



Code Sample: Intel® Software Guard Extensions Remote Attestation End-to-End Example

By John P Mechalas

Published:07/04/2018 Last Updated:07/04/2018

File(s):	Download
License:	Intel Sample Source Code License Agreement
Optimized for...	
Operating System:	Windows® 10, Linux*
Hardware:	6th gen Intel® Core™ or later, Intel® Xeon® E3 v6
Software: (Programming Language, tool, IDE, Framework)	Windows*: Microsoft Visual Studio* 2015 Professional Edition or later, Intel® Software Guard Extensions SDK for Windows* (Intel® SGX SDK for Windows*) Linux*: gcc, Intel® Software Guard Extensions SDK for Linux* (Intel® SGX SDK for Linux*) All platforms: libcurl, OpenSSL* 1.1.0
Prerequisites:	C/C++ programming

Introduction

Today's real-world applications are increasingly tasked with handling sensitive data. A user's authentication credentials, confidential documents, or highly valued intellectual property are all examples of information that must be kept safe from exposure and disclosure to unwanted parties. Because security breaches can cause significant damage to a company's finances and reputation, there is strong demand for developing applications with the ability to guard and protect their secrets. Recent advances in hardware and software security are giving developers a number of tools and technologies to choose from in order to achieve this goal.

Intel® Software Guard Extensions (Intel® SGX) Mission

One such technology is Intel® Software Guard Extensions (Intel® SGX), which became available with the introduction of 6th-generation Intel® Core™ processors. Intel SGX allows an application to control its own security by creating guarded enclaves within the application space that are not vulnerable to inspection by malicious actors—even when attacks originate from privileged software, whether that be a compromised operating system, a virtual memory manager, or device drivers.

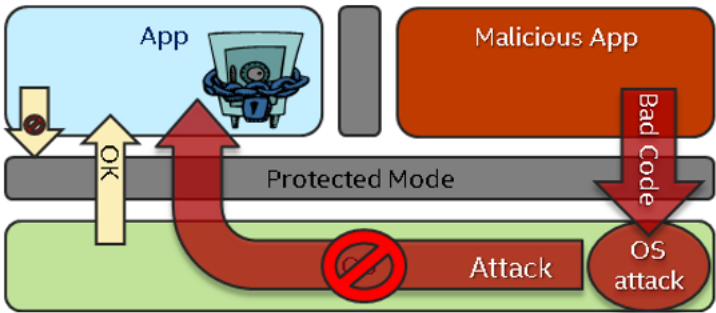





Figure 1. Intel® Software Guard Extensions (Intel® SGX) protection model

This article assumes that the reader is familiar with the basic concepts of Intel SGX technology. To learn more about it, please visit the Intel SGX landing zone.

USA (English)   

Provisioning Secrets with Remote Attestation

Utilizing Intel SGX in conjunction with other modern software and hardware technologies can help an application achieve the best possible protection for its existing secrets. But that's not the end of the story—those secrets are still vulnerable while in transit. Software vendors face a common problem that is difficult to address: How does the application obtain the secrets in the first place?

In some cases, a user may be asked to supply that information, but in the absence of trusted input/output on the platform, that input may be leaked to malware. Other sensitive content, such as media protected by digital rights management (DRM), may not even be user generated at all. This class of content may originate from a remote server, and even there, vulnerabilities abound: While the communication path between client and server may be encrypted, there is no guarantee that the client machine has not been compromised by malware.

These vulnerabilities make it potentially dangerous to trust the client machine with sensitive data. However, an advanced feature of Intel SGX called Remote Attestation can significantly increase the level of trust afforded to the client.

Using the Remote Attestation flow, a client's enclave can attest to a remote entity that it is trusted, and establish an authenticated communication channel with that entity. As part of attestation, the client's enclave proves the following:

- Its identity
- That it has not been tampered with
- That it is running on a genuine platform with Intel SGX enabled
- That it is running at the latest security level, also referred to as the Trusted Computing Base (TCB) level

At that point, the remote server can safely provision secrets to the enclave.

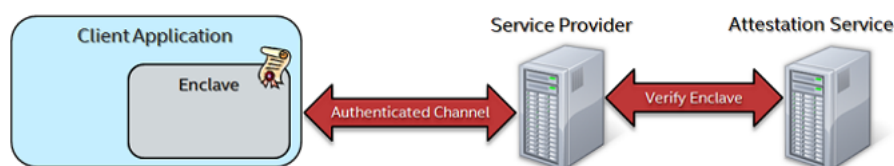


Figure 2. Remote Attestation at a glance

The rest of this article will focus on describing the Remote Attestation flow in detail through a new, end-to-end code sample that was developed at Intel. There are several reasons why this new sample was created:

- As shown in Figure 2, there are multiple components to Remote Attestation. After seeing the complete attestation flow in action, developers should have a clear understanding of how those components communicate with each other.
- While the Intel SGX SDK provides convenient wrappers for the entire attestation flow, including encapsulating cryptographic functionality, no such convenience exists for implementing the remote service provider. This sample shows how to create a service provider capable of facilitating Remote Attestation with a client, utilizing cryptographic routines from the OpenSSL* libraries.
- The server also communicates with the Intel® Attestation Service (IAS) to perform the actual enclave verification. The sample code demonstrates the IAS APIs used when performing attestation. Please refer to the Intel Attestation Service API document for details.
- The sample has been reduced to the bare minimum number of components necessary to implement a Remote Attestation flow with a thin client-server protocol layer. This allows developers to more easily focus on, and isolate, the Remote Attestation procedure.

Remote Attestation Flow

Prerequisites

In order to utilize the Intel IAS for Remote Attestation, Intel requires that the independent software vendor (ISV) obtain API keys from the Intel® SGX Attestation Service Utilizing Enhanced Privacy ID (EPID) (English) (USA) (France) (Germany) (Japan) (Korea) (Russia) (Taiwan) (UK) (India) (Brazil) (Canada) (China) (Mexico) (Poland) (Spain) (Sweden) (Switzerland) (Ukraine) (Vietnam) (Other). Certificate-based authentication will continue to be supported on the production server in order to minimize disruption to existing customers and there is no EOL date at this time.)

You can choose use either a linkable or unlinkable attestation policy. A linkable policy allows the ISV to determine whether multiple attestation requests originated from the same platform. Though this does not identify the individual platform, it does allow the ISV to determine whether multiple requests originated from the same platform.

The ISV will be issued a service provider ID (SPID) along with their API keys. This ID is used by the service provider when communicating with IAS, as described later.

Client-Server protocol

Remote Attestation utilizes a modified Sigma protocol to facilitate a Diffie-Hellman Key Exchange (DHKE) between the client and the server. The shared key obtained from this exchange can be used by the service provider to encrypt secrets to be provisioned to the client. The client enclave is able to derive the same key and use it to decrypt the secret.

Communication sequence

Provisioning request

The first step in Remote Attestation involves the client asking the service provider to provision secrets. This is usually a specific API endpoint that the service provider implements for making such a request. The service provider responds to this request by issuing a challenge requesting the client to attest itself.

The details of this handshake are left up to the ISV implementing its service.

Msg0 (client to server)

In response to the challenge request, the client performs several steps to construct the initial message of the Remote Attestation flow. Assuming the client's enclave has already been initialized, the untrusted code takes the following steps.

1. Execute an ECALL to enter the enclave
2. Inside the enclave:
 1. Call `sgx_ra_init()`
 2. Return the result and the DHKE context parameter to the untrusted application
3. Call `sgx_get_extended_epid_group_id()`

The function `sgx_ra_init()` accepts the service provider's public key as an argument and returns an opaque context for the DHKE that will occur during Remote Attestation. Using a context that is opaque to the client provides replay protection.

The format of the public key is set by the `sgx_ec25_public_t` data type in `sgx_tcrypto.h`. The EC key is represented as its x coordinate followed by its y coordinate:

```
1  typedef struct _sgx_ec256_public_t
2  {
3      uint8_t gx[SGX_ECP256_KEY_SIZE];
4      uint8_t gy[SGX_ECP256_KEY_SIZE];
5  } sgx_ec256_public_t;
```

These key coordinates must be in little-endian byte order.

The Service Provider's public key should be hardcoded into the enclave. This, combined with enclave signing, ensures that the key cannot be changed by end users so that the enclave can only communicate with the intended remote service.

If the enclave needs to access Platform Services, then the Platform Services Enclave (PSE) must be included in the attestation sequence. The PSE is an architectural enclave included in the Intel SGX software package that supplies services for trusted time and a monotonic counter. These can be used for replay protection during nonce generation and for securely calculating the length of time for which a secret should be valid.

To use the PSE, the procedure becomes:

1. Execute an ECALL to enter the enclave

2. Inside the enclave, perform these steps (in order):
1. `sgx_create_pse_session()`

2. `sgx_ra_init()`

3. `sgx_close_pse_session()`

4. Return the results and the DHKE context parameter to the untrusted application
3. Call `sgx_get_extended_epid_group_id()`

Note that enclaves running on server hardware do not have a Platform Services Enclave, and cannot utilize client specific features.

Regardless of whether or not the enclave makes use of Platform Services, it is up to the enclave writer to expose a wrapper function for the ECALL that executes `sgx_ra_init()` `sgx_ra_init()`

Upon a successful result from `sgx_ra_init()`, the client next makes a call to `sgx_get_extended_epid_group_id()` to retrieve the extended group ID (GID) of the Intel® Enhanced Privacy ID (Intel® EPID). Intel EPID is an anonymous signature scheme used for identification. A detailed discussion of Intel EPID is beyond the scope of this article; interested users can consult the following resource: Intel SGX: EPID Provisioning and Attestation Services

The extended GID is sent to the service provider as the body of msg0. The format of msg0 and details of the response from the server are left up to the service provider. However, the service provider must verify that the GID it received is supported and abort the attestation process if it is not. The Intel Attestation Service only supports the value of zero for the extended GID.

Msg0 and msg1 (see below) may optionally be sent together.

Msg1 (client to server)

Given a successful response to msg0, or if msg0 is to be sent in conjunction with msg1, the client calls the `sgx_ra_get_msg1()` function to construct the msg1 object containing the client's public key for the DHKE and sends it to the service provider. Additional parameters to this method include the DHKE context obtained in the previous step and a pointer to the `sgx_ra_get_ga()` stub function for computing the client-side DHKE secret. This function is automatically generated by the SDK when the enclave developer links in the `sgx_tkey_exchange` library and imports `sgx_tkey_exchange.edl` in the Enclave Definition Language (EDL) file.

The format of msg1 is given in Table 1.

Table 1. Structure of msg1

Element		Size
Ga	Ga _x	32 bytes
	Ga _y	32 bytes
Intel® EPID GID		4 bytes

The Intel SGX SDK defines the data structure for msg1 in `sgx_key_exchange.h`:

```
1 typedef struct _ra_msg1_t
2 {
3     sgx_ec256_public_t    g_a;
4     sgx_epid_group_id_t   gid;
5 } sgx_ra_msg1_t;
```

All components of msg1 are in little-endian byte order. The client function `sgx_ra_get_msg1()` returns msg1 with all elements in the correct byte order. The service provider, however, must convert the values from little-endian order when processing msg1.

Msg2 (server to client)

Upon receiving msg1 from the client, the service provider checks the values in the request, generates its own DHKE parameter, and sends a query to the IAS to retrieve the Signature Revocation List (SigRL) for the Intel EPID GID sent by the client.

To process msg1 and generate msg2, the service provider follows these steps:

1. Generate a random EC key using the P-256 curve. This key will become Gb.

2. Derive the key derivation key (KDK) from Ga and Gb:

1. Compute the shared secret using the client's public session key, Ga, and the service provider's private session key (obtained from Step 1), Gb. The result of this operation will be the x coordinate of Gab, denoted as Gab_x.

2. Convert Gab_x to little-endian byte order by reversing its bytes.

3. Perform an AES-128 CMAC on the little-endian form of Gab_x using a block of 0x00 bytes for the key.

4. The result of 2.3 is the KDK.

3. Derive the SMK from the KDK by performing an AES-128 CMAC on the byte sequence:

0x01 || SMK || 0x00 || 0x80 || 0x00

using the KDK as the key. Note that || denotes concatenation and “SMK” is a literal string (without quotes).

1. Determine the quote type that should be requested from the client (0x0 for unlinkable, and 0x1 for linkable). Note that this is a service provider policy decision, and the SPID must be associated with the correct quote type.

2. Set the KDF_ID. Normally this is 0x1.

3. Calculate the ECDSA signature of:

Gb_x || Gb_y || Ga_x || Ga_y

(traditionally written as r || s) with the service provider's EC private key.

4. Calculate the AES-128 CMAC of:

Gb || SPID || Quote_Type || KDF_ID || SigSP

using the SMK (derived in Step 3) as the key.

5. Query IAS to obtain the SigRL for the client's Intel EPID GUID.

The format of msg2 is shown in Table 1.

Table 2. Structure of msg2

Element		Size
A	Gb	
	Gb _x	32 bytes
	Gb _y	32 bytes
	SPID	16 bytes
	Quote_Type	2 bytes (uint16_t)
	KDF_ID	bytes (uint16_t)
	SigSP	
	Sig(Gb_Ga) _x	32 bytes
	Sig(Gb_Ga) _y	32 bytes
CMAC _{SMK} (A)		16 bytes
SigRL	SigRL_Size	4 bytes (uint32_t)
	SigRL_data	SigRL_Size bytes

As a reminder, the unsigned integer values (Quote_Type, KDF_ID, and SigRL_Size), the x and y coordinates of SigSP, and the x and y coordinates of Gb must be in little-endian byte order.

The Intel SGX SDK defines a data structure for msg2 in `sgx_key_exchange.h`:

1typedef struct _ra_msg2_t

2{

3sgx_ec256_public_tg_b;

4sgx_spid_tspid;

5uint16_tquote_type;

6uint16_tkdf_id;

7sgx_ec256_signature_tsign_gb_ga;

8sgx_mac_tmac;

9uint32_tsig_rl_size;

10uint8_tsig_rl[];

11} sgx_ra_msg2_t;

USA (English)

Again, refer to the Intel EPID article resource mentioned in the Msg0 section for a detailed explanation of the SigRL. This service provider's public key and the SigRL information is packaged into msg2 and sent to the client in response to the msg1 request.

Msg3 (client to server)

On the client side, when msg2 is received the application calls the `sgx_ra_proc_msg2()` function to generate msg3. This call performs the following tasks:

- Verifies the service provider signature.
- Checks the SigRL.
- Returns msg3, which contains the quote used to attest that particular enclave.

Two of the parameters passed to `sgx_ra_proc_msg2()` are `sgx_ra_proc_msg2_trusted` and `sgx_ra_get_msg3_trusted`. These are pointers to functions that are automatically generated by the edger8r tool, which means your enclave project must do the following:

- Link against the trusted service library (`libsgx_tservice.a` on Linux and `sgx_tservice.lib` on Windows)
- Include the following in the **trusted** section of the enclave's EDL file:

1include "sgx_tkey_exchange.h"

2

3from "sgx_tkey_exchange.edl" import *;

Part of the processing performed by `sgx_ra_get_msg2_trusted()` is obtaining an enclave quote. The quote includes a cryptographic hash, or measurement, of the current running enclave which is signed with the platform's EPID key. Only the Intel Attestation Service can verify this signature. It also contains information about the platform services enclave (PSE) on the platform, which the IAS will also verify.

The structure of msg3 is given in Table 3.

Table 3. Structure of msg3.




Element			Size
CMAC _{SMK} (M)			16 bytes
M	Ga	Ga _x	32 bytes
		Ga _y	32 bytes
	Ps_Security_Prop		256 bytes
	Quote		(436 + <i>quote.signature_len</i>) bytes

The Intel SGX SDK defines a data structure for msg3 in `sgx_key_exchange.h`:

```

1  typedef struct _ra_msg3_t
2  {
3      sgx_mac_t          mac
4      sgx_ec256_public_t g_a;
5      sgx_ps_sec_prop_desc_t ps_sec_prop;
6      uint8_t           quote[];
7  } sgx_ra_msg3_t;

```

USA (English)   

The structure of the quote is defined in `sgx_quote.h`:

```

1  typedef struct _quote_t
2  {
3      uint16_t          version;          /* 0 */
4      uint16_t          sign_type;        /* 2 */
5      sgx_epid_group_id_t epid_group_id; /* 4 */
6      sgx_isv_svn_t     qe_svn;           /* 8 */
7      sgx_isv_svn_t     pce_svn;          /* 10 */
8      uint32_t          xeid;             /* 12 */
9      sgx_basename_t    basename;         /* 16 */
10     sgx_report_body_t  report_body;      /* 48 */
11     uint32_t          signature_len;     /* 432 */
12     uint8_t           signature[];      /* 436 */
13 } sgx_quote_t;

```

Msg4 (server to client)

Upon receiving msg3 from the client, the service provider must do the following, in order:




1. Verify that Ga in msg3 matches Ga in msg1.
2. Verify CMAC_{SMK}(M).
3. Verify that the first 32-bytes of the report data match the SHA-256 digest of (Ga || Gb || VK), where || denotes concatenation. VK is derived by performing an AES-128 CMAC over the following byte sequence, using the KDK as the key:
0x01 || "VK" || 0x00 || 0x80 || 0x00
4. Verify the attestation evidence provided by the client.
 1. Extract the quote from msg3.
 2. Submit the quote to IAS, calling the API function to verify attestation evidence.
 3. Validate the signing certificate received in the report response.
 4. Validate the report signature using the signing certificate.
5. If the quote is successfully validated in Step 3, perform the following:
 1. Extract the attestation status for the enclave and, if provided, the PSE.
 2. Examine the enclave identity (MRSIGNER), security version and product ID.
 3. Examine the **debug** attribute and ensure it is not set (in a production environment).
 4. Decide whether or not to trust the enclave and, if provided, the PSE.
6. Derive the session keys, SK and MK, that should be used to transmit future messages between the client and server during the session. The client can simply call `sgx_ra_get_keys()`, but the server must derive them manually by performing an AES-128 CMAC over the following byte sequences, using the KDK as the key:
MK: 0x01 || "MK" || 0x00 || 0x80 || 0x00
SK: 0x01 || "SK" || 0x00 || 0x80 || 0x00
7. Generate msg4 and send it to the client.

Verifying the attestation evidence requires that the service provider submit the quote to IAS and obtain an attestation report. This report is signed by the IAS Report Signing private key, and the service provider must validate this signature using the IAS Report Signing public key.

The report itself contains response headers and a payload that contains the attestation status for the enclave and PSE manifest. A value of "OK" means that the component can be trusted. Any other value means that the component is either untrusted, or trustable if certain actions are taken (such as a software or BIOS update). Please refer to the IAS API document for full details. It is up to the ISV to pass these status messages on to the client.

If the attestation status is not "OK", the IAS may optionally send a Platform Info Blob (PIB), which the service provider may choose to forward to the client. The client can pass the PIB to the `sgx_report_attestation_status()` function to obtain more detailed information about the failure.

The format of msg4 is up to the service provider. It must, at minimum, contain:

USA (English)   

- Whether or not the enclave is trusted.
- Whether or not the PSE manifest is trusted, if a PSE manifest was submitted.

It may optionally contain:

- The PIB, if present.
- Secrets to provision to the enclave. Any secrets should be encrypted using a key derived from the KDK (see msg2).
- An enclave re-attestation timeout.
- The current time.
- Public keys of other servers the client needs to trust.

Figure 3 shows the full Remote Attestation flow, as described in the previous sections:

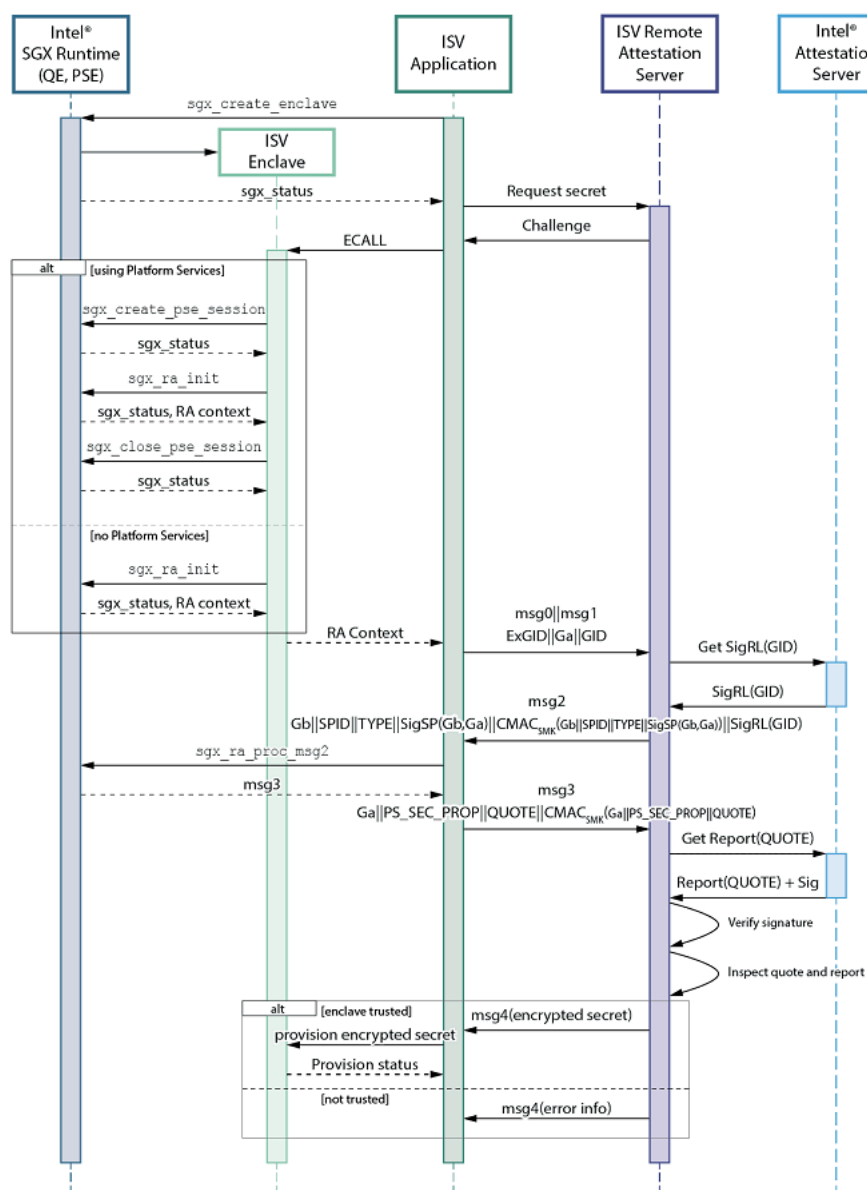




Figure 3. Full Remote Attestation flow.

USA (English)

Sample Code

The provided code sample includes both a client and service provider (server) component to demonstrate the concepts explained in this article. The code has been built and tested under:

- Ubuntu* Linux* 16.04
- Ubuntu* Linux* 18.04
- CentOS* Linux 7.5
- Windows® 10 64-bit with Microsoft Visual Studio* 2015 Professional Edition

This code is made available under the terms of the Intel Sample Source Code [license](#).

Runtime Requirements

Client

To successfully execute the client, it must be run on a system that supports Intel SGX and Intel SGX must be enabled. This means that the Intel SGX Platform Software package must be installed.

On Linux, the client must also have:

- OpenSSL 1.1.0. This sample was tested using OpenSSL 1.1.0h, built from source
On Windows*, the client must also have:
- OpenSSL 1.1.0 64-bit for Windows. The sample was tested using the Win64 OpenSSL 1.1.0h (the full, not the light) distribution from Shining Light Productions

Server

The server does not require Intel SGX (though building the server depends on header files from the Intel SGX SDK; see below).

On Linux, the server system requires:

- OpenSSL 1.1.0 (see the client requirements)
- One (or both) of the following packages from your Linux distribution
 - libcurl
 - wget

On Windows, the server must have:

- OpenSSL 1.1.0 64-bit for Windows (see the client requirements)

Build Requirements

Whether you build the client or server on Linux or Windows, the build system must have the Intel SGX SDK.

Client




On Linux, building the client requires:

- GNU Automake*
- gcc and g++
- OpenSSL 1.1.0 (see the runtime requirements) and headers

On Windows, building the client requires:

- Microsoft Visual Studio 2015 (Professional Edition or greater)

- [OpenSSL 1.1.0 64-bit for Windows](#) (see the runtime requirements) and headers

 USA (English)   

Server

On Linux, building the server requires:

- GNU Automake
- gcc and g++
- OpenSSL 1.1.0 (see the runtime requirements) and development headers
- (Optional) libcurl libraries and development headers (built against either Network Security Services or OpenSSL 1.0.x)

On Windows, building the server requires:

- Microsoft Visual Studio 2015 (Professional Edition or greater)
- OpenSSL 1.1.0 64-bit for Windows (see the runtime requirements) and headers

Running the Samples

The sample applications are a very simple client and server. By default, the server listens on port 7777 and the client connects to port 7777 on localhost. They are executed via wrapper scripts named **run-server** and **run-client** (shell scripts on Linux, and batch/CMD files on Windows) that parse the **settings** file and supply the necessary command-line arguments to the underlying executables, **client** and **sp**.

Simple usage is:

```
1 run-server [ options ] [ port ]
2 run-client [ options ] [ host:[port] ]
```

By supplying the **-z** option, you can force both the client and server to run in interactive mode. In this mode, they output their messages to stdout and take input on stdin. Thus, you can copy output from the client and paste it to the server, and vice versa. This allows you to see the attestation process one step at a time (as well as communicate between a client and server that may not have direct network access to one another).

The settings file is heavily documented. It, too, uses the shell syntax on Linux and the batch/CMD syntax on Windows.

Both the client and server generate log files (**client.log** and **sp.log** in the current directory) of the session. They also support a verbose and debug mode (settable via the settings, or using **-v** and **-d**, respectively).

Note that the server is very simplistic: it processes one connection at a time, serially. The communication protocol is a clear-text, raw transmission of the messages, encoded in base 16 (hex strings) for readability. There is minimal error checking on the client-server communication. The emphasis in this sample is on the Remote Attestation procedures, and it's assumed that service providers in real-world applications will use their existing infrastructure (for example, a representational state transfer (REST)-based service) to manage the protocol details.

Further Reading

Interested readers should take advantage of the following Intel SGX resources available on the Intel Developer Zone:

[Intel SGX Homepage](#) . Contains links to the SDK landing page (see below), access to the development services, an online user's guide, and the resources library. The latter is a treasure trove of informational articles and blog posts about Intel SGX. Some of the more useful are:

[Intel SGX: EPID Provisioning and Attestation Services](#)

[Innovative Technology for CPU Based Attestation and Sealing](#)

[Intel SGX Product Licensing](#)

[Intel SGX Attestation Signature Policy](#)

[Intel SGX SDK Landing Page](#) . Here you will find links to download the Intel SGX SDK and platform software. Also check out the FAQ section for answers to a number of common questions as well as additional useful resource links.



Product and Performance Information

USA (English)   

¹ Performance varies by use, configuration and other factors. Learn more at www.intel.com/PerformanceIndex

[Give Feedback](#)

Manage Your Tools

[Product Support](#)

[Product Downloads](#)

[Product Forums](#)

[License & Renewal FAQ](#)

[Intel® DevCloud](#)

Forums

[Intel® Active Management Technology SDK](#)

[Intel® Collaboration Suite for WebRTC](#)

[Intel® Software Guard Extensions](#)

[Useful Packages & Modules](#)

Company Information

[Our Commitment](#)

[Diversity & Inclusion](#)

[Communities](#)

[Investor Relations](#)

[Contact Us](#)

[Newsroom](#)

[Jobs](#)

Topics

[Artificial Intelligence](#)

[Game Development](#)

[Internet of Things](#)

[Data Parallel C++](#)



[© Intel Corporation](#)

[Terms of Use](#)

[*Trademarks](#)

[Privacy](#)

[Cookies](#)

[Supply Chain Transparency](#)

[Site Map](#)

USA (English)



Intel technologies may require enabled hardware, software or service activation. // No product or component can be absolutely secure. // Your costs and results may vary. // Performance varies by use, configuration and other factors. // See our complete legal [Notices and Disclaimers](#). //

Intel is committed to respecting human rights and avoiding complicity in human rights abuses. See Intel's [Global Human Rights Principles](#). Intel's products and software are intended

only to be used in applications that do not cause or contribute to a violation of an internationally recognized human right.

