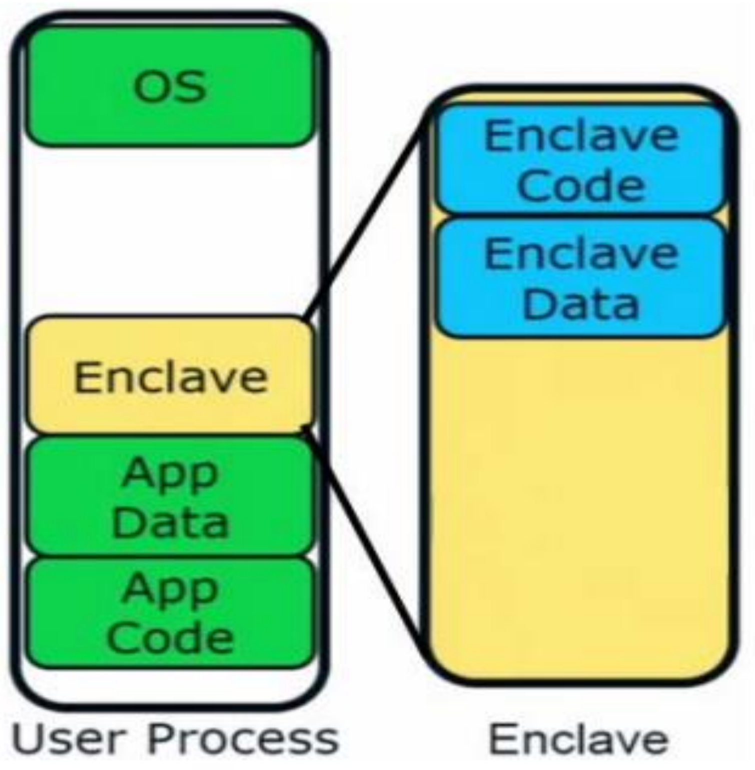
Intel SGX:

Intel Software Guard Extensions (Intel SGX) is an Intel technology for application developers who are seeking to protect select code and data from disclosure or modification.

Enclave:

A trusted execution environment embedded in a process.
The core idea of SGX is the creation of a software 'enclave'.
The enclave is basically a separated and encrypted region for code and data.
The enclave is only decrypted inside the processor, so it is even safe from the RAM being read directly.



- It is part of the application & has full access to its memory.
- Measured and verified by HW on load.
- Operation visible only within CPU borders.
- Manageable by the OS, e.g.: - CPU time slots - Paging and memory policy
- Secrets are provisioned or acquired only after enclave initialization.

Enclave Based Security:

Intel designed Intel® Software Guard Extensions to protect against both hardware and software attacks.

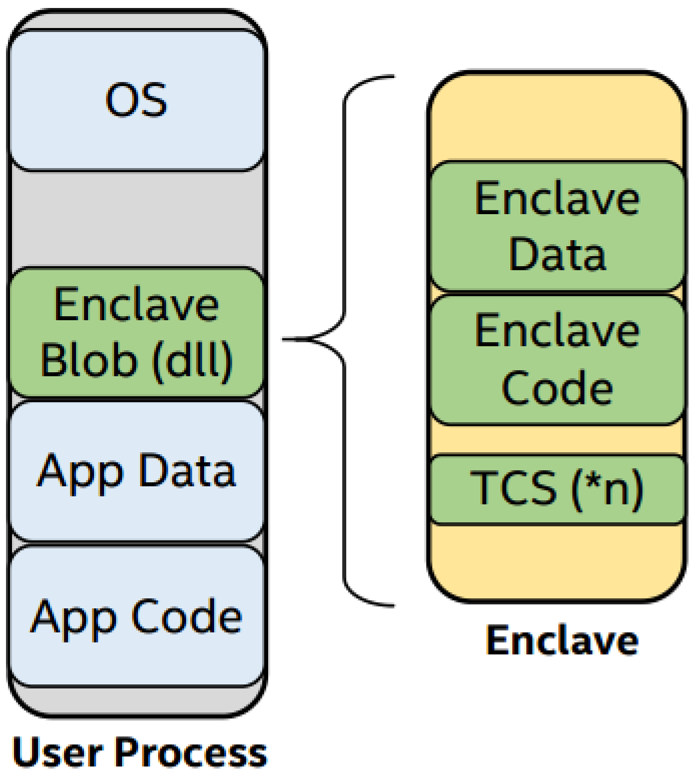
For software protection:

- The enclave memory cannot be read or written from outside the enclave regardless of current privilege level and CPU mode (ring3/user-mode, ring0/kernel-mode, SMM, VMM or another enclave). The abort page is returned in such conditions.
- An enclave can be created with a debug attribute that allows a special debugger (Intel® Software Guard Extensions debugger) to view its content like a standard debugger. Production enclaves (non-debug) cannot be debugged by software or hardware debuggers.
- The enclave environment cannot be entered via classic function calls, jumps, register manipulation or stack manipulation. The only way to call an enclave function is via a new instruction that performs several protect checks. Classic function calls initiated by enclave code to functions inside the enclave are allowed.
- CPU mode can only be 32- or 64-bit when executing enclave code. Other CPU modes are not supported. An exception is raised in such conditions.

For hardware protection:

- The enclave memory is encrypted using industry-standard encryption algorithms with replay protection.
- Tapping the memory or connecting the DRAM modules to another system will only give access to encrypted data.
- The memory encryption key changes every power cycle randomly (for example, boot/sleep/hibernate). The key is stored within the CPU and is not accessible.
- Intel® Software Guard Extensions is not designed to handle side channel attacks or reverse engineering. It is up to the Intel® SGX developers to build enclaves that are protected against these types of attack.

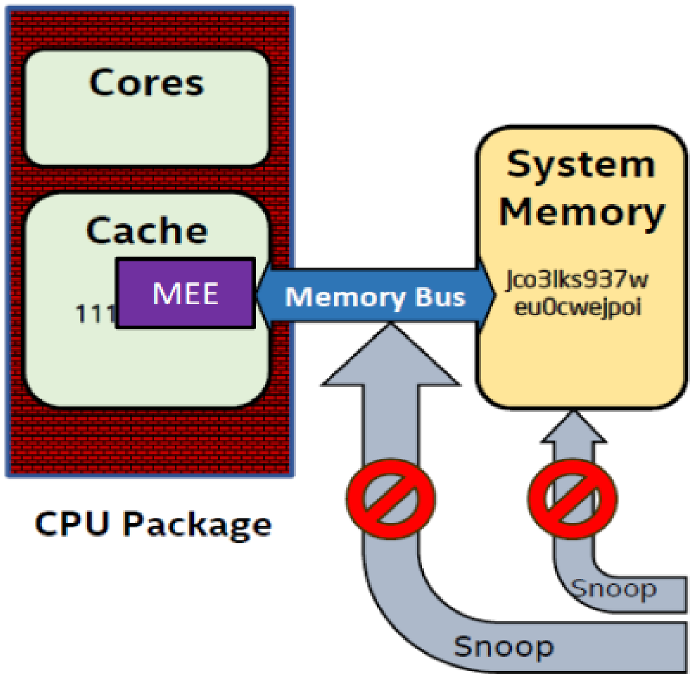
Protected execution environment embedded in a process:



- With its own code and data.
- Providing Confidentiality & Integrity.
- Controlled entry points.
- Multi-thread support.
- Full access to app memory and processor performance

Security Perimeter:

- Security perimeter is the CPU package boundary.
- Data and code unencrypted inside CPU package.
- Data and code outside CPU is encrypted and integrity checked.
- External memory reads and bus snoops see only encrypted data.
- Single chip TCB avoids inter-chip HW attacks that threaten TPMs.
- If the single TCB is the CPU, then we gain the opportunity for richer semantics by understanding platform state & app code.



Enclave Measurement:

- The Intel® SGX architecture is responsible for establishing identities for attestation and sealing.
- For each enclave it provides two measurement registers, MRENCLAVE and MRSIGNER;
    a. MRENCLAVE provides an identity of the enclave code and data as it's constructed
    b. MRSIGNER provides an identity of an authority over the enclave.
- These values are recorded while the enclave is built, and are finalized before enclave execution commences.
- Only the TCB has access to write to these registers in order to ensure an accurate reflection of the identities is available when attesting and sealing.

MRENCLAVE - Enclave Identity:

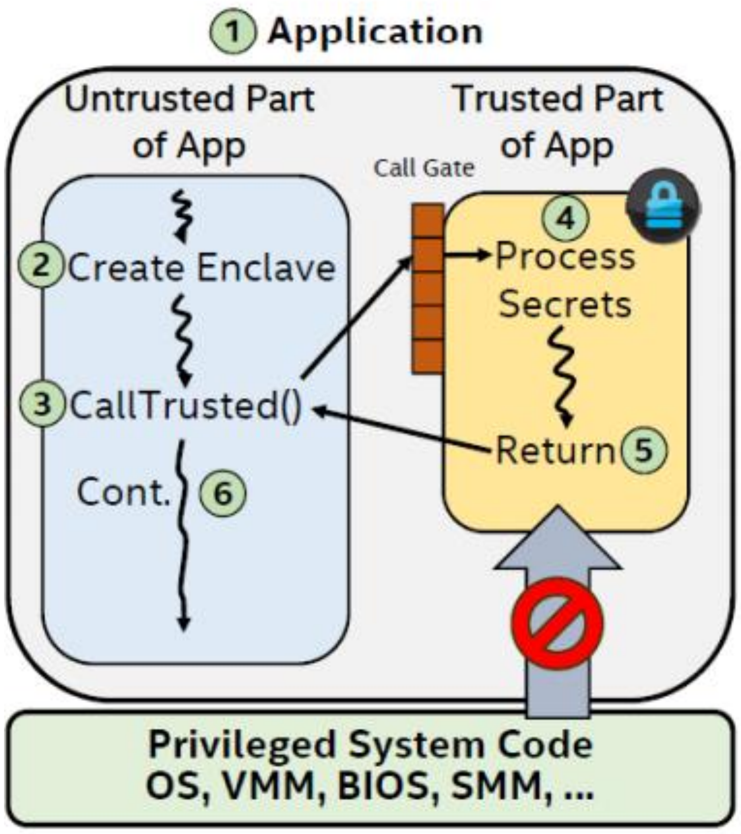
- The "Enclave Identity" is the value of MRENCLAVE, which is a SHA-256 digest of an internal log that records all the activity done while the enclave is built.
- The log consists of the following information:
    a. The contents of the pages (code, data, stack, heap).
    b. The relative position of the pages in the enclave.
    c. Any security flags associated with the pages.

- Once enclave initialization is complete, through the EINIT instruction, no more updates are made to MRENCLAVE.
- The final value of MRENCLAVE is a SHA-256 digest that identifies, cryptographically, the code, data, and stack placed inside the enclave, the order and position in which the enclave's pages were placed, and the security properties of each page.
- Any change to any of these variables would lead to a different value in MRENCLAVE.

MRSIGNER - Sealing Identity:

- The enclave has a second identity used for data protection called the "Sealing Identity."
- The Sealing Identity includes a "Sealing Authority," a product ID and a version number.
- The Sealing Authority is an entity that signs the enclave prior to distribution, typically the enclave builder.
- The enclave builder presents the hardware with an RSA signed enclave certificate (SIGSTRUCT) that contains the expected value of the Enclave Identity, MRENCLAVE, and the public key of the Sealing Authority.
- The hardware checks the signature on the certificate, using the public key contained within, and then it compares the value of the measured MRENCLAVE against the signed version.
- If these checks pass, a hash of the public key of the Sealing Authority is stored in the MRSIGNER register.
- It is important to note that if multiple enclaves are signed by the same Sealing Authority, they will all have the same MRSIGNER value.
- The value of Sealing Identity can be used for sealing data in a way that enclaves from the same Sealing Authority (e.g., different versions of the same enclave) can share and migrate their sealed data.

Enclave Execution Flow:



1. App is built with trusted and untrusted parts
2. App runs & creates enclave which is placed in trusted memory. All secret functions needs to include inside enclave (Ex: Encryption algorithms, Decryption algorithms and secret keys, etc.)
3. Trusted function is called, execution transitioned to the enclave (enclave associated application can only call particular function).
4. Enclave sees all process data in clear; external access to enclave data is denied.
5. Function returns; enclave data remains in trusted memory.
6. Application continues normal execution.

Note:

Secrets are protected from bad actors with access to the platform.
Prevents SW attacks even when OS/drivers/BIOS/VMM/SMM are compromised.
Production enclaves can be deleted at any time but not inspected after build.

Developing a Sample Enclave Application

Step 1

In this topic, you will see a quick guide of how to develop an enclave application.

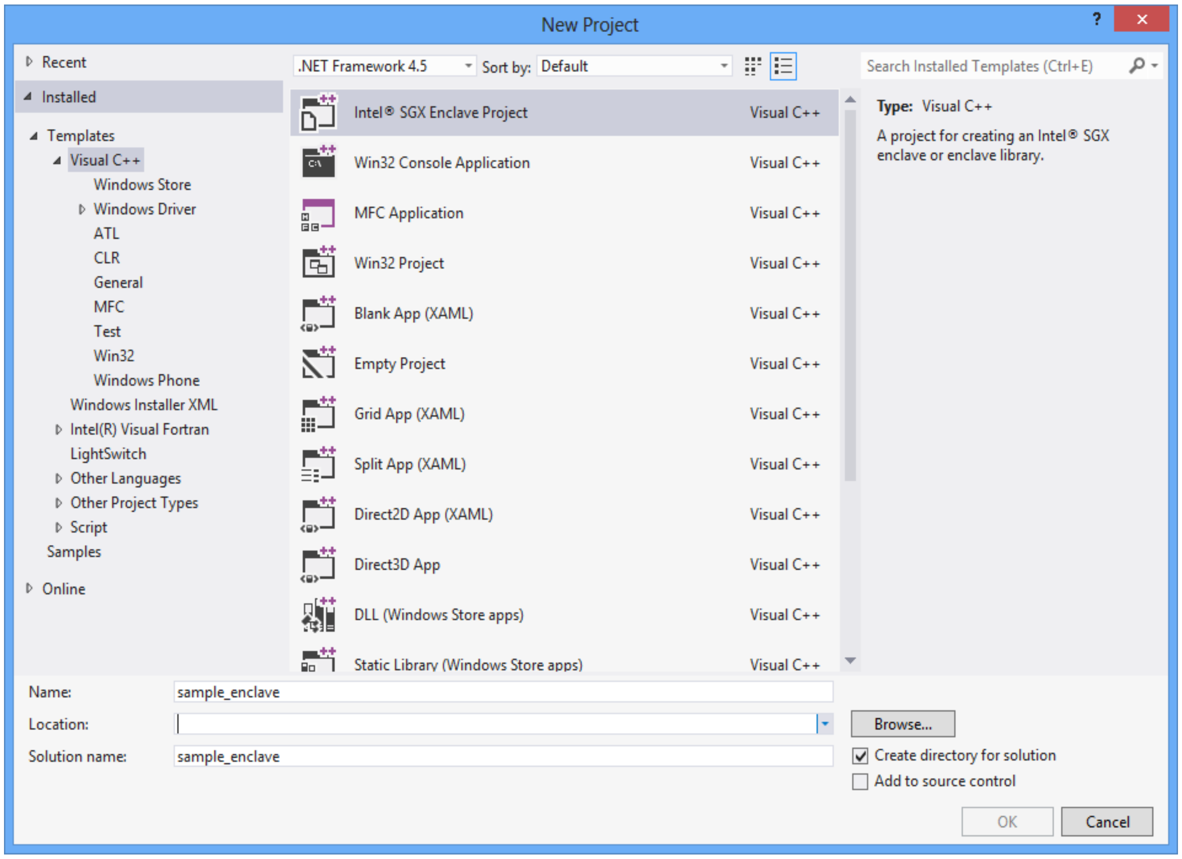
```
#include <stdio.h>
#include <string.h>
#define MAX_BUF_LEN 100

void foo(char *buf, size_t len)
{
	const char *secret = "Hello App!";
	if (len > strlen(secret))
	{
		memcpy(buf, secret, strlen(secret) + 1);
	}
}

int main()
{
	char buffer[MAX_BUF_LEN] = "Hello World!";
	foo(buffer, MAX_BUF_LEN);
	printf("%s", buffer);
	return 0;
}
```

The program displays the string Hello App!

Step 2: Create an Enclave

1. On the menu bar of Microsoft* Visual Studio*, **choose File-->New-->Project**.
    The New Project dialog box opens.

2. Select **Templates-->Visual C++-->Intel® SGX Enclave Project**. Enter name, location, and solution name in the appropriate fields like any other Microsoft* Visual Studio* project.



Type: Visual C++

A project for creating an Intel® SGX enclave or enclave library.

Name: sample_enclave

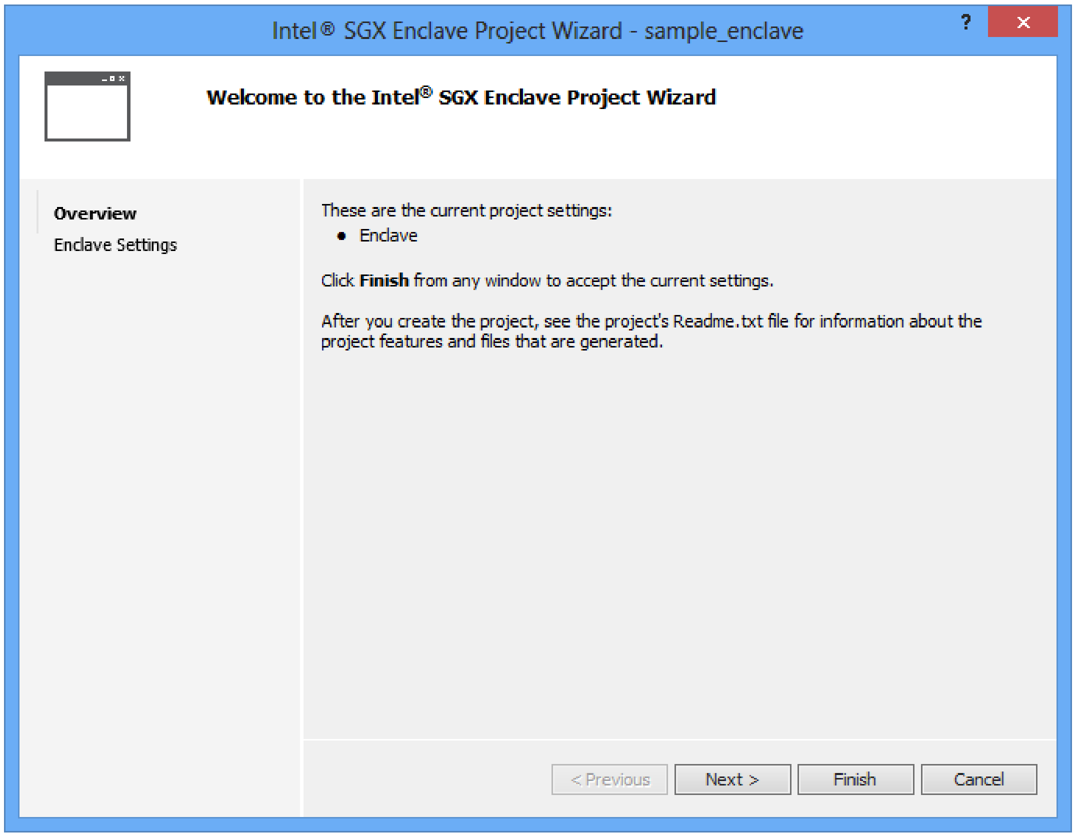
Location:

Solution name: sample_enclave

☑ Create directory for solution

☐ Add to source control

3. Click **OK** and the welcome dialog appears.



Welcome to the Intel® SGX Enclave Project Wizard

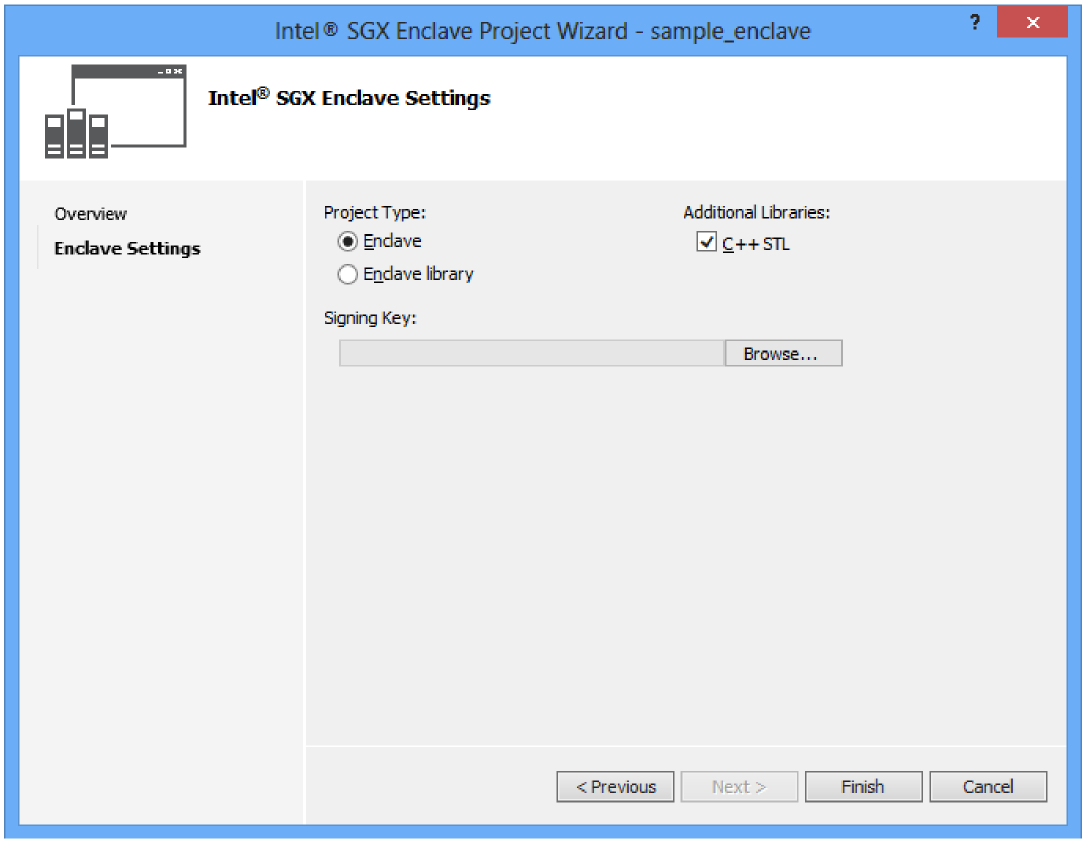
These are the current project settings:

- Enclave

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's Readme.txt file for information about the project features and files that are generated.

4. Click **Next** to go to the Enclave Settings page.


Intel ® SGX Enclave Project Wizard - sample_enclave

Intel® SGX Enclave Settings

Overview

Enclave Settings

Project Type:
(●) Enclave
( ) Enclave library

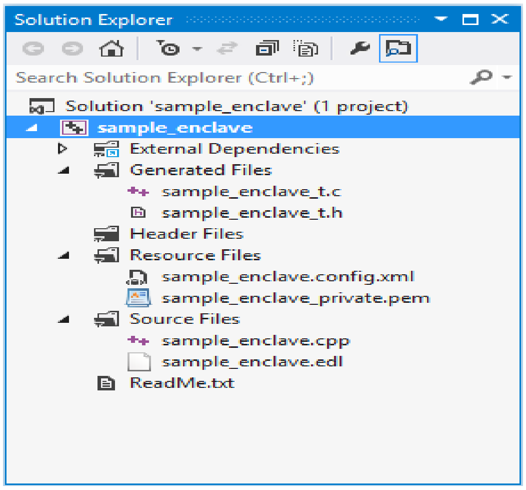
Additional Libraries:
[✓] C++ STL

Signing Key:
Browse...

5. Configure the enclave with proper settings

- **Project Type:**
  - Enclave – Create an enclave project.
  - Enclave library – Create a static library for an enclave project.
- **Additional Libraries:**
  - C++ STL – Link C++ STL with the enclave project.
- **Signing Key:**
  - 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.

When the enclave project is created, the wizard ensures that the enclave project has proper settings.


Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'sample_enclave' (1 project)
  sample_enclave
    External Dependencies
    Generated Files
      sample_enclave_t.c
      sample_enclave_t.h
    Header Files
    Resource Files
      sample_enclave.config.xml
      sample_enclave_private.pem
    Source Files
      sample_enclave.cpp
      sample_enclave.edl
    ReadMe.txt
```

Step 3: Define Enclave Interface

Use an EDL file to define the enclave interface, which exposes a trusted interface foo. The EDL file might look like the following:

```
// sample_enclave.edl
enclave {
   trusted {
      public void foo([out, size=len] char* buf, size_t len);
   };
};
```

Step 4: Import Enclave to Application

To call the enclave interface in the application, import the enclave to the application using Microsoft* Visual Studio* Intel® Software Guard Extensions Add-in.

1. Right click the application project and select **Intel® SGX Configuration -> Import Enclave**. The Import Enclave dialog box opens.

2. Check the **sample_enclave.edl** box, and then press **OK**.

Step 5: Implement Application and Enclave Functions

To implement application and enclave functions, use the following code samples:

The enclave code

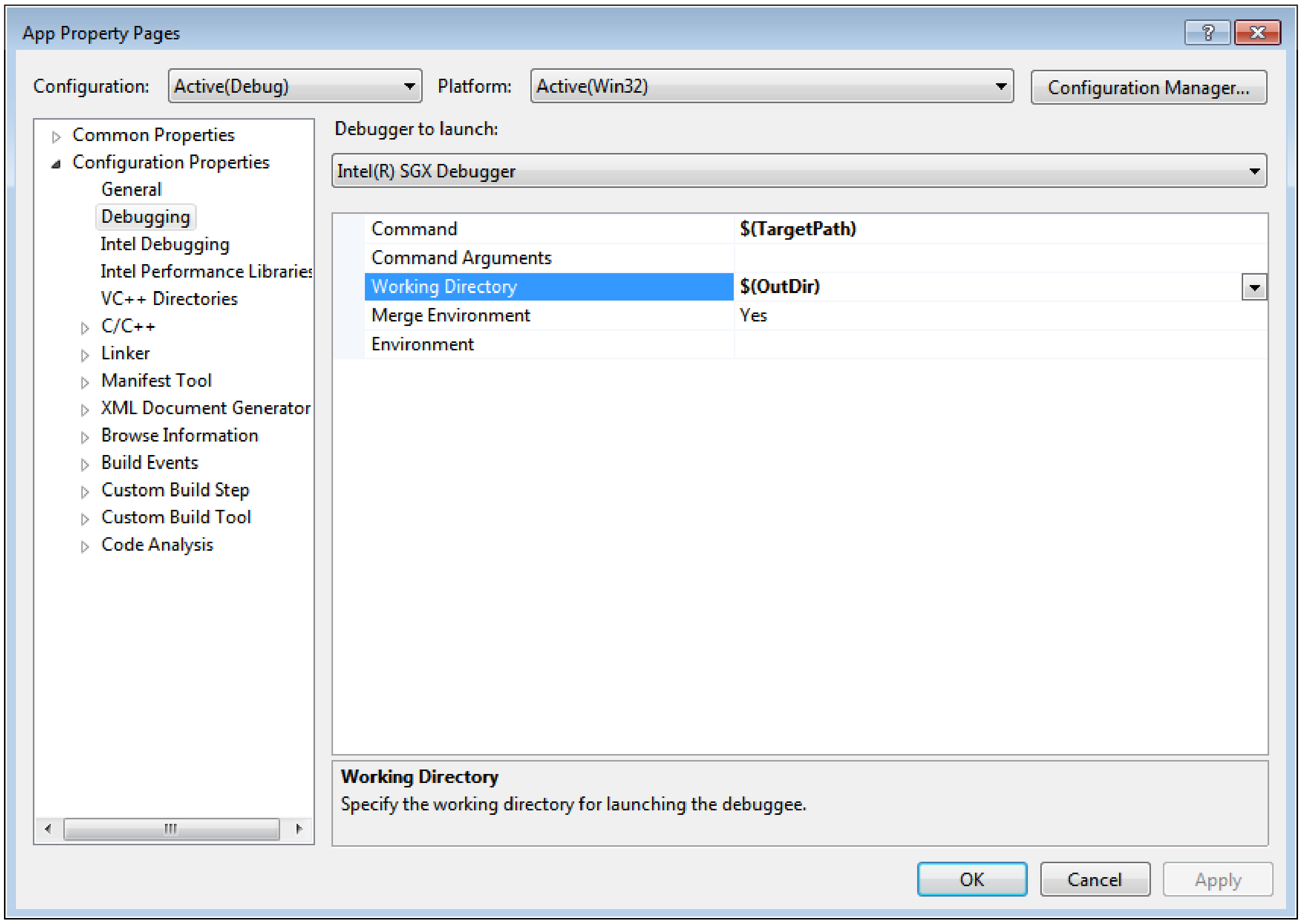
```
// sample_enclave.cpp
#include "sample_enclave_t.h"
#include <string.h>

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

The application code

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

int main()
{
	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;
	return 0;
}
```

Step 6: Compilation and Execution

Now you can compile the application and enclave projects. After the compilation, set the working directory to the output directory and run the program. You should get the string Hello Enclave!


App Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Debugger to launch: Intel(R) SGX Debugger

Command	$(TargetPath)
Command Arguments	
Working Directory	$(OutDir)
Merge Environment	Yes
Environment	

Working Directory
Specify the working directory for launching the debuggee.