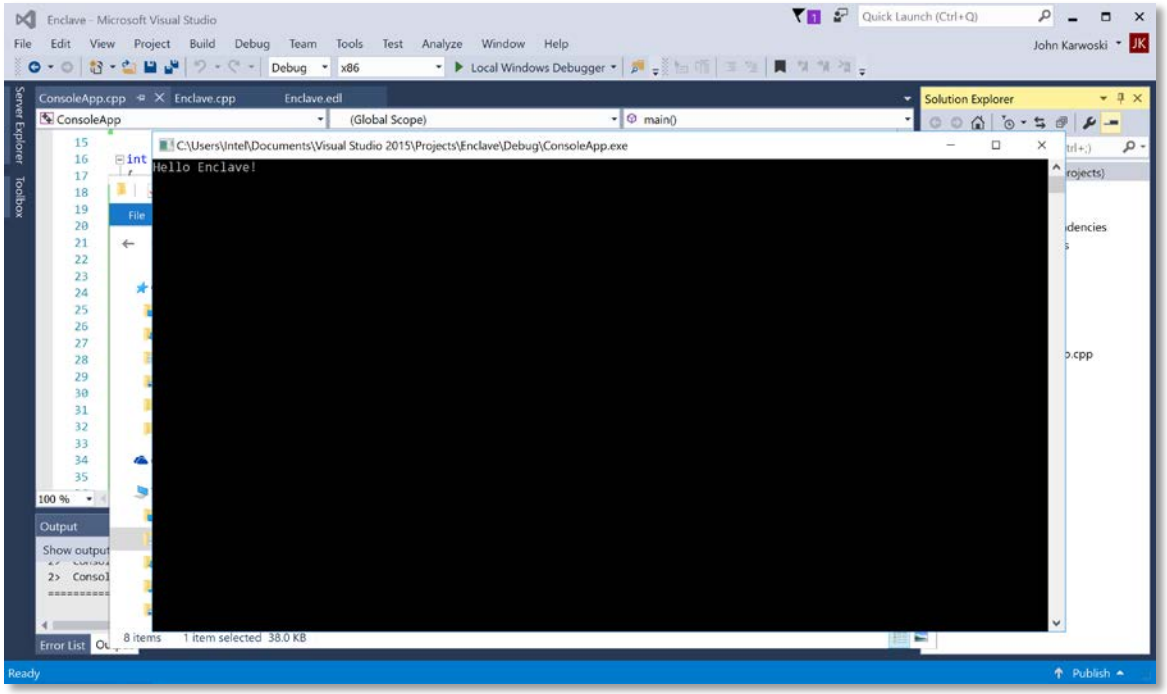
7. Change the working directory for the debugger:
   a. Right-click **ConsoleApp**.
   b. Select **Properties**.
   c. Select **Debugging**.
   d. Change the **Working Directory** path to **$(Outdir)**.



8. Complete step 7 for the enclave application.
9. Compile the console application.
10. Once built successfully, browse to the compiled executable.



11. Run the executable and verify the result.



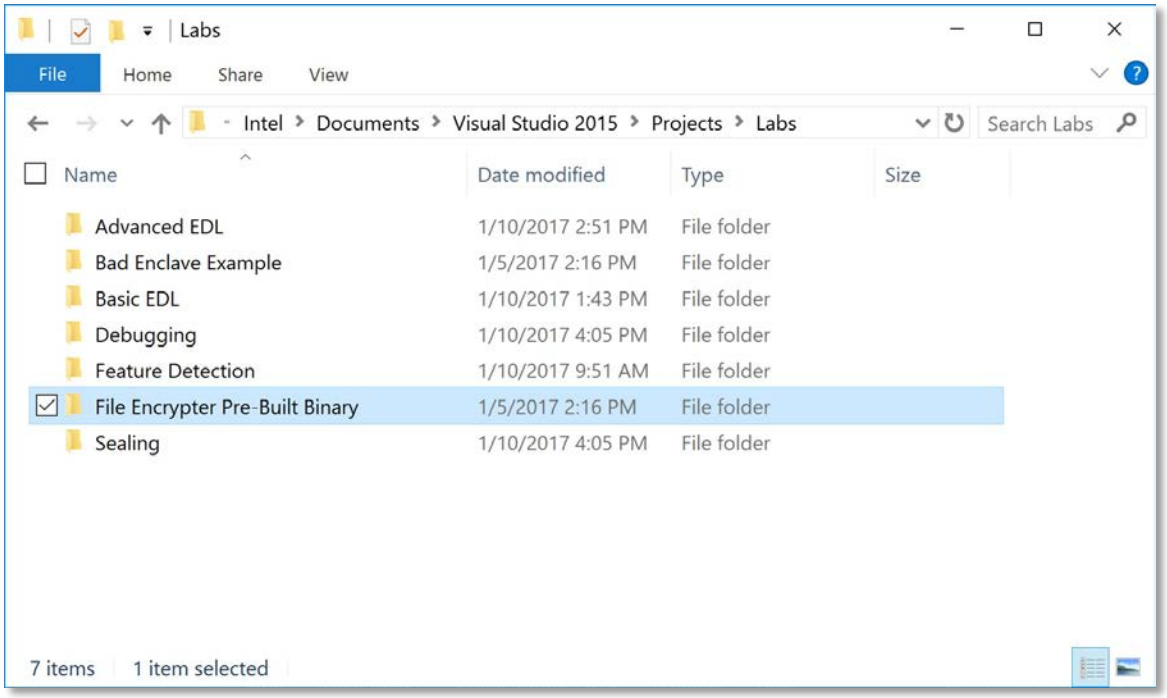
Result The application should be fully functional at this point.

Task 3: Review of Enclave Components

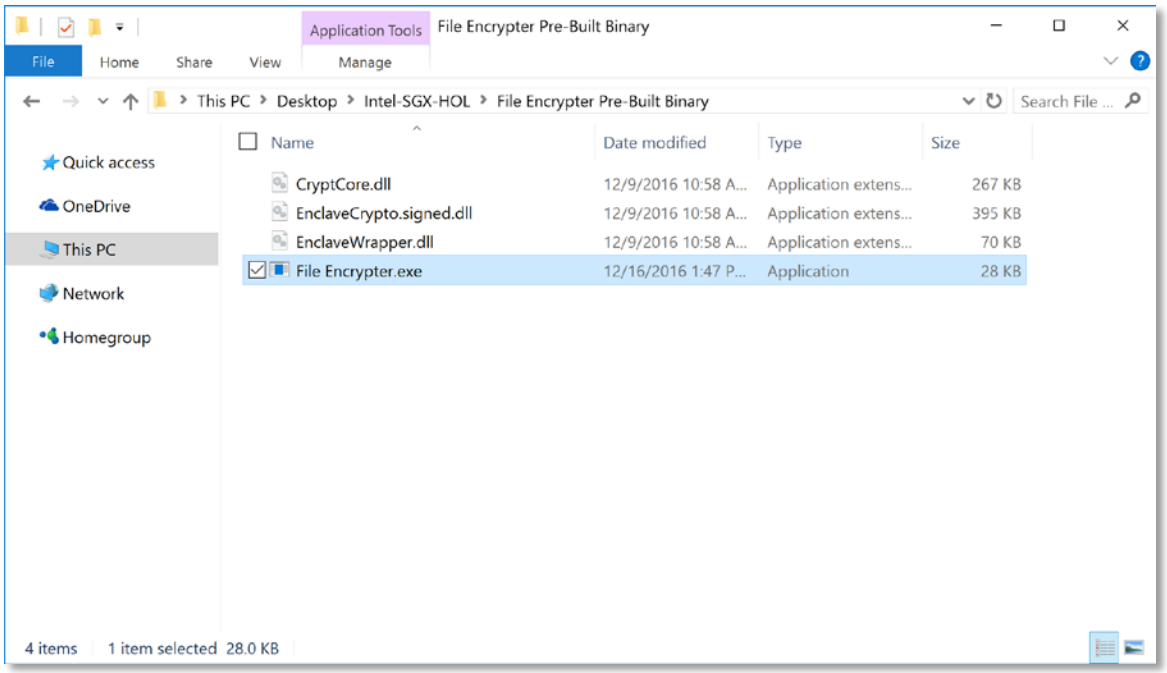
Task 3.1: Run the File-Encrypter Application

Lab Objective	In this lab, we will run the file-encrypter application to review the final product of the application that will be created during the course of this hands-on lab.

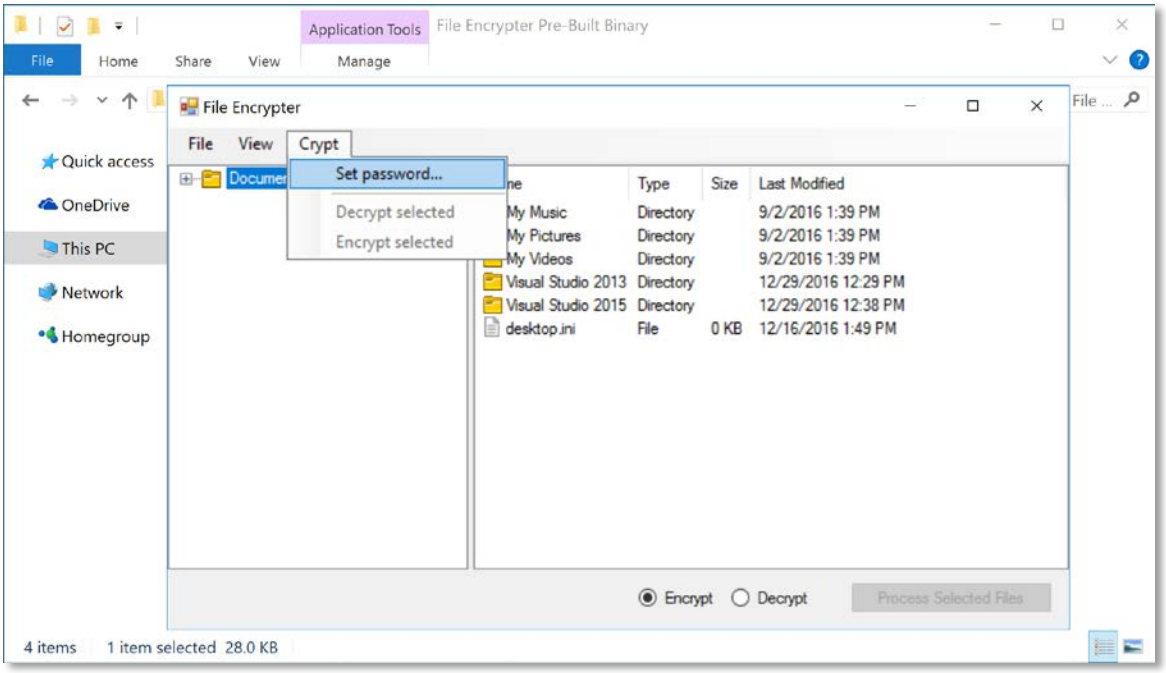
1. Browse to the **Intel-SGX-HOL > File Encrypter Pre-Built Binary** folder.



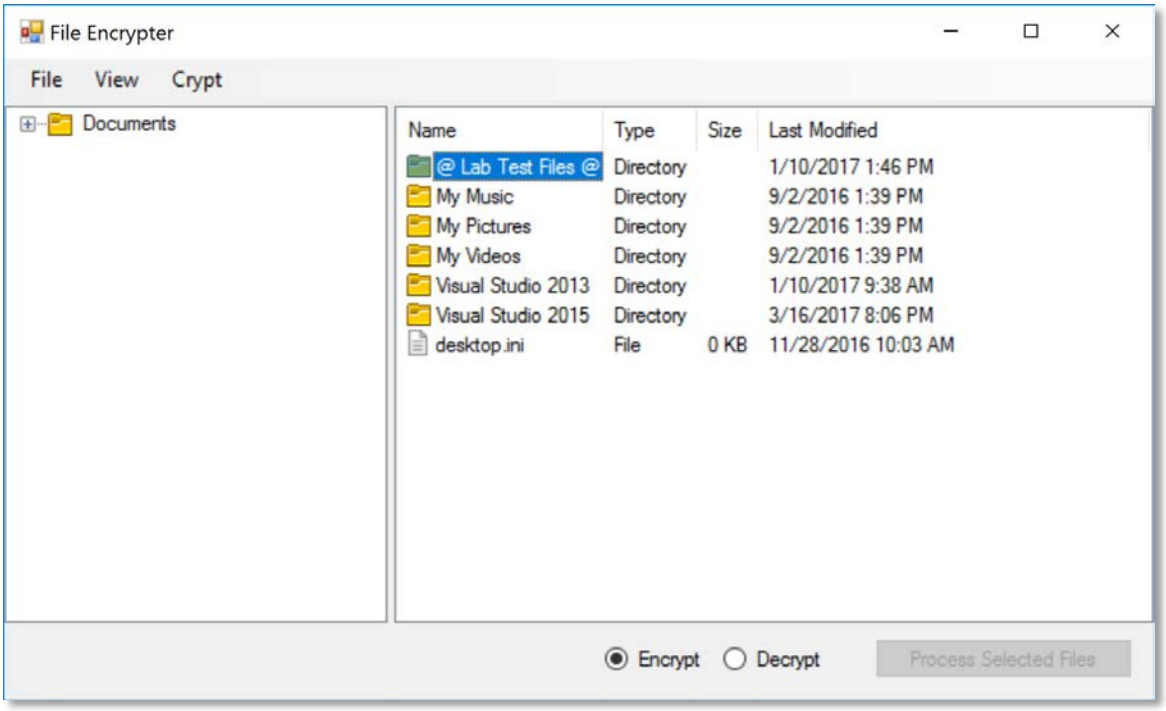
2. Run the file-encrypter application.



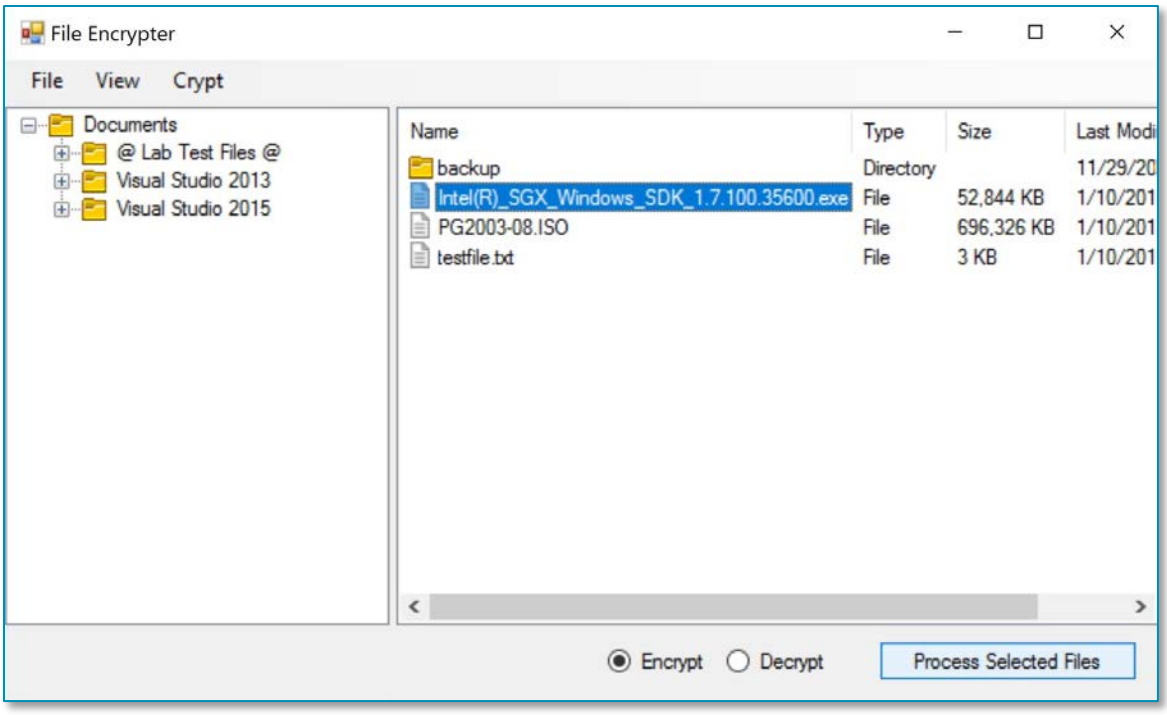
3. Click **Crypt** > **Set password**.



4. Enter a password.
5. In the directory pane, select the **@ Lab Test Files @** folder.



6. Select the file **Intel(R)_SGX_Windows_SDK_1.7.100.35600.exe**.

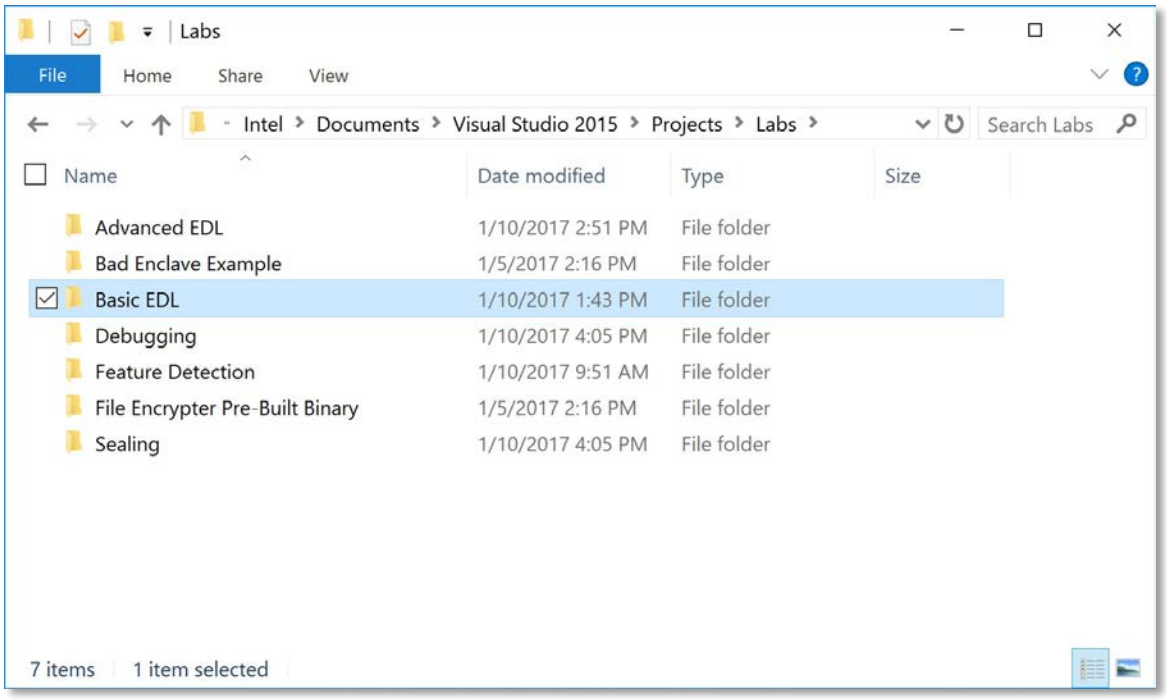


7. Click **Process Selected Files**.
8. Click **Crypt** > **Set password**.
9. Enter an incorrect password.
10. Select an encrypted file.
11. Click **Decrypt**.
12. Click **Process Selected Files**.
13. Run through the application a few more times to get familiar with the process.

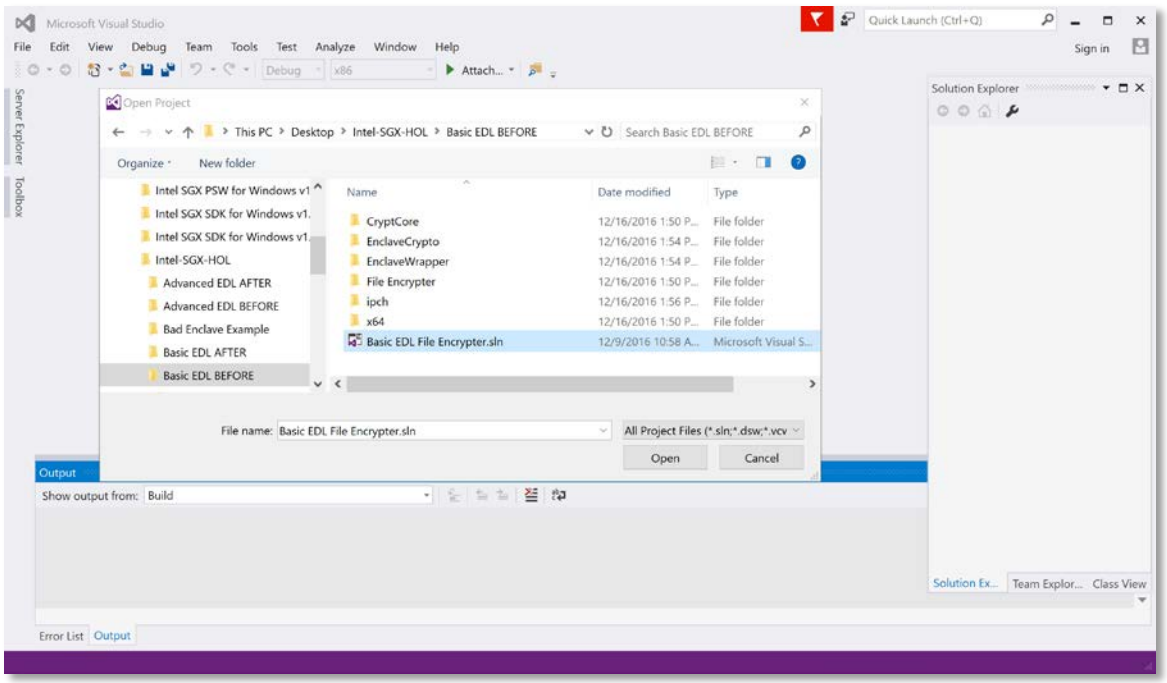
Task 3.2: Review the File-Encrypter Project

Lab Objective	In this lab, before we begin to make changes to the file-encrypter project, we will review the four projects found in the solution, giving us an understanding and foundation to build on.

1. Launch Visual Studio.
2. Browse to **Intel-SGX-HOL** > **Basic EDL**.



3. Open **Basic EDL File Encrypter.sln**.



4. This solution contains four projects:
    a. **File Encrypter**: This is the C# graphical user interface (GUI) of the file-encrypter application. The file-encrypter application talks directly to the CryptCore DLL.
    b. **CryptCore**: This contains two modules, CryptCore and CryptCoreNative. These two modules make up the CryptCore DLL and they are responsible for marshalling code between .NET and native C/C++. The CryptCore DLL talks directly to the EnclaveWrapper DLL. This module is a mix of managed and unmanaged code:
        i. **CryptCore**: This is managed C++ code.
        ii. **CryptCoreNative**: This is managed and native C++ code.
    c. **EnclaveWrapper**: This is 100-percent native C code. The EnclaveWrapper DLL talks to the enclave, EnclaveCrypto.
    d. **EnclaveCrypto**: This is 100-percent native C++ code. This is the enclave.

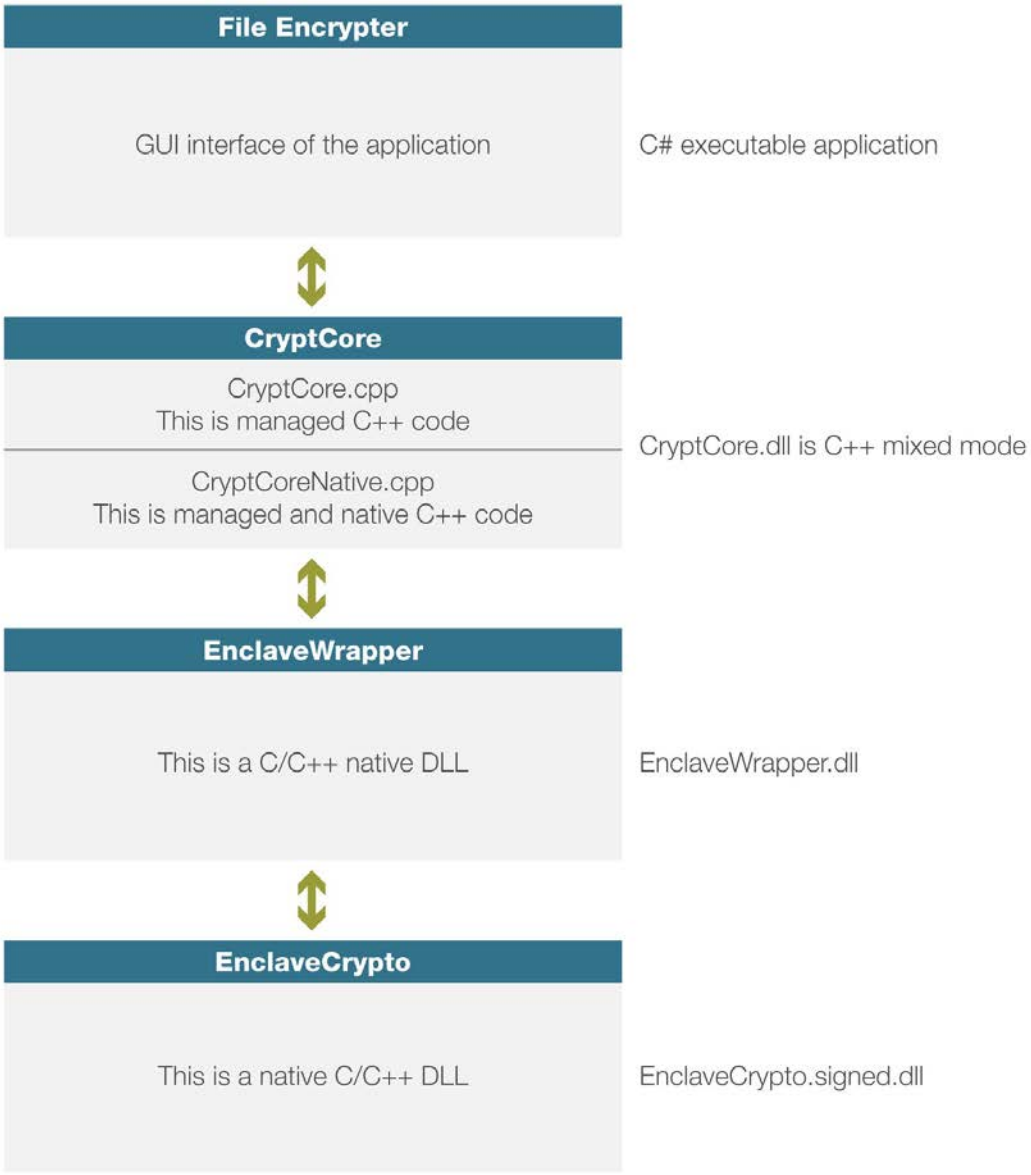
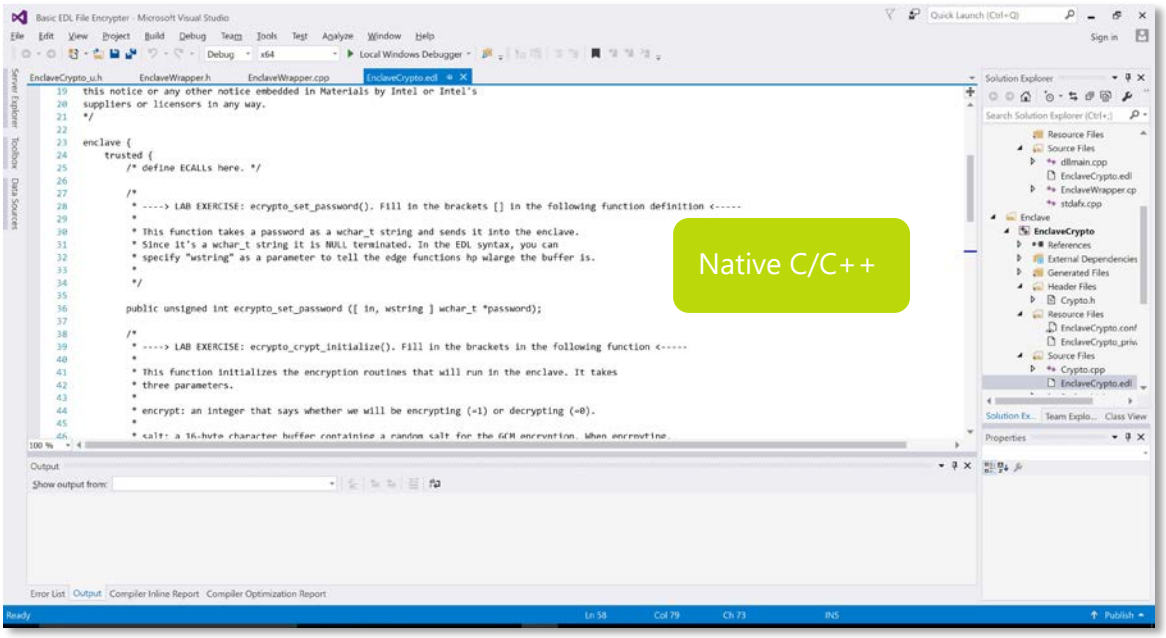


Figure 1 Relationship of the File Encrypter, CryptCore, EnclaveWrapper, EnclaveCrypto projects

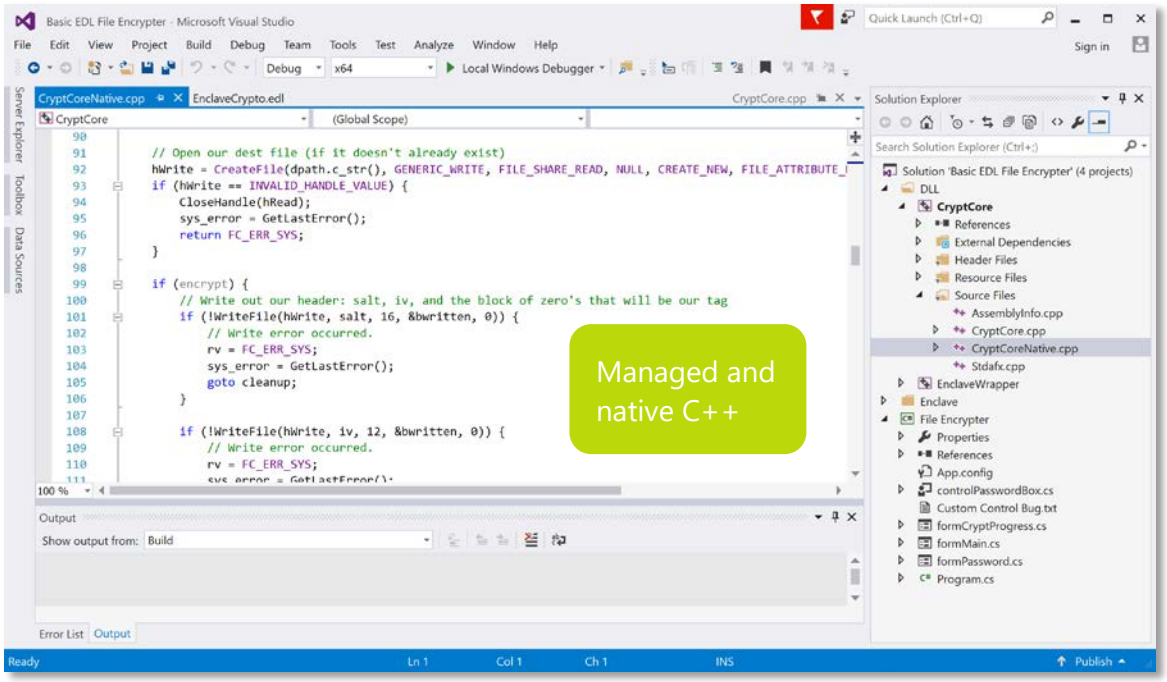
Task 3.3: Walkthrough of the Enclave Project's Components

Lab Objective	In this lab, we will further explore the enclave interfaces and wrapper functions.

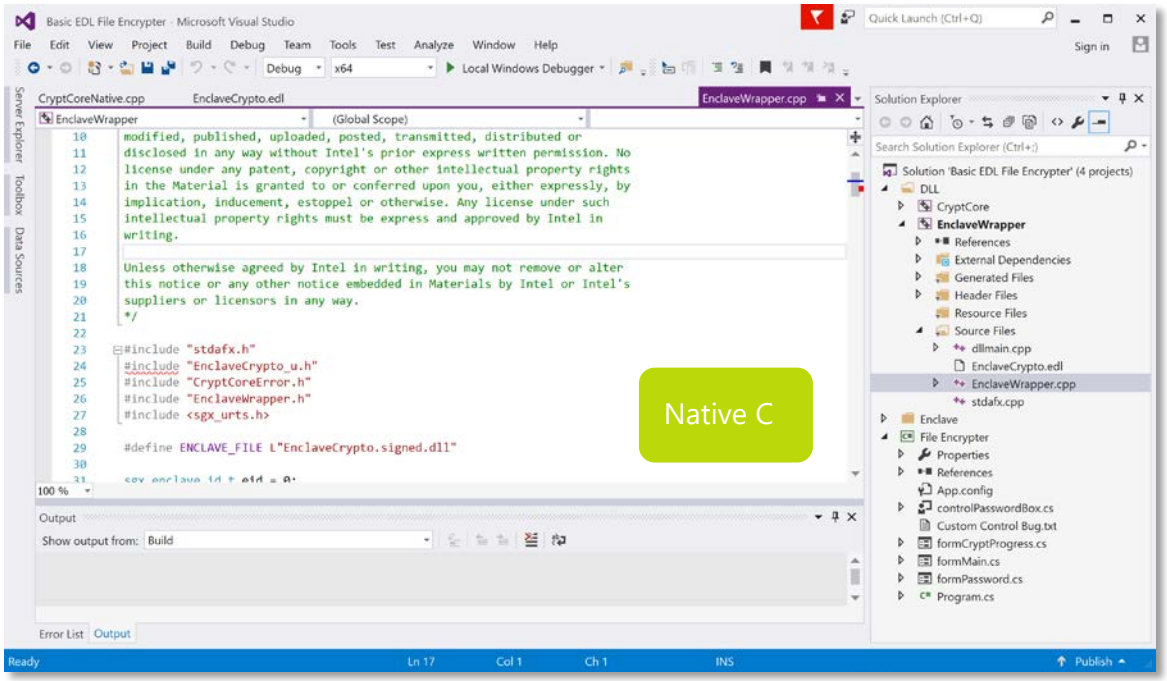
1. Return to Visual Studio.
2. Browse to **Intel-SGX-HOL** > **Basic EDL**.
3. Open **Basic EDL File Encrypter.sln**.
4. Expand **EnclaveCrypto** > **Source Files**.
5. Select **EnclaveCrypto.edl**. This file is used to set the trusted and untrusted zones of the enclave interface.



6. Expand **DLL** > **CryptCore** > **Source Files**.
7. Select **CryptCoreNative.cpp**. This file interfaces with the EnclaveWrapper functions and allows the transition of the managed code to the unmanaged native C++ code.



8. Expand **EnclaveWrapper** > **Source Files**.
9. Select **EnclaveWrapper.cpp**. This file creates the wrapper functions around the functions that will interface between the EnclaveCrypto and CryptCoreNative functions.



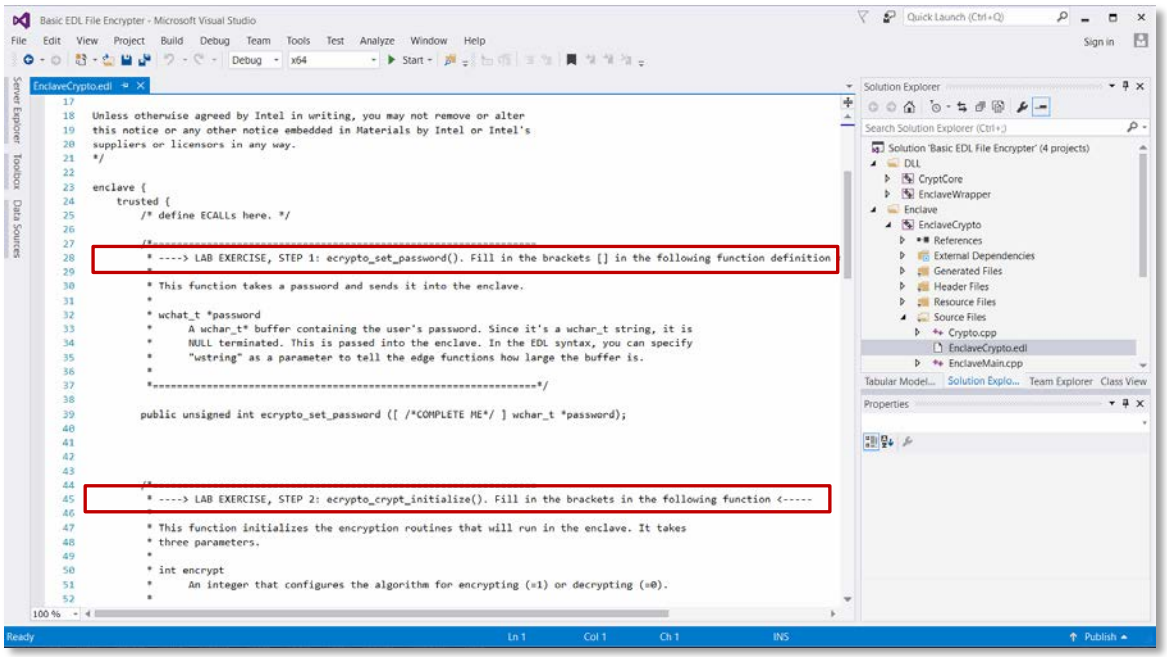
Task 4: Basic Enclave-Definition Language (EDL) and ECALL

Task 4.1: File-Encrypter Project EDL

Lab Objective	In this lab, we will create the EDL trusted calls for the file-encrypter project.

1. Return to Visual Studio.
2. Browse to **Intel-SGX-HOL** > **Basic EDL**.
3. Open **Basic EDL File Encrypter.sln**.
4. Expand **EnclaveCrypto** > **Source Files**.
5. Select **EnclaveCrypto.edl**.
6. Read through the information provided under each "lab exercise" heading to complete the four existing exercises. Each exercise contains information needed to fill out the required attributes.

Complete Functions	`ecrypto_set_password()`
	`ecrypto_crypt_initialize()`
	`crypto_crypt_block()`
	`ecrypto_crypt_finish()`



7. Save changes when complete.
8. Right-click **EnclaveCrypto.edl**, and then select **Compile**. If there are no errors, the EDL file has compiled successfully.
9. Navigate to **Generated files**.
10. Review **EnclaveCrypto_t.c**.
11. Review **EnclaveCrypto_t.h**.
12. Leave the project open for the next exercise.

Task 4.2: File-Encrypter Project EnclaveWrapper ECALL Functions

Lab Objective	In this lab, we will add ECALLs to several functions to EnclaveWrapper.cpp using commented instructions found in the file.

1. Expand **EnclaveWrapper** > **Source Files**.
2. Select **EnclaveWrapper.cpp**.
3. EnclaveWrapper.cpp contains five lab exercises. Complete each exercise using the information provided in the solution file.

Add ECALLs	`Add ECALLs to the following functions;`
	`ecrypto_set_password()`
	`ecrypto_crypt_initialize()`
	`crypto_crypt_block()`
	`ecrypto_crypt_finish()`
	`ecrypto_crypt_close()`

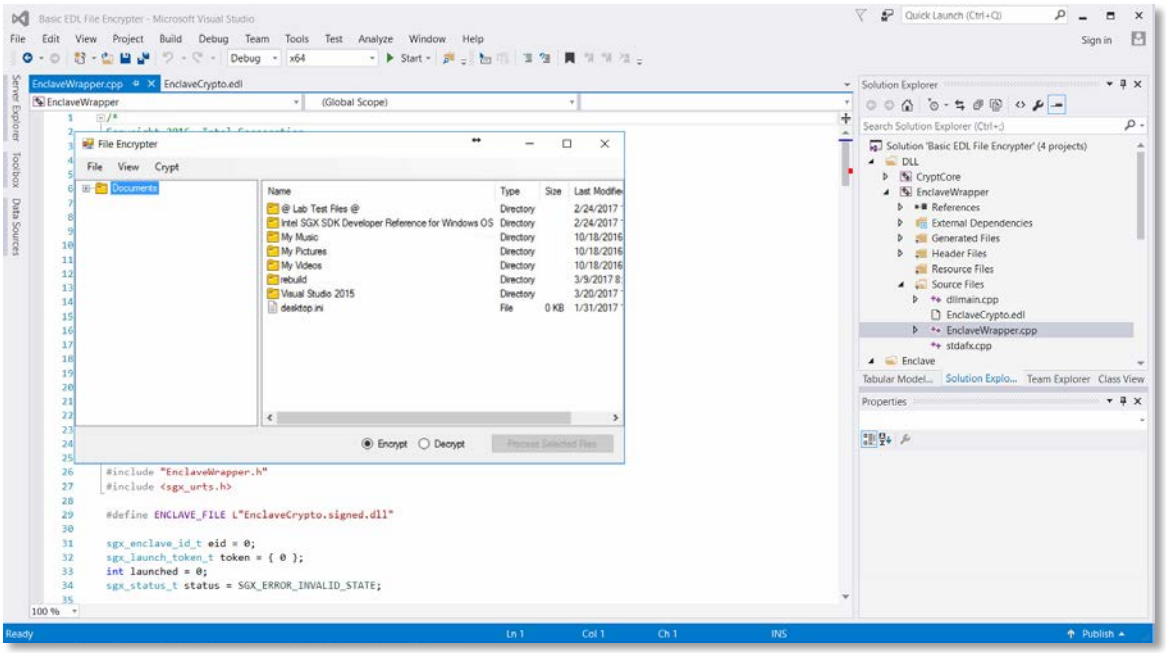
4. Save changes when completed.

Task 4.3: Compile Basic File-Encrypter Application

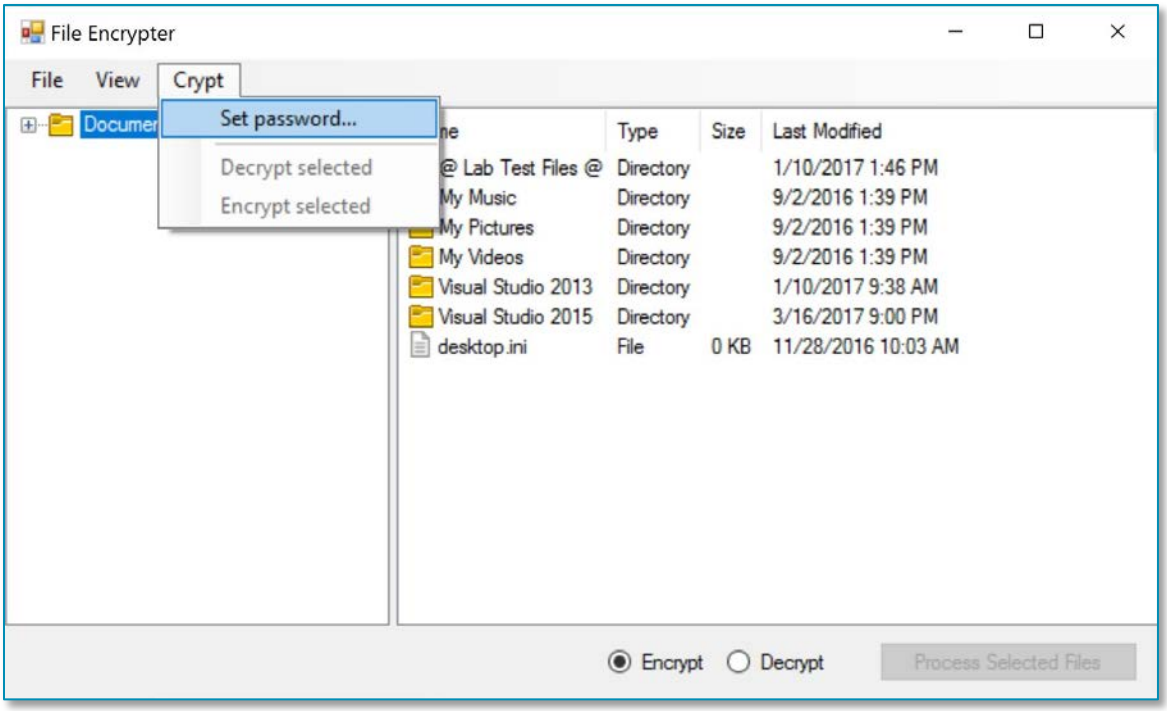
Lab Objective	In this lab, we will compile and test the file-encrypter application.

1. Compile the project. If the project compiles successfully, the exercises have been completed successfully.

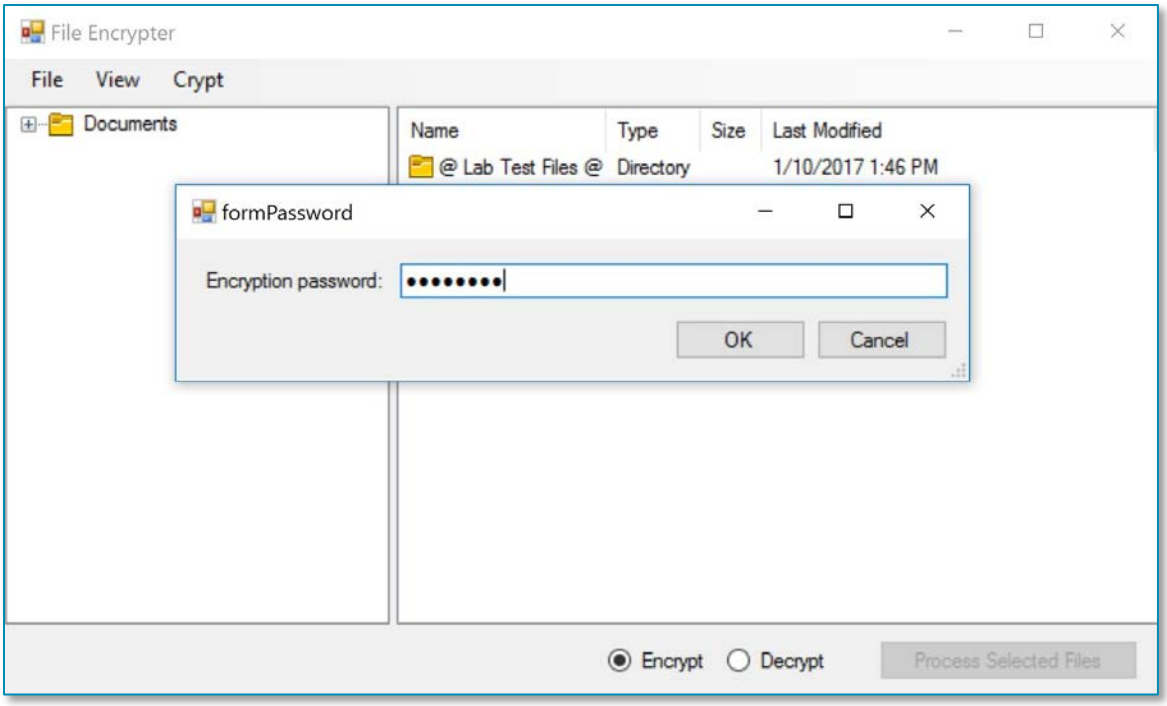
   Build the application to show how encrypting and decrypting a file works.



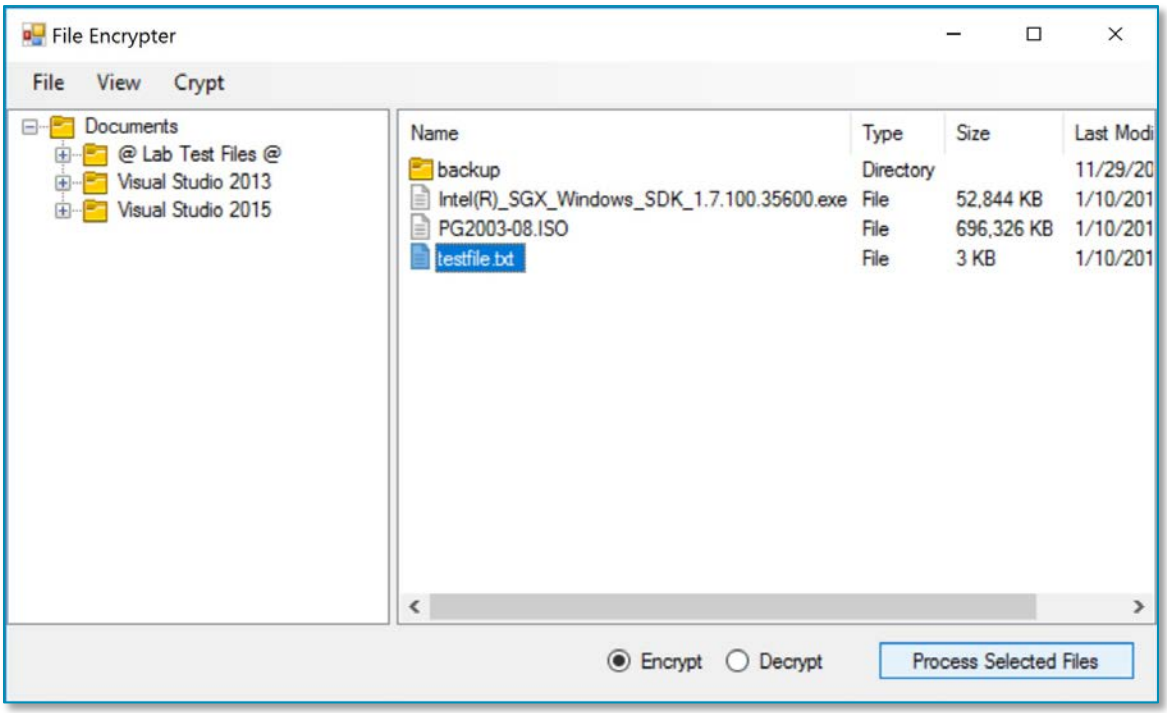
2. Select **Crypt** > **Set Password**.



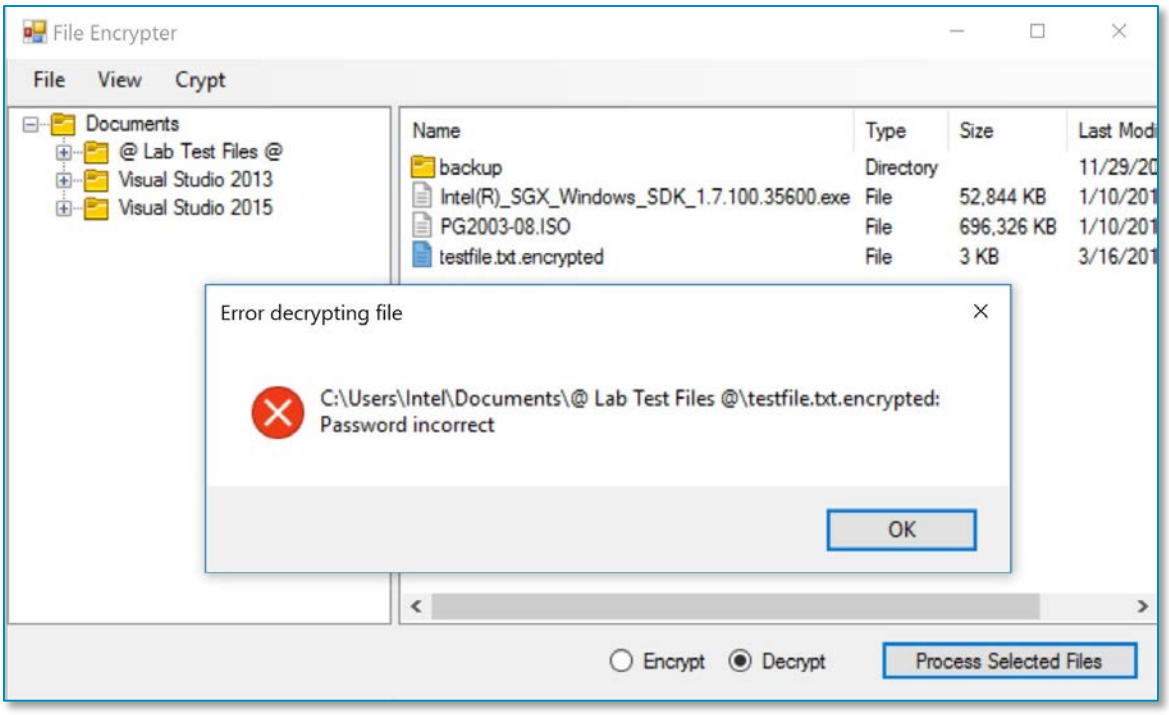
3. Enter a password, and then click **OK**.



4. Select a test file.
5. Click **Encrypt**, and then click **Process Selected Files**.



6. The file is now encrypted.
7. Click **Crypt** > **Password**, and then enter an incorrect password.
8. Select the encrypted file, click **Decrypt**, and then click **Process Selected Files**.

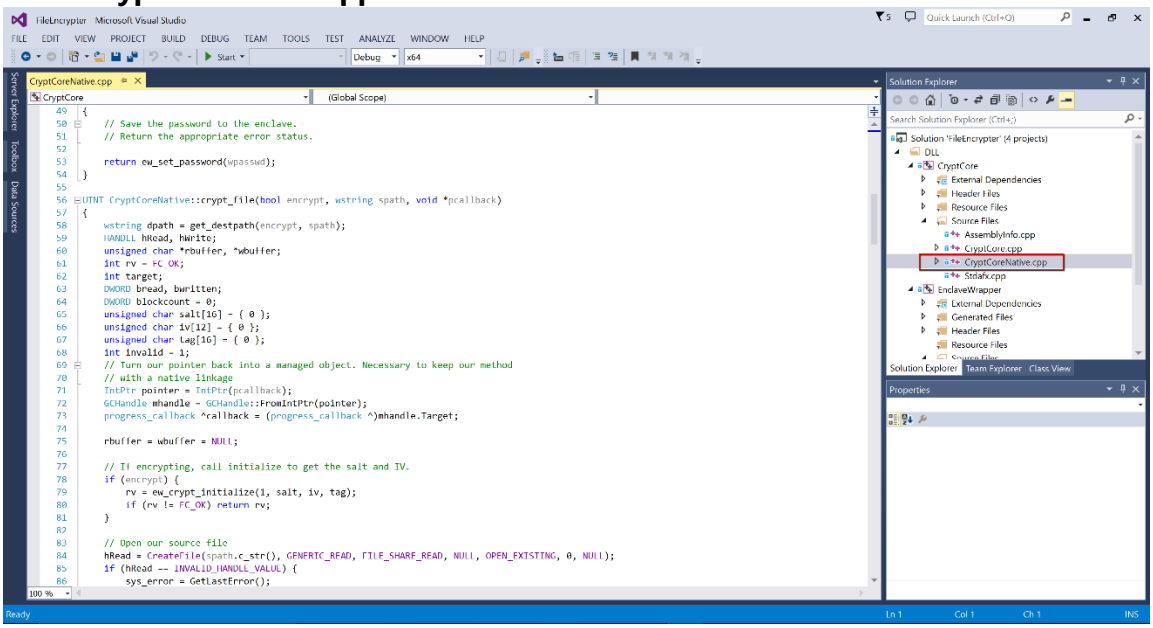


9. Run through the process a couple more times with the other files provided to view what the outcome is.

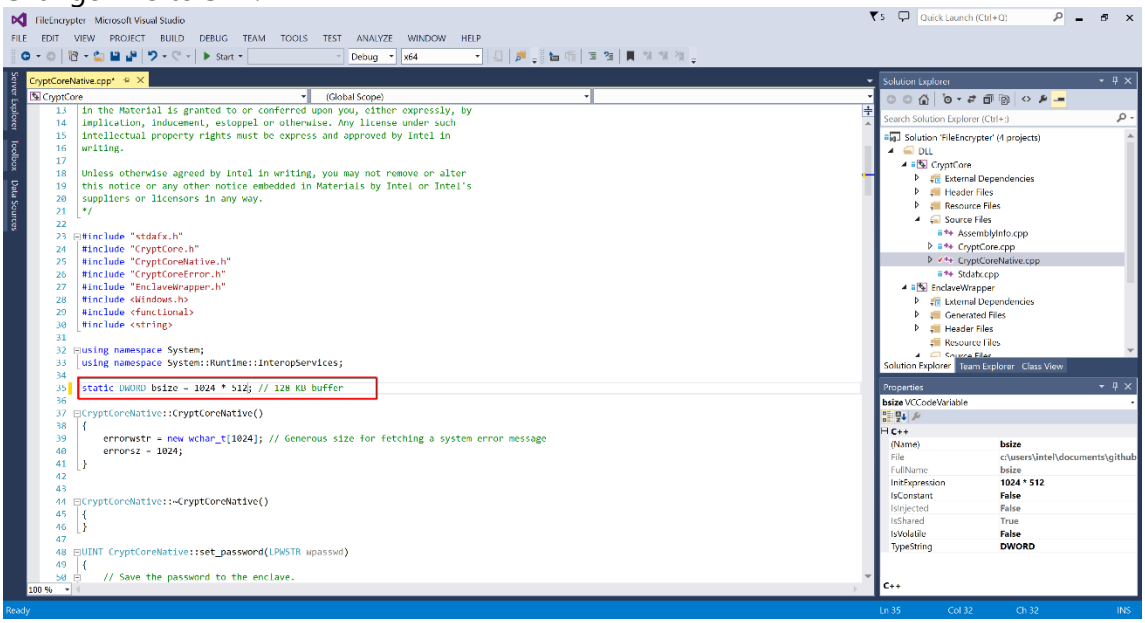
Task 4.4: Review Enclave Configuration Advanced Lab

Lab Objective	In this lab, we will explore and modify buffer allocation used to encrypt and decrypt files.

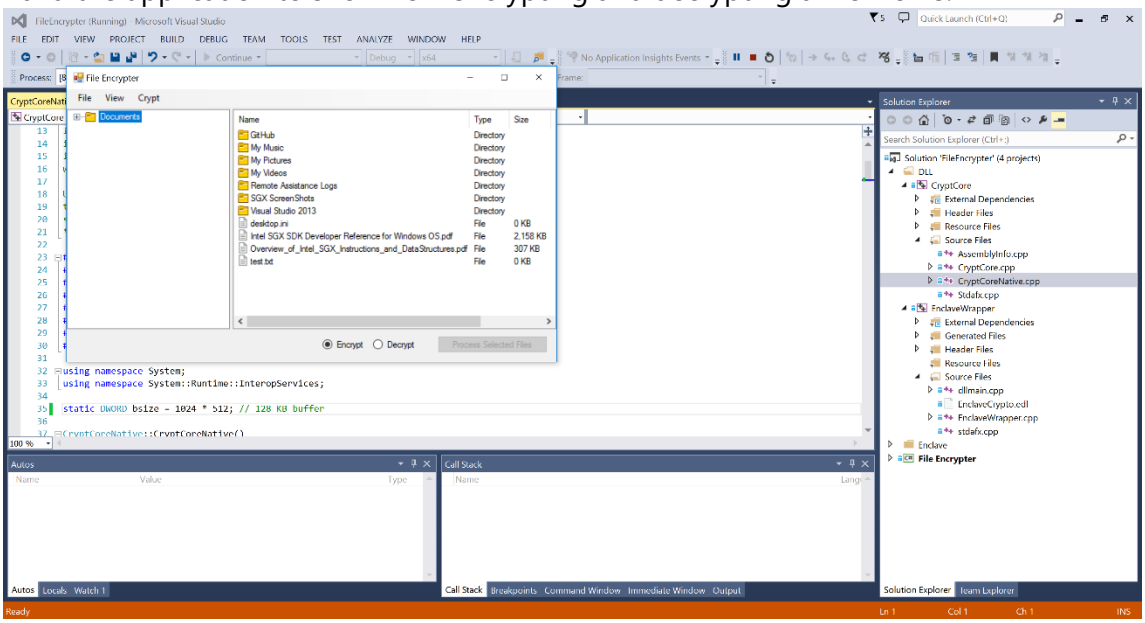
1. Launch **Visual Studio 2015**.
2. Click **Open Project**.
3. Navigate to **Intel-SGX-HOL** > **Basic EDL**.
4. Select **FileEncrypter.sln**, and then click **Open**.
5. Expand **DLL** > **CryptCore** > **Source Files**.
6. Select **CryptCoreNative.cpp**.



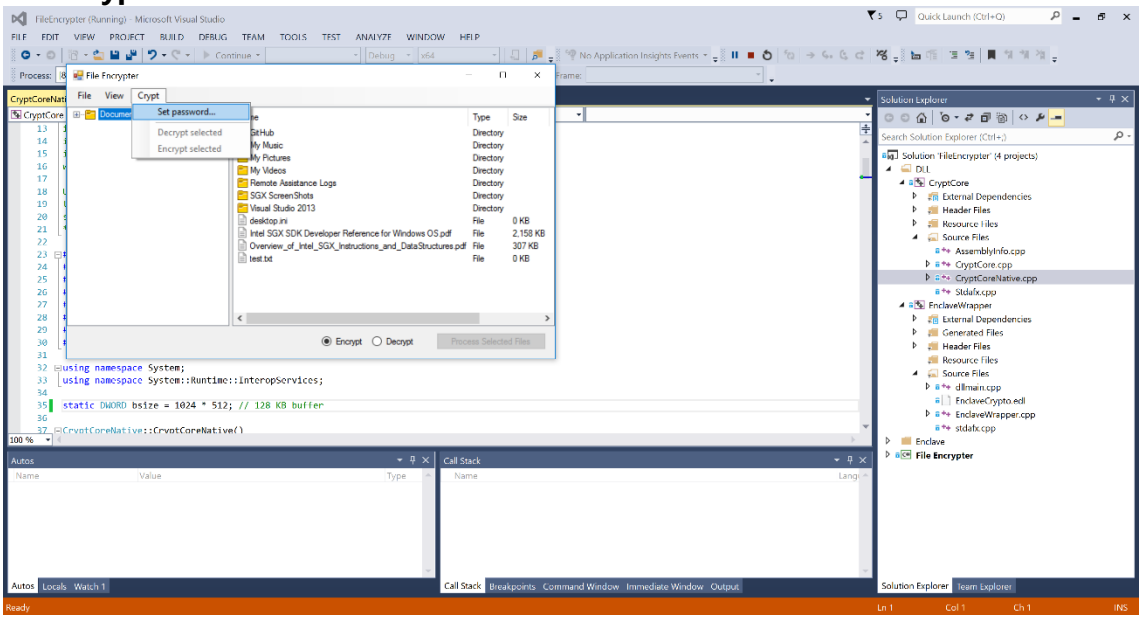
7. The file-encryption program uses a 128-KB buffer size to encrypt and decrypt files. Let's see what happens if the buffer size is increased to 512-KB.
8. Change **128** to **512**.



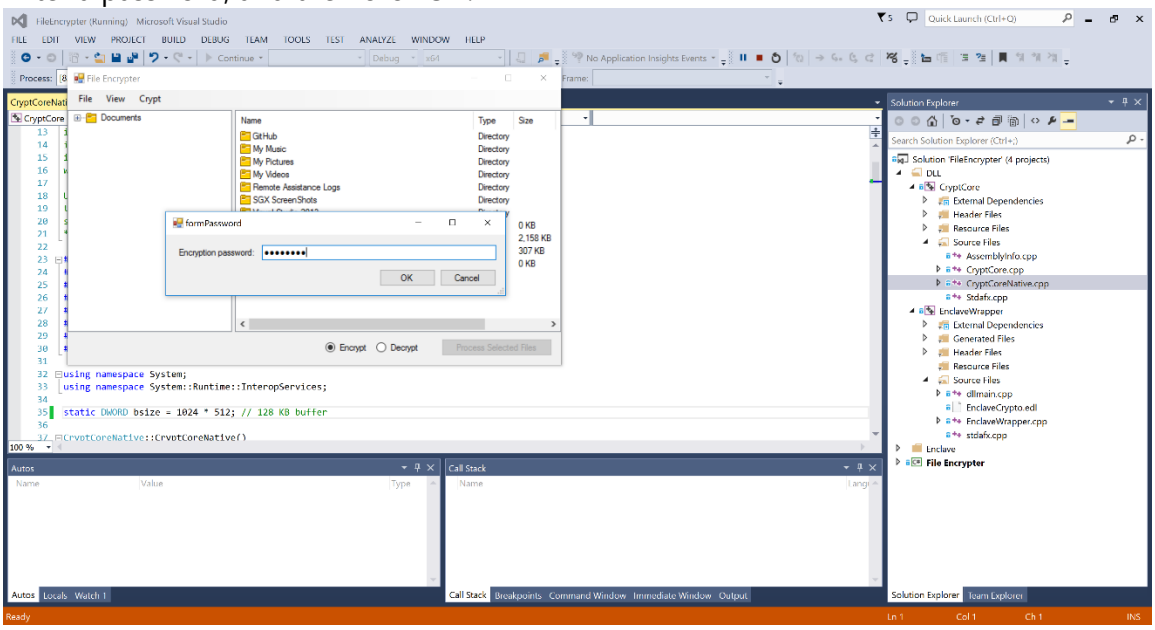
9. Build the application to show how encrypting and decrypting a file works.



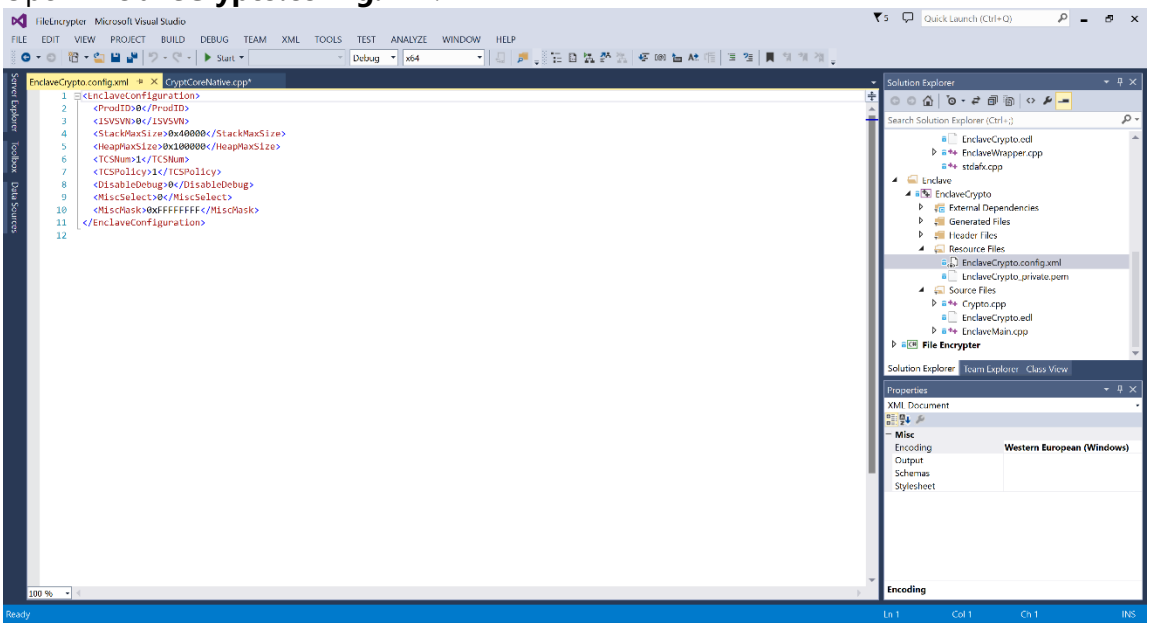
10. Click **Crypt** > **Set Password**.



11. Enter a password, and then click **OK**.



12. Select a file that is greater in size than 512 KB.
13. Click **Process Selected Files**. What error do you receive? How does it help explain what the issue is?
14. Reset the buffer size to **128 KB**.
15. Expand **Enclave** > **EnclaveCrypto** > **Source Files**.
16. Open **EnclaveCrypto.config.xml**.



17. The **StackMaxSize** and **HeapMaxSize** limits are set for the enclave size; adjust these settings as needed. What are the problems that might be created by increasing these limits?

Advanced Exercises	If you have completed the previous exercises, here are some additional lab exercises to try on your own:

1. Right now, the file encryption uses a 128 KB input/output (I/O) buffer. This is defined in CryptCoreNative.cpp:

   static DWORD bsize = 1024 * 128; // 128 KB buffer

   What happens if you increase the buffer size to 512 KB?

   What causes this error?

2. What change can you make to the enclave configuration (EnclaveCrypto.config.xml) to allow this larger buffer size?

3. What other problems might your solution to number 3 create?

Task 5: Advanced Enclave-Definition Language (EDL), ECALL, and OCALL

Task 5.1: Advanced EDL

Lab Objective	In this lab, we will improve the security of our encryption and then use OCALLs to provide updates to a progress bar.

Part 1

In this part, we will change the EDL for ecrypto_crypt_block so that it doesn't copy the blocks to be encrypted/decrypted into and out of the enclave. The encryption still takes place inside the enclave, but the enclave reads directly from unprotected memory and writes directly back to it.

Procedure

1. From the **Enclave** folder, expand the **EnclaveCrypto** project.
2. Open **EnclaveCrypto.edl**.
3. Find **LAB EXERCISE, PART 1** in the comments.
4. Replace the declaration of **ecrypto_crypt_block** as indicated in the comments.
5. Rebuild and test.

Result	The application should be fully functional at this point.

Part 2

In this part, we move the progress bar update from CryptCoreNative.cpp to an OCALL made from the enclave. This is more or less an academic exercise to show how to create OCALLs. The program is actually less efficient this way, but the point is to demonstrate the concept.

Procedure

1. Add the OCALL definition to the enclave:
    a. From the **Enclave** folder, expand the **EnclaveCrypto** project.
    b. Open **EnclaveCrypto.edl**.
    c. Find **LAB EXERCISE, PART 2, STEP 1** in the comments (in the "untrusted" section).
    d. Add the **o_update_progress** function definition by using the comments as a guide.
    e. Keep this file open for Step 2.
2. Modify an ECALL declaration to pass in a callback pointer:
    a. Find **LAB EXERCISE, PART 2, STEP 2** in the comments.
    b. Modify the definition for **ecrypto_crypt_block** to add a callback pointer parameter at the end, using the comments as a guide.
    c. In the **EnclaveCrypto** project, right-click **EnclaveCrypto.edl**.

d. Select **Compile**.
e. This should build with no errors.

Result

```
1>------ Build started: Project: EnclaveCrypto, Configuration: Debug x64 ------
1> Creating proxy/bridge routines
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
```

3. Modify the ECALL function to accept the incoming callback parameter:
    a. Open **EnclaveMain.cpp**.
    b. Find **LAB EXERCISE, PART 2, STEP 3** in the comments.
    c. Modify **ecrypto_crypt_block** to add the callback parameter.
4. Add the OCALL to ecrypto_crypt_block:
    a. Find **LAB EXERCISE, PART 2, STEP 4** in the comments.
    b. Add code to perform the OCALL. Use the comments as a guide.

Our enclave is ready for our OCALL. Now we have to modify the application to pass our callback pointer in and implement the OCALL function.

5. Remove the old callback method from the untrusted code:
    a. From the **DLL** folder, expand the **CryptCore** project.
    b. Open **CryptCoreNative.cpp**.
    c. Locate **LAB EXERCISE, PART 2, STEP 5** in the comments.
    d. Remove the indicated code block.
6. Remove the old callback method from the untrusted code (continued):
    a. Locate **LAB EXERCISE, PART 2, STEP 6** in the comments.
    b. Remove the indicated code block.
7. Pass the callback parameter to the enclave wrapper ew_crypt_block:
    a. Locate **LAB EXERCISE, PART 2, STEP 7** in the comments.
    b. Add the callback parameter using the comments as a guide.
8. Modify the declaration for ew_crypt_block to accept the callback parameter:
    a. From the **DLL** folder, expand the **EnclaveWrapper** project.
    b. Open **EnclaveWrapper.h**.
    c. Find **LAB EXERCISE, PART 2, STEP 8** in the comments.
    d. Modify the declaration for ew_crypt_block using the comments as a guide.
9. Modify the function definition for ew_crypt_block to accept the callback parameter:
    a. Open **EnclaveWrapper.cpp**.
    b. Find **LAB EXERCISE, PART 2, STEP 9** in the comments.
    c. Modify the function to accept the callback pointer parameter.
10. Pass the callback pointer to the ecrypto_crypt_block ECALL:
    a. Find **LAB EXERCISE, PART 2, STEP 10** in the comments.
    b. Modify the call to the ecrypto_crypt_block ECALL to pass the callback pointer parameter.
11. Add the OCALL function:
    a. Find **LAB EXERCISE, PART 2, STEP 11** in the comments.

b. Create the OCALL that makes our callback using the comments as a guide.

12. Clean the solution and rebuild.

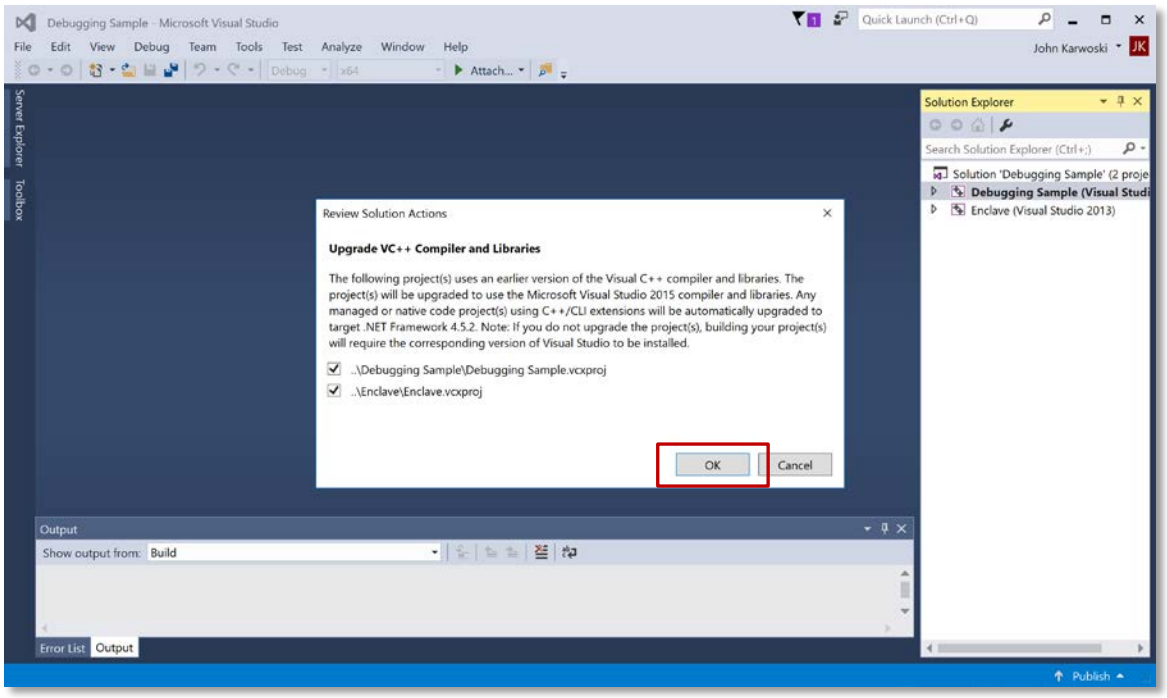
Advanced Exercises	If you have completed the previous exercises, here are some additional lab exercises to try on your own: 1. This example is using an OCALL to update the progress bar as an academic exercise. We don't need to use an OCALL at all here. Why not? 2. What performance issue might we create by making OCALLs where they aren't needed? 3. Rewrite the application to eliminate the OCALL. Does the performance improve significantly? 4. Many enclaves that have to use I/O loops can be implemented as either multiple ECALLs or one ECALL with multiple OCALLs. How would you rewrite this application to do the latter? What are the tradeoffs in doing this?

Task 6: Debugging

Lab Objective	In this lab, we will explore debugging and where to place breakpoints when developing your Intel SGX application.

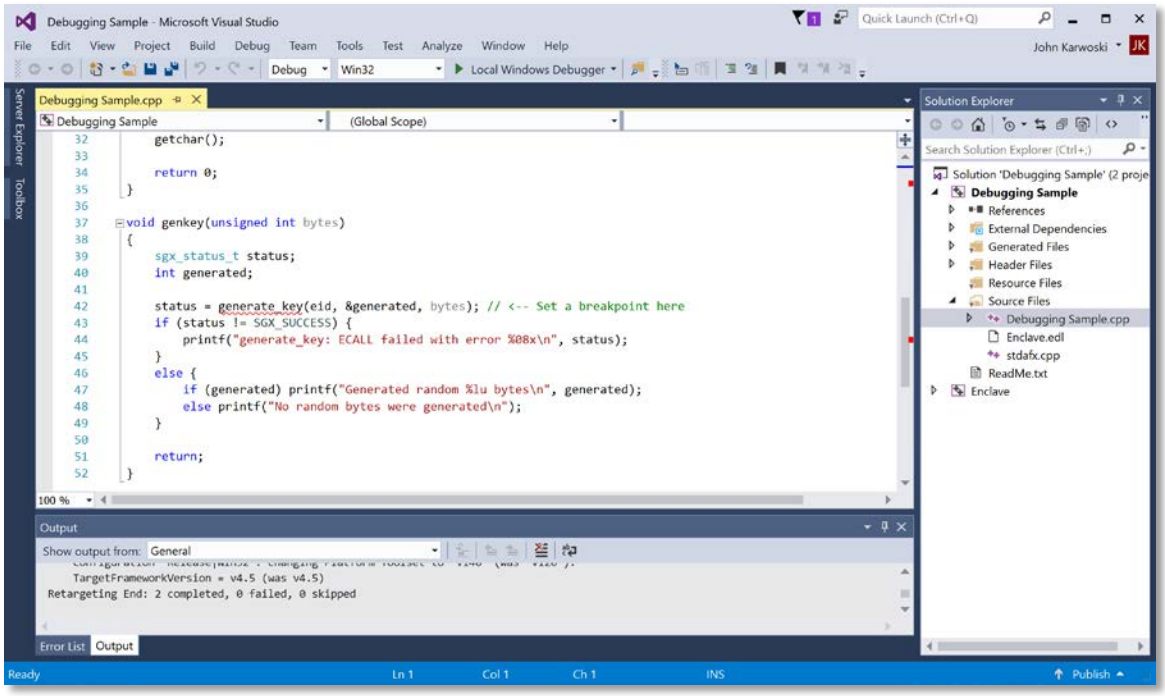
Task 6.1: Debugging Intel SGX Applications

5. Launch Visual Studio 2015.
2. Click **Open Project**.
3. Browse to **Intel-SGX-HOL** > **Debugging**.
4. Select **Debugging Sample.sln**, and then click **Open**.
5. You might see the following notification; if so, click **OK**.

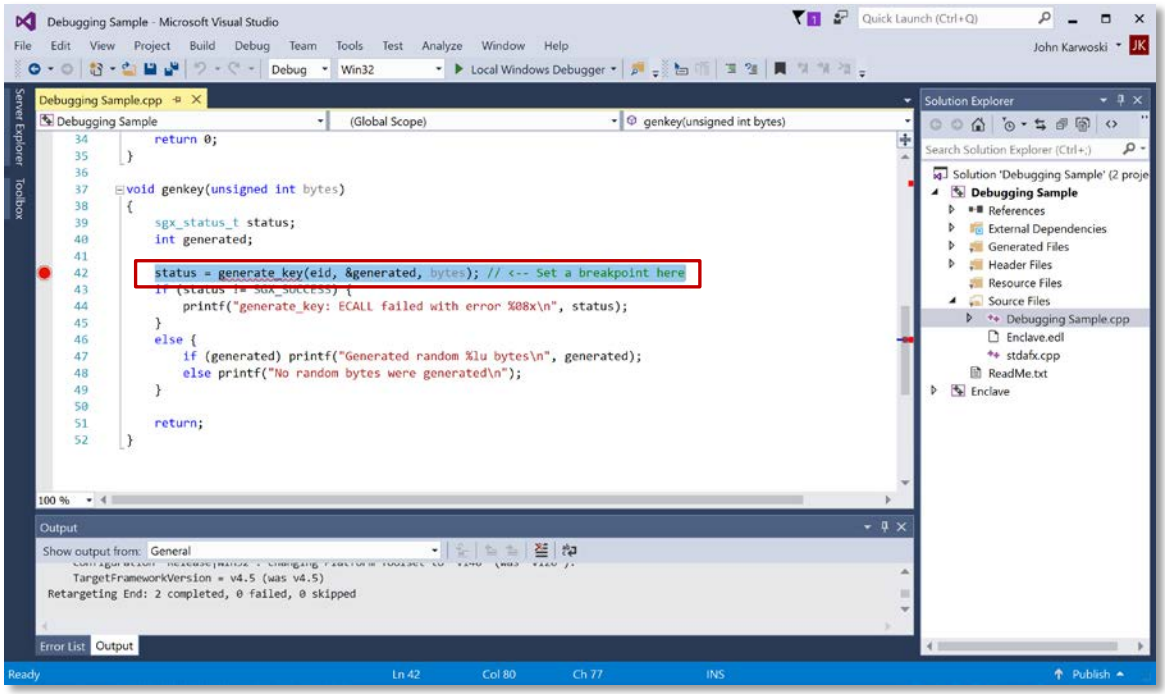


6. Expand **Debugging Sample** > **Source Files**.

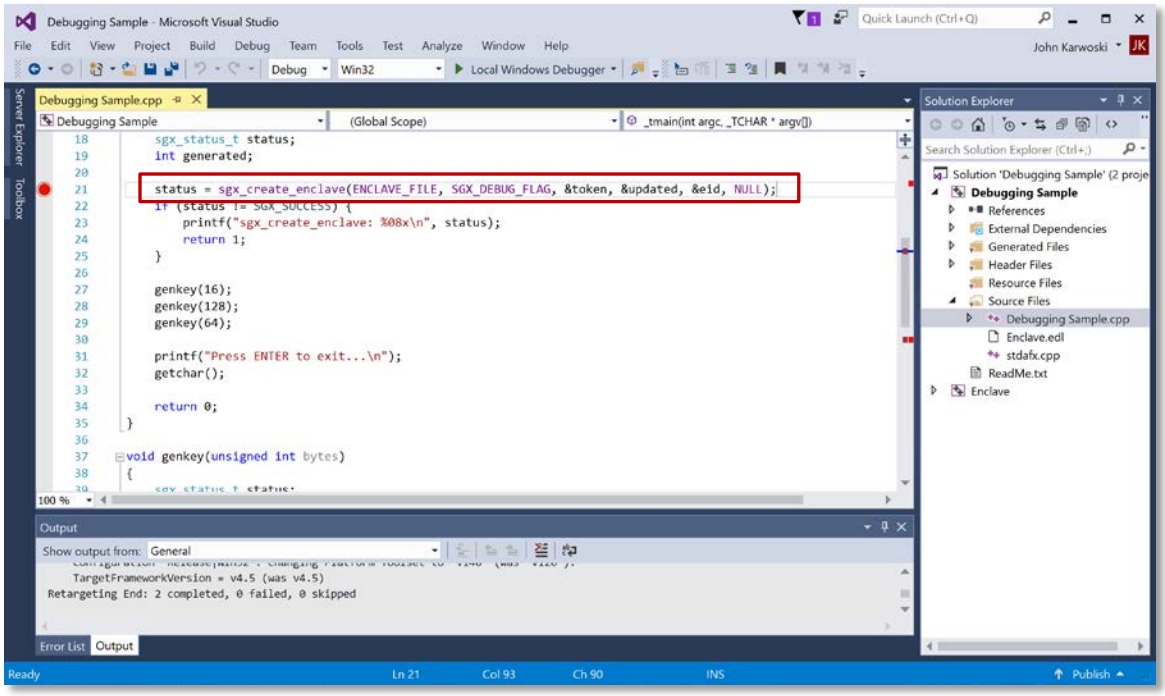
7. Open **Debugging Sample.cpp**.



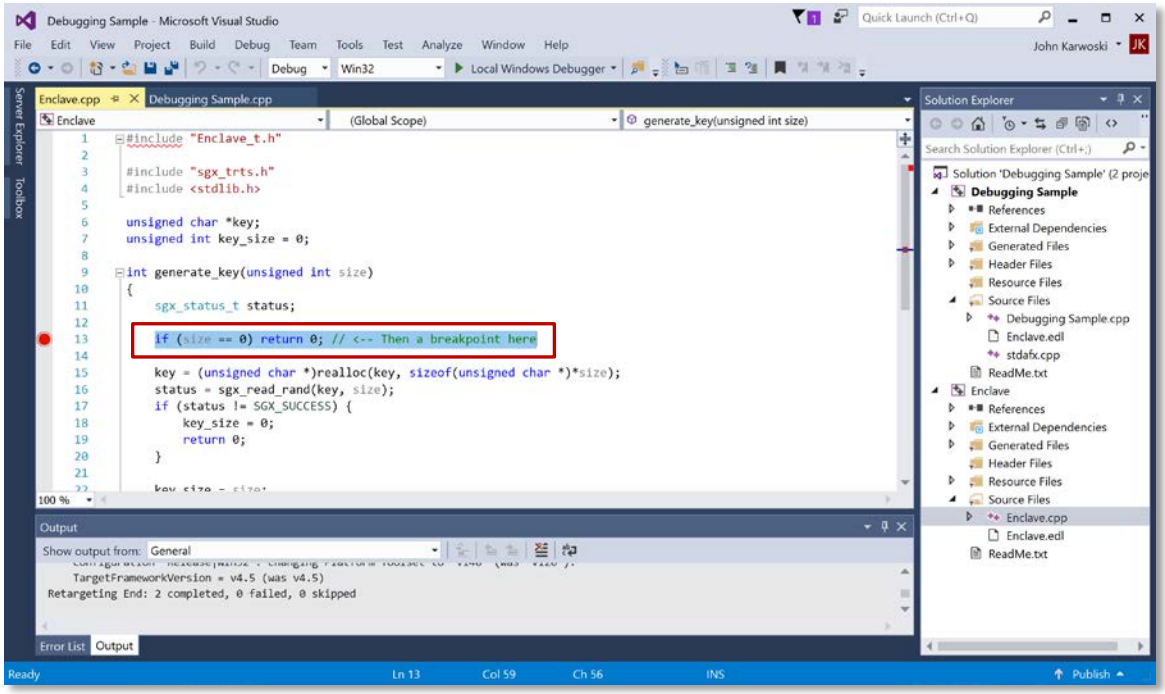
8. Next to the **generate_key** enclave function, locate **Lab Exercise**.
9. Press **F9** to create a breakpoint.



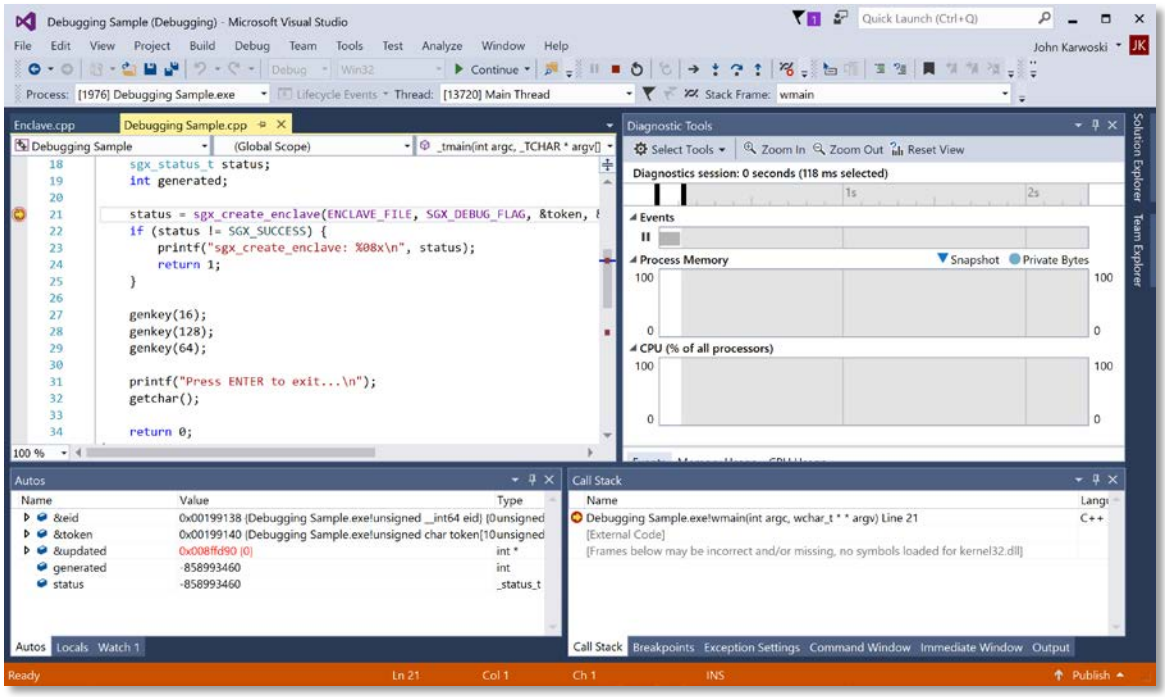
10. Place a breakpoint at **status = sgx_create_enclave(...)**.



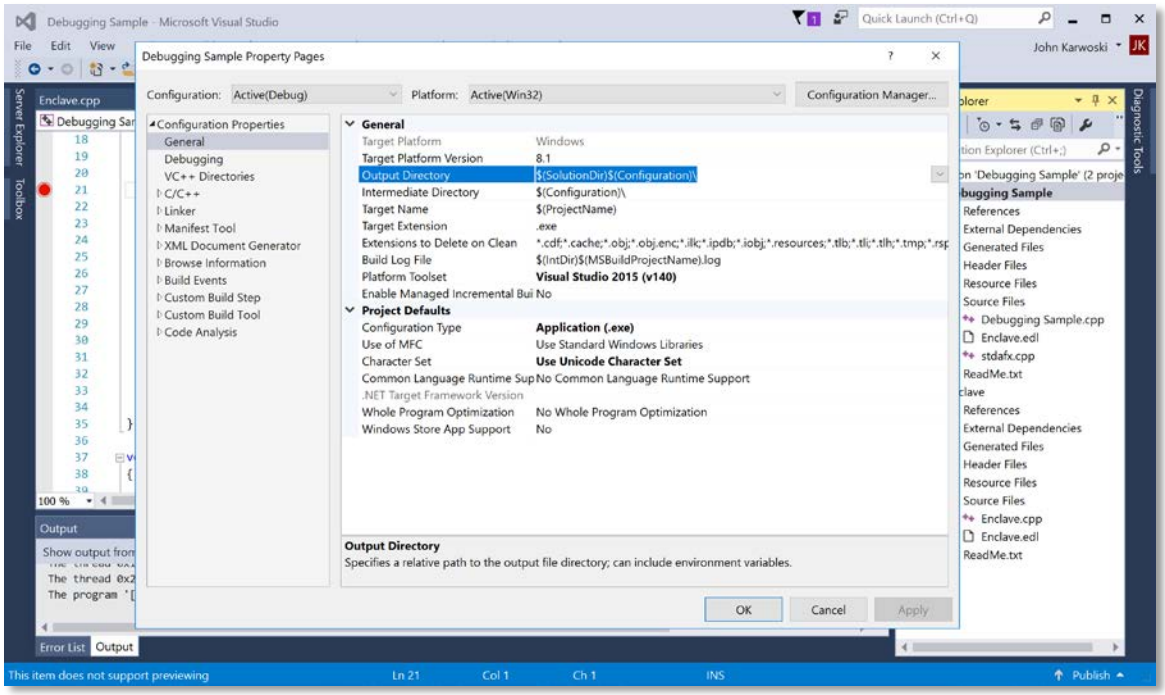
11. Expand **Enclave** > **Source Files**.
12. Open **Enclave.cpp**.
13. Place a breakpoint in the **generate_key** enclave function.



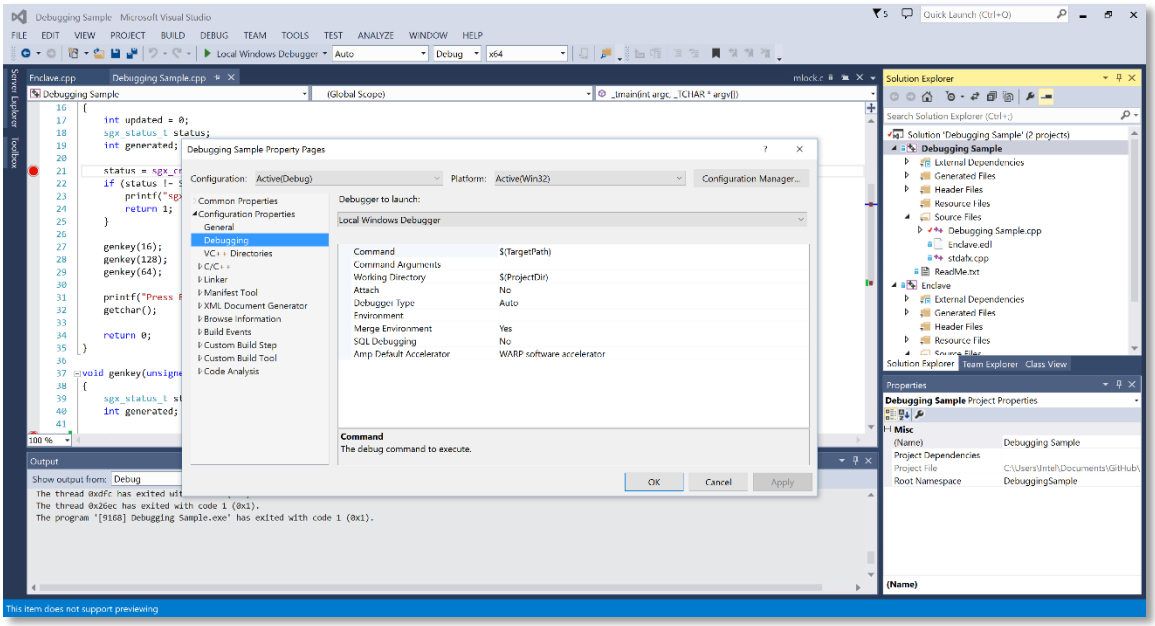
14. Run the **Local Windows Debugger**. What happened in the debugging session?



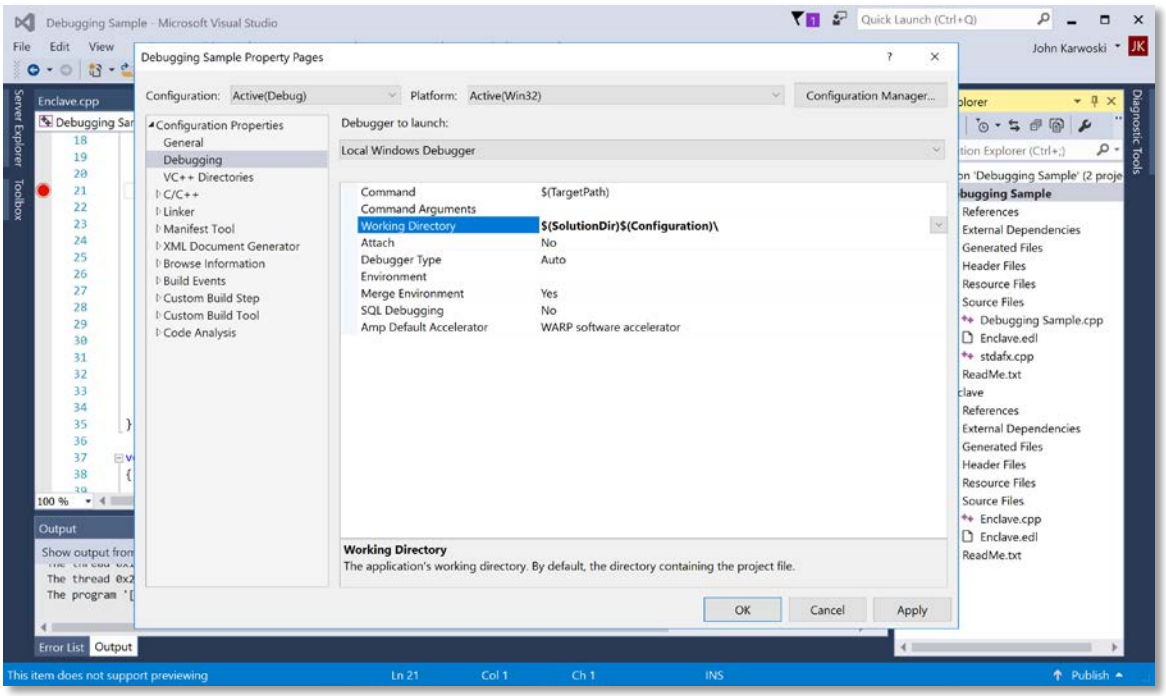
15. Right-click the **Debugging Sample** project, and then select **Properties**.
16. Select the **General** page, and then copy the **Output Directory** path.



17. Select **Debugging**.



18. Paste the path to **Working Directory**.

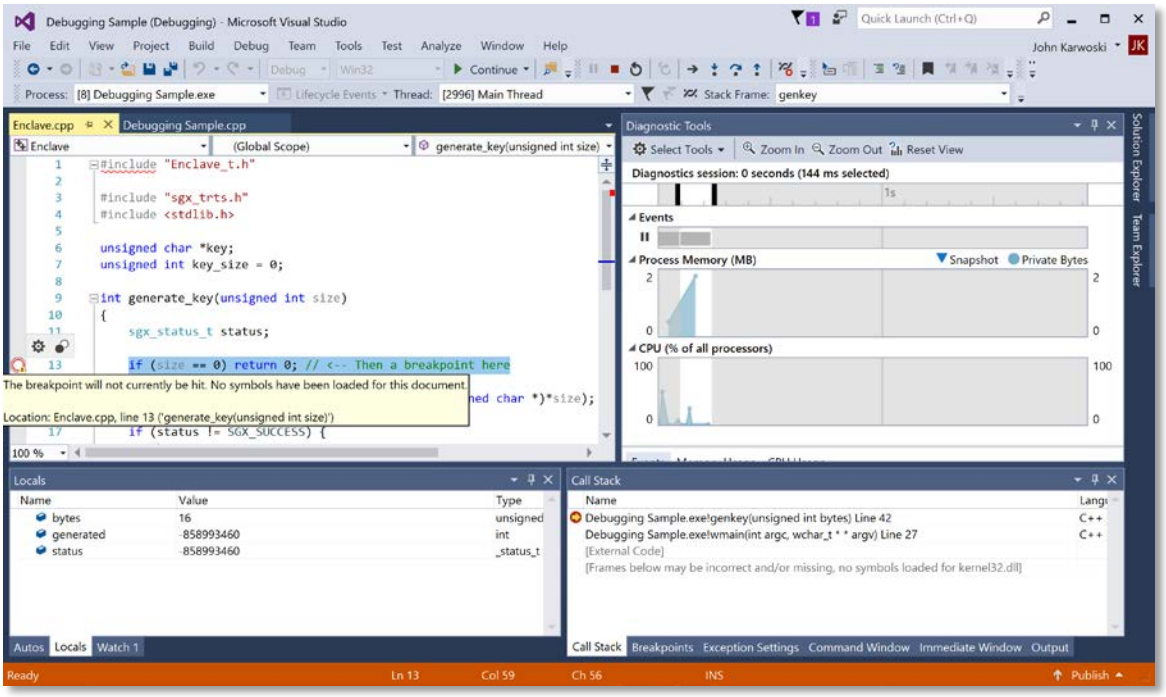


19. Repeat the steps for the **Enclave** project.

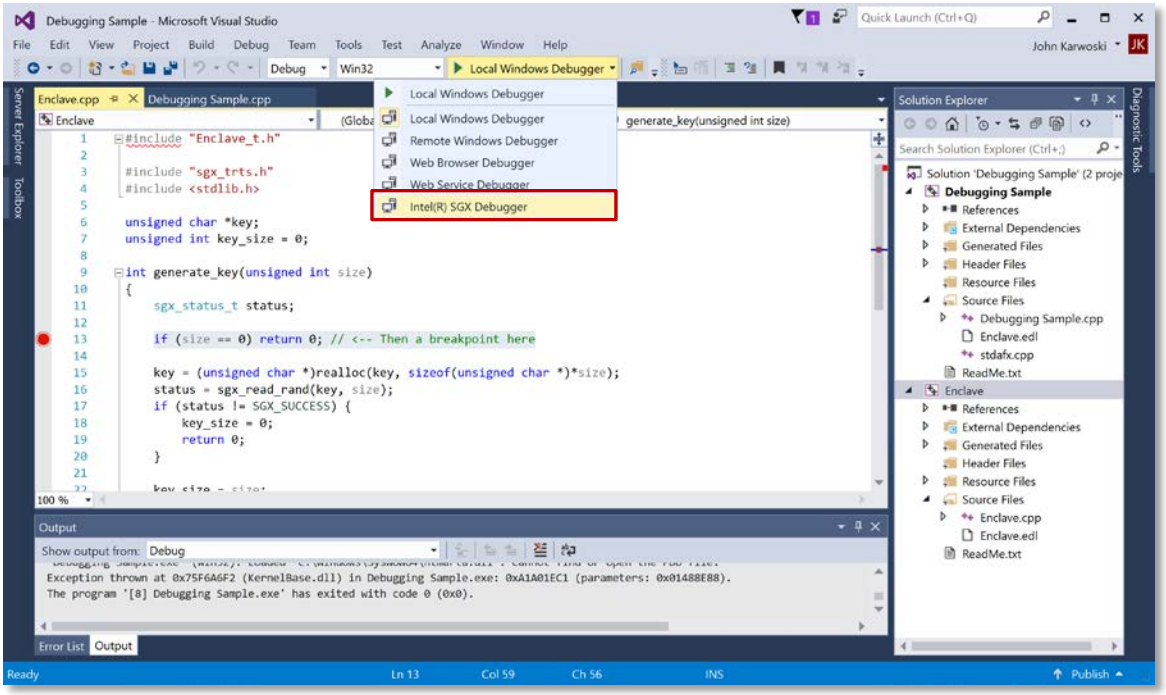
    Run the **Local Windows Debugger** once again. Did the debugger step into the enclave function?

20. Open the **Enclave.cpp** file.

21. Hover over the breakpoint for the error.



22. From the **Local Windows Debugger** drop-down menu, select **Intel SGX Debugger**.



23. For each project, repeat steps 19–22 to set the debugging working directory.
24. Run the **Intel SGX Debugger**.
25. Watch the Autos window to view the changing "bytes" value.

Result	After you run the Intel SGX debugger, you can step into the enclave function.

MORE INFO	Additional information regarding Intel SGX debugging can be found this article.

Task 7: Data Sealing

Lab Objective	In this lab, we will add support for keeping state across sessions. The user's password is sealed and saved in unprotected memory, so it can be restored if the system suspends/resumes. We can't pick up in the middle of encrypting/decrypting a file, but we can prevent the user from having to re-enter a password.

Task 7.1: Data Sealing

1. Add an ECALL to get the sealed-data buffer size:
    a. From the **Enclave** folder, expand the **EnclaveCrypto** project.
    b. Open **EnclaveCrypto.edl**.
    c. Find **LAB EXERCISE, STEP 1** in the comments.
    d. Add the ECALL declaration using the comments as a guide.
    e. Keep the file open for the next step.
2. Modify the ecrypto_set_password ECALL to obtain the sealed-data buffer:
    a. Find **LAB EXERCISE, STEP 2** in the comments.
    b. Add the ECALL declaration using the comments as a guide.
    c. Keep the file open for the next step.
3. Add an ECALL to restore the password from the sealed-data buffer:
    a. Find **LAB EXERCISE, STEP 3** in the comments.
    b. Add the ECALL declaration using the comments as a guide.
4. Add the ECALL ecrypto_get_seal_size to the enclave:
    a. Open **EnclaveMain.cpp**.
    b. Find **LAB EXERCISE, STEP 4** in the comments
    c. Add a function/ECALL named ecrypto_get_seal_size using the comments as a guide.
    d. Keep the file open for the next step.
5. Add the data-sealing parameters to the ECALL ecrypto_set_password:
    a. Find **LAB EXERCISE, STEP 5** in the comments.
    b. Add the parameters to this function using the comments as a guide.
6. Seal the password inside the ECALL ecrypto_set_password:
    a. Find **LAB EXERCISE, STEP 6** in the comments.
    b. Add the call to seal the data using the comments as a guide.
7. Add the ECALL that restores the password from the sealed-data buffer:
    a. Find **LAB EXERCISE, STEP 7** in the comments.
    b. Add the function to unseal the data and restore the user's password using the comments as a guide (a function stub is commented out that you can use as a starting point).
8. Complete the function ew_recreate_enclave that recreates an enclave after a power event:

a. From the **DLL** folder, expand the EnclaveWrapper project.
b. Open **EnclaveWrapper.cpp**.
c. Find **LAB EXERCISE, STEP 8** in the comments.
d. Add the needed subroutine calls to the function stub using the comments as a guide.

9. Create the sealed-data buffer:
    a. Find **LAB EXERCISE, STEP 9** in the comments.
    b. Add the function call and the necessary code to the ew_set_password function using the comments as a guide.
10. Add the data-sealing parameters to the ECALL for ecrypto_set_password:
    a. Find **LAB EXERCISE, STEP 10** in the comments.
    b. Add the parameters to the call to the ecrypto_set_password ECALL using the comments as a guide.
11. Add the ECALL that restores the password to the function ew_restore_password:
    a. Find **LAB EXERCISE, STEP 11** in the comments.
    b. Add the call to the ECALL ecrypto_restore_password to the function stub for ew_restore_password.
12. Add code to recreate the enclave and restore the password to the function ew_crypt_initialize:
    a. Find **LAB EXERCISE, STEP 12** in the comments.
    b. Add the calls to recreate the enclave and restore the password using the comments as a guide.
13. Compile and test.

Advanced Exercises	If you have completed the previous exercises, here are some additional lab exercises to try on your own: Enclaves can be launched by any application, not just the ones "intended" to use them. This is why Intel uses the term "untrusted" to refer to applications and memory outside of an enclave: the enclave cannot and must not trust the calling application. 1. The sealed data is currently saved to memory. What security risk does this create in the file-encrypter program? 2. What would be the more serious security vulnerability if the sealed data was saved to disk? 3. Can this program be "fixed" to eliminate the vulnerabilities in number 1 and number 2 while still preserving the core feature of saving the user's password across a power event? How would you do that?

Conclusion

In the course of this lab, you experienced Intel SGX. Specifically, you learned about:

- How Intel SGX helps protect your data through enclaves, and how Intel SGX enclaves work
- Basic and advanced EDL
- Debugging Intel SGX applications
- Sealing data in order for Intel SGX applications to maintain state

For more information, the Intel SGX SDK manual can be found here:
https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Default.htm

Appendix A—Intel SGX Glossary

This glossary is provided as a convenient means of looking up Intel SGX terminology without having to hunt through other sources. While thorough, it is not all-inclusive, nor should the entries be considered definitive.

AESM

Intel SGX Application Enclave Services Manager, the Intel launch enclave installed as part of the PSW.

Attestation

Attestation is the process of demonstrating that a piece of software has been properly instantiated on the platform. In Intel SGX, it is the mechanism by which another party can gain confidence that the correct software is securely running within an enclave on an enabled platform. See this article for reference.

- **Local Attestation**: Two or more enclaves sharing secrets to address size and efficiency issues, or where applications have more than one enclave that need to work together. Each enclave must verify the other; once done, they establish a protected session and use an Elliptic curve Diffie-Hellman (ECDH) key exchange to share a session key.
- **Remote Attestation**: The process by which one enclave attests (verifies) its trusted computing base (TCB) to another entity outside of the platform. Intel SGX extends local attestation by allowing a quoting enclave (QE) to use the Intel® Enhanced Privacy ID (EPID) to create a quote report. Hardware and software is used to generate a quote that is sent to a third-party server to establish trust.

ECALL

Enclave call: A call from the application into an interface function within the enclave. See this article for reference.

Edger8r

The Edger8r tool ships as part of the Intel SGX evaluation SDK. It generates edge routines by reading a user-provided EDL file. These edge routines provide the interface between the untrusted application and the enclave. Normally, the tool will run automatically as part of the enclave build process. However, an advanced enclave writer might invoke the Edger8r manually.

EDL

Enclave definition language (EDL) files are meant to describe trusted and untrusted enclave functions and types used in the function prototypes. The Edger8r tool uses this file to create C-wrapper functions for both enclave exports (used by ECALLs) and imports (used by OCALLs). See this article for reference.

EMMT

Enclave Memory Management Tool (EMMT): Intel SGX SDK–provided tool (sgx_emmt) to measure the real usage of protected memory by the enclave at runtime. See this article for reference.

Enclave

Protected areas (containers) of execution containing secrets and application code. Enclaves are loaded as DLLs and managed by the CPU.

EPC

Enclave Page Cache (EPC) is a part of physical memory (RAM) that is reserved for enclaves; the EPC is encrypted and trusted by the CPU. The EPC verifies access from inside an enclave and that the linear address is correct.

EPCM

Enclave Page Cache Map (EPCM) is a secure structure used by the processor to track the contents of the EPC. See this article for reference.

EPID

Intel Enhanced Privacy ID (EPID) is the algorithm for attestation of a trusted system while preserving privacy. Intel EPID lets an enclave attest that it's running a given software on a given CPU. See this article for reference.

KDF

Key Derivation Function (KDF) derives one or more secret keys from a secret value such as a master key, a password, or a passphrase using a pseudo-random function. Used by MRSIGNER and MRENCLAVE to seal data. See this article for reference.

ME

Intel® Manageability Engine (ME) resides in the chipset platform controller hub (PCH). Amongst other features, it provides several protection-related functions such as trusted time, monotonic counters, and non-volatile storage. Intel ME is operating-system independent. See this article for reference.

OCALL

Outside call: A call made from within the enclave to the outside application. See this article for reference.

PSE

Platform service enclaves (PSEs) are architectural enclaves from Intel.

PSW

Intel SGX platform software (PSW) installs Intel SGX application enclaves, AESM, Intel SGX Runtime System Library, and drivers. See this article for reference.

SDK

Software-development kit (SDK): The Intel SGX SDK is a collection of APIs, libraries, documentation, sample source code, and tools that allows software developers to create and debug Intel SGX–enabled applications in C/C++. See this article for reference.

SGX

Intel Software Guard Extensions (Intel SGX) is a set of new CPU instructions that can be used by applications to set aside private regions of code and data. See this article for reference.

QE

Quoting enclave (QE) is an architectural enclave from Intel that is involved in the quoting service.

Sealing

Sealing is the process of encrypting enclave secrets for persistent storage to disk. This allows secrets to be retrieved if the enclave is torn down (either due to power event or by the application itself), and subsequently brought back up. See this article for reference.

- **MRSIGNER**: Is a signing identity provided by an authority, which signs the enclave prior to distribution. This value is called MRSIGNER and will be the same for all enclaves signed with the same authority. Thus, it will allow the same sealing key to be used for different enclaves, including different versions of the same enclave. A developer can take advantage of sealing using the signing identity to share sensitive data via a sealed data blob between multiple enclaves for a given application and enclaves of different applications produced by the same development firm. See this article for reference.
- **MRENCLAVE**: Is an enclave identity represented by the value of MRENCLAVE, which is a cryptographic hash of the enclave log (measurement) as it goes through every step of the build and initialization process. MRENCLAVE uniquely identifies any particular enclave, so using the enclave identity will restrict access to the sealed data only to instances of that enclave. Note that different builds/versions of an enclave will result in different MRENCLAVE values. Thus, when sealing using the enclave identity, sealed data will not be available to different versions of the same enclave, only to identical enclave instantiations. See this article for reference.

Secrets

Refers to code or constructs (such as passwords, account numbers, encryption keys, health records, or personally identifiable information [PII]).

TCB

A trusted computing base (TCB) is the portions of hardware and software that are considered safe and uncompromised. A system's protection is improved if the TCB is as small as possible, making an attack harder.

Trusted

"Trusted" refers to code or constructs that run in the Trusted Execution Environment inside an enclave.

Trusted Run-Time System (tRTS)

Code that executes within an enclave environment and that performs functions such as:

- Receiving calls (ECALLs) from the application and making calls (OCALLs) outside the enclave
- Managing the enclave itself

Untrusted

"Untrusted" refers to code or constructs that run in the application environment outside the enclave.

Untrusted Run-Time System (uRTS)

Code that executes outside an enclave environment and performs functions such as:

- Loading and manipulating an enclave (for example, destroying an enclave)
- Making calls (ECALLs) to an enclave and receiving calls (OCALLs) from an enclave

Appendix B—Intel SGX Developer Notes and Best Practices

- What is the signing key when creating an enclave project in Visual Studio? Import an existing signing key to the enclave project; a random key will be generated if no file is selected. The enclave signer will sign the enclave with the key file.
- Every time a secret is provisioned, you should make an OCALL to seal data (based on performance and the effort that was expended to gather the secret).
- Intel SGX AESM will download the latest whitelist.
- The PSW requires administrator privileges to run the installer. From an administrator command prompt, run the following:
    - msiexec /i SGX_PSW.msi
    - To force Intel SGX PSW installation with an administrative account, use the following command:
        - msiexec /i SGX_PSW.msi FORCE_INSTALL=1
        - Silent/unattended installations can be done by adding the /qn or /quiet switch: msiexec /i SGX_PSW.msi /qn
- Unsafe C++11 attributes:

- Developers should use C++11 attributes inside an enclave with care. The attribute noreturn, in particular, might cause a potential security risk. For instance, if a trusted function calls a noreturn function, any clean-up code placed after the function call will be ignored. For example:

```
[noreturn]]void foo(parameters…) { ... } int
ecall_function(parameters…) { ... foo(...); // Clean-up code
below will be ignored ... return 0; }
```

- File I/O inside an Intel SGX enclave: While all the code and data inside an enclave are protected, not all code can be executed inside an enclave; for example, all privileged instructions are invalid in an enclave. This means that all system calls and I/O operations are not available in an enclave. Thus, the standard C library shipped with the Intel SGX SDK is intentionally left incomplete, missing lots of common and useful procedures, such as open, read, write, close, exit, and so on. This makes porting existing applications into an enclave a painful job. The following list of instructions are INVALID inside an enclave:

Types	Instructions
VMEXIT generating instructions are not allowed because a VMM cannot update the enclave. Generates a #UD.	`CPUID`, `GETSEC`, `RDPMC`, `RDTSC`, `RDTSCP`, `SGDT`, `SIDT`, `SLDT`, `STR`, `VMCALL`, `VMFUNC`
I/O instructions (also VMEXIT). Generates #UD.	`IN`, `INS/INSB/INSW/INSD`, `OUT`, `OUTS/OUTSB/OUTSW/OUTSD`
Instructions which access segment registers will also generate #UD.	Far `CALL`, Far `JMP`, Far `RET`, `INT n` / `INTO`, `IRET`, `LDS/LES/LFS/LGS/LSS`, `MOV` to `DS/ES/SS/FS/GS`, `POP DS/ES/SS/FS/GS`, `SYSCALL`, `SYSENTER`
Instructions that try to reenter the enclave. Generates #GP.	`ENCLU[EENTER]`, `ENCLU [ERESUME]`

- Thread-Binding Policy: When an enclave writer develops an enclave that might employ more than one thread, the developer must be aware that untrusted code controls the binding of an untrusted thread to the trusted-thread context (composed of a TCS page, SSA, stack, and thread-local storage variables). Thus, the developer must follow the policies on using thread-local storage and thread-synchronization objects within the enclave. See the SDK developers guide for details.
- The EPC is limited to just 128 MB (of which 88 MB is usable) as of this writing. Best practices suggest using default memory allocations for development. Then, use the

EMMT provided in the SDK to determine actual memory use, and adjust accordingly. The EMMT can identify enclave memory usage where other memory tools cannot.

- An enclave-configuration file is used to set/update the following enclave settings: ProdID, ISVSVN, StackMaxSize, HeapMaxSize, TCSNum, TCSPolicy, DisableDebug, MiscSelect, and MiscMask. See this article for reference.
- Rely on the SDK as the authority on supported operating systems; this will likely be following the "latest" operating system.
- Consider sealing after the state of your enclave has changed, based on a balance of performance.
- There is not a current published roadmap for the SDK.
- Do not use sealing as a general-purpose encryption tool; instead, target keys rather than data.
- Library Patching: Say you are going to port something like the following into enclave (see this article for reference):

```
#include <stdio.h>
int log_lvl = 4;
#define log(level, msg) {                       \
    if (level < log_lvl) fprintf(stderr, msg);  \
} /* compiler error! */
```

The problem is that stdio.h, shipped with the Intel SGX SDK, has neither stderr nor fprintf. Commenting out every occurrence of log is not ideal; not only is it tedious and error-prone, it deprives you of the ability to do logging, which is critical for debugging purposes. In contrast, the following solution is simple and elegant, because it keeps your logging code and requires modifying only one-line of code (for each file):

```
#include "stdio.h" /* use a patched header file! */

int log_lvl = 4;
#define log(level, msg) {                       \
    if (level < log_lvl) fprintf(stderr, msg);  \
} /* compiler ok! */
```

  - The idea is simple: we write a patched version of the C-library header, which includes the stuff that is missing. This technique is called library patching. The code below is our new stdio.h:

```
/* stdio.h */
#ifndef __STDIO_H
#define __STDIO_H

#include <stdio.h>

#ifndef _INC_FCNTL
#define _INC_FCNTL

#define O_RDONLY       0x0000  /* open for reading only */
#define O_WRONLY       0x0001  /* open for writing only */
#define O_RDWR         0x0002  /* open for reading and writing */
#define O_APPEND       0x0008  /* writes done at eof */

#define O_CREAT        0x0100  /* create and open file */
#define O_TRUNC        0x0200  /* open and truncate */
#define O_EXCL         0x0400  /* open only if file doesn't already ex
ist */

#define O_TEXT         0x4000  /* file mode is text (translated) */
#define O_BINARY       0x8000  /* file mode is binary (untranslated)
 */
#define O_WTEXT        0x10000 /* file mode is UTF16 (translated) */
#define O_U16TEXT      0x20000 /* file mode is UTF16 no BOM (translate
d) */
#define O_U8TEXT       0x40000 /* file mode is UTF8  no BOM (translate
d) */

#endif

#ifdef __cplusplus
extern "C" {
#endif

extern int stdin, stdout, stderr;

int open(const char* filename, int mode);
int read(int file, void *buf, unsigned int size);
int write(int file, void *buf, unsigned int size);
void close(int file);

void fprintf(int file, const char* format, ...);

#ifdef __cplusplus
}
#endif

#endif
```

- Include a list of enclave project files. Assuming the enclave project name is sample_enclave, here is the list of files generated by the wizard):
    - **Source files**:
        - sample_enclave.cpp: main source file, to be filled with user functions and variables. The user can add additional source files.
        - sample_enclave_t.c: trusted auto-generated wrapper functions. Do not modify this file, because every build recreates it.

        - sample_enclave.edl: EDL file. Declares which functions are exported (trusted) and imported (untrusted) by the enclave. EDL syntax is explained in a separate section.
    - **Header files:**
        - sample_enclave_t.h: trusted auto-generated header for wrapper functions. Do not modify this file, because every build recreates it.
    - **Resource files:**
        - sample_enclave.config.xml: specifies the enclave configuration. Details are explained in a separate section.
        - sample_enclave.private.pem: RSA private key used to sign the enclave.
    - Note: The private key must be kept secret and safe. If exposed, the key could be used by malware writers to create a valid signed enclave. If you do not want to expose the private key in the enclave project, you can use sgx_sign to sign the enclave in a separate environment.
- An EDL file is used to define the enclave interface. There are two parts to an EDL file:
    - The **trusted section** defines the ECALLs:
        - ECALLs define entry points into the enclave.
        - It is important to note that an Intel SGX–enabled application should always have at least one public ECALL to enter the enclave.
    - The **untrusted section** defines the OCALLs:
        - OCALLs define the transfer of control from inside the enclave to the application to perform system calls and other I/O operations.
        - OCALLs could also be used in cases where the enclave needs to transfer data back to the application.
        - OCALLs are optional.
    - A sample EDL template is shown below.

```
enclave {
//Include files
//Import other edl files
//Data structure declarations to be used as parameters of the
//function prototypes in edl
 trusted {
//Include file if any.
//It will be inserted in the trusted header file (enclave_t.h)
//Trusted function prototypes
 };
 untrusted {
//Include file if any.
//It will be inserted in the untrusted header file (enclave_u.h)
//Untrusted function prototypes
 };
};
```