Microsoft

Leveraging Attestations with Trusted Execution Environment (TEE) based applications on Azure (and on the Edge)

A starter guide for developers

Version 2.0 (Draft), July 2020

For the latest information about Azure, please see
https://azure.microsoft.com/en-us/overview/

For the latest information on Azure Confidential Computing (ACC), please see
https://azure.microsoft.com/en-us/solutions/confidential-compute/

For the latest information about open source on Azure, please see
https://azure.microsoft.com/en-us/overview/choose-azure-opensource/

For the latest information on the Open Enclave (OE) SDK, please see
https://openenclave.io/sdk/

This page is intentionally left blank.

Table of contents

Notice

This guide for developers is intended to illustrate how to leverage attestation along with so-called Trusted Execution Environment (TEE) based applications using the Microsoft Open Enclave SDK in C and C++. The Microsoft Open Enclave SDK (OESDK) is available in open source at https://openenclave.io/sdk/.

It features for that purposes the Intel® Software Guard Extensions (Intel® SGX) technology that comes with the latest generation of Intel Xeon families of processors for hardware-based TEE.

MICROSOFT DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, IN RELATION WITH THE INFORMATION CONTAINED IN THIS WHITE PAPER. The white paper is provided "AS IS" without warranty of any kind and is not to be construed as a commitment on the part of Microsoft.

Microsoft cannot guarantee the veracity of the information presented. The information in this guide, including but not limited to internet website and URL references, is subject to change at any time without notice. Furthermore, the opinions expressed in this guide represent the current vision of Microsoft France on the issues cited at the date of publication of this guide and are subject to change at any time without notice.

Publication: July 2020
Version 2.0

About this guide

Welcome to the **Leveraging Attestations with Trusted Execution Environment (TEE) based applications on Azure** starter guide for developers.

This document is part of a series of guides that covers confidential computing (with Trusted Execution Environments[1] (TEE)) in the Cloud, and the Edge, and considerations that pertain to it from a development perspective and/or an infrastructure one. This series of guides is available https://aka.ms/CCDevGuides.

In this starter guide, and as its title suggest, the basics of attestations with TEE-based application development will be covered.

The concept of TEE is given in the document Trusted Execution Environment: What It is, and What It is Not[2]:

"Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties."

As such, only authorized code is permitted to run and to access data, so data is protected against viewing and modification from outside of TEE, a.k.a. enclave. As of this writing, multiple TEE architectures, technologies and platforms[3] are available whether they are hardware based or software based - you can read the guide **Building and Executing Trusted Execution Environment (TEE) based applications on Azure** in this series of guides to learn about TEE -.

The Intel® Software Guard Extensions (Intel® SGX) technology[4], as available with the latest generation of Intel Xeon with the latest generation of Intel Xeon families of processors, is one of them for hardware based TEE. Intel SGX is a set of extensions to the Intel CPU architecture that aims to provide integrity and confidentiality guarantees to sensitive computation performed on a computer, where all the privileged software (kernel, hypervisor and so on) might potentially be compromised.

The above definition highlights the important points that are the separation between two spaces, i) the space of the OS and ii) that of the TEE controlled by what is designated as a "separation kernel", the guarantee of integrity of the code and confidentiality of data in the TEE, but adds the centrally important notion of an **attestation**. This function of attestation allows an external code to ensure that it is communicating with a real, expected TEE and that this TEE it is loaded without any alteration with the code and data provided to it.

One of the interesting capabilities of the Intel SGX platform of course notably resides in its ability to attest TEE, a.k.a. SGX enclave. This capability will be covered in detail with all the considerations that pertain to the keys' hierarchy, the related services in place to sustain all the guarantees conveyed but the notion of attestation.

As also covered by the above guide in this series, the Microsoft Open Enclave SDK[5] (OESDK), i.e. an open source framework available on GitHub over two years, aims at creating a single unified TEEs' abstraction and a consistent API surface for developers to build applications once that run across the multiple TEE "flavors" that exist – it

---

[1] Trusted execution environment: https://en.wikipedia.org/wiki/Trusted_execution_environment

[2] Trusted Execution Environment: What It is, and What It is Not, M. Sabt, M. Achemlal, A. Bouabdallah, 18 December 2015 https://hal.archives-ouvertes.fr/hal-01246364/file/trustcom_2015_tee_what_it_is_what_it_is_not.pdf

[3] The future of computing: intelligent cloud and intelligent edge: https://azure.microsoft.com/en-us/overview/future-of-cloud

[4] Intel Software Extension Guard: https://software.intel.com/en-us/sgx

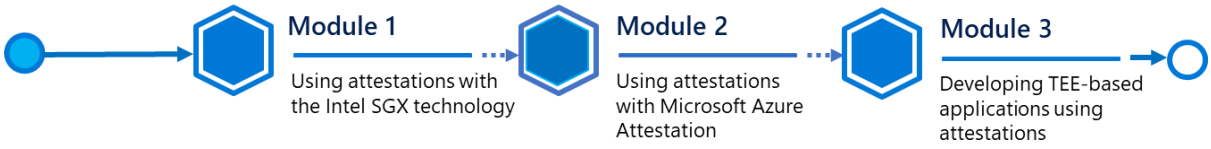
[5] Open Enclave SDK: https://Open Enclave.io/sdk/

unsurprisingly supports the Intel SGX amongst others. It thus offers a universal secure application model that minimizes platform specificities. It unsurprisingly supports the Intel SGX. TEE OE is Azure's de rigueur for designing customized secure apps for deployment on Confidential Computing VMs with Azure Confidential Computing[6] (ACC).

So, you will also learn how to use of attestation through "its lenses" to ease your own development of such applications. (Microsoft views it as an essential stepping-stone toward democratizing enclave technologies such as Intel SGX and increasing their uptake on Azure.)

For that purposes, you're invited to follow a short series of modules, each of them illustrating a specific aspect of attestations for the TEE-based application development, in terms of core understanding of some underlying important principles and concepts, or practical coding experience to get your hands a bit dirty with nitty-gritty details of a working implementation.

Each module within the guide builds on the previous. You're free to stop at any module you want, but our advice is to go through all the modules.



At the end of the starter guide, you will be able to:

- Understand how both local and remote attestations work with the Intel SGX technology, and what all this implies in terms of key hierarchy and infrastructure.
- Understand what benefits the new Microsoft Azure Attestation (MAA) service will provide in this space.
- Leverage an MAA instance, i.e. an attestation provider in Azure, for your TEE-based applications in the Cloud, and on the Edge.
- Use attestation as part of your TEE-based applications, whether it is about using Intel local and remote attestations, or ones delivered by an attestation provider.

Guide elements

In the starter guide modules, you will see one or all of the following elements:

- **Important concepts and principles**. An explanation of some of the concepts important to the procedures in the module, and what happens behind the scenes.
- **Step-by-step directions**. Click-through instructions – along with relevant snapshots – or links to online documentation for completing each procedure or part.

[6] Azure Confidential Computing: https://azure.microsoft.com/en-us/solutions/confidential-compute/

- **Sample applications, and files**. A downloadable or cloneable version of the project containing the code that you will use in this guide, and other files you will need. **Please go to the following repos on GitHub to download or clone all necessary assets:**
    - **https://github.com/openenclave/openenclave**
    - **https://github.com/microsoft/azure-tee-attestation-samples/**

Guide prerequisites

To successfully leverage the provided code in this starter guide, you will need:

- A Microsoft account[7].
- An Azure subscription. If you don't have an Azure subscription, create a free account[8] before you begin.
- A Windows 10 local machine.
- A code editor of your choice, such as Visual Studio[9] or Visual Studio Code[10], with C++ for Linux and Open Enclave installed. The related installation and configuration will be further covered later in this guide.
- A terminal console for your Windows 10 local machine, which allows you to remotely connect to a virtual machine (VM) in SSH, such as PuTTY[11], Git for Windows[12] (2.10 or later).

Important note With Git, ensure that long paths are enabled: `git config --global core.longpaths true`.

Note Recent versions of Windows 10 provide OpenSSH client commands to create and manage SSH keys and make SSH connections from a command prompt. For more information, see blogpost What's new for the Command Line in Windows 10 version 1803[13].

[7] Microsoft Account: https://account.microsoft.com/account?lang=en-us

[8] Create your Azure free account today: https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F

[9] Visual Studio: https://visualstudio.microsoft.com/

[10] Visual Studio Code: https://code.visualstudio.com/

[11] PuTTY: https://www.chiark.greenend.org.uk/~sgtatham/putty/

[12] Git for Windows: https://git-for-windows.github.io/

[13] What's new for the Command Line in Windows 10 version 1803: https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/

Using attestations with the Intel SGX technology

Introducing the attestation process

The attestation process consists in assuring a client application which wants to execute a code inside an enclave that it is communicating with a genuine enclave and that the latter is executing the code unmodified. Indeed, the platform that implements the enclave – for example the Intel SGX processor – must be able to prove that it is not a software emulating an enclave and also guarantee the integrity of the code that has been loaded into the enclave for processing.

This guarantee is essential for the client application before starting to communicate with the enclave and transfer sensitive data to be processed or secrets (for example cryptographic keys).

The attestation mechanism is provided by the platform on which are executed the enclaves. The attestation is a data structure including characteristics of the enclave and a cryptographic proof which confirms the link with the platform (TCB – Trusted Computing Base binding) hosting the enclave. Among the characteristics of the enclave is a measurement of the code loaded during the creation of the enclave.

For the attestation to be considered trustworthy, the client must trust the platform hosting the enclaves. The trusted root, for example in the case of the Intel SGX implementation, is based on the presence of two root keys fused in the processor itself, these two generated keys being specific to each processor and integrated during the processor construction stage.

The attestation can take two flavors: local or remote. In the case of local attestation, two enclaves running on the same processor want to make sure that each one communicates with another genuine enclave. The platform (the processor) implements the mechanism for providing these attestations through processor instructions. The implementation by the platform relies on cryptographic keys and special "architectural" enclaves.

In the remote attestation scenario, an application wants to make sure that it communicates with a genuine remote enclave. The mechanism is more complex since the client application does not necessarily run in an enclave and that the processor or platform boundary is crossed: for example the client application runs on an on-prem computer and is requesting to process data in an enclave created in a server in the cloud.

Through the attestation mechanism, it is possible to transfer data – only a few bytes in the case of Intel SGX technology – while ensuring their integrity. This functionality will be used in the implementation examples, later in the document, to exchange encryption keys, then allowing encrypted data to be transferred to an enclave.

The following paragraphs detail these mechanisms as implemented by Intel SGX processors.

Note For a definition of attestation, see article Intel® Software Guard Extensions Developer Guide : Attestation[14].

[14] Intel® Software Guard Extensions Developer Guide: Attestation https://software.intel.com/en-us/node/702982

A first look at local attestation

The principle of local attestation consists, for an enclave, in ensuring that the enclave with which it is communicating is indeed an enclave running on the same processor and that the state of this enclave has not been modified since its creation.

Therefore, it is first of all necessary to have a guarantee of the enclave integrity obtained by a digest (SHA256) on a set of information characteristics of the enclave following its creation (code, data, stack, heap…). This digest is called `MRENCLAVE` and defines a kind of enclave identity card.

The attestation process involves the exchange between two enclaves of a structure called `REPORT` describing the characteristics of the enclave which must prove its genuineness. This `REPORT` contains attributes of the enclave, its measurement (`MRENCLAVE`) and a User Data field (`report_data`) of limited size but available for a particular use, for example to exchange a key or a hash.

The attestation process is implemented by the processor and is based on two instructions:

- The `EREPORT` instruction allows an enclave to ask the processor to generate its attestation structure (the report).
- The `EGETKEY` instruction allows an enclave to access its own `ReportKey` which will be used to validate the attestation report.

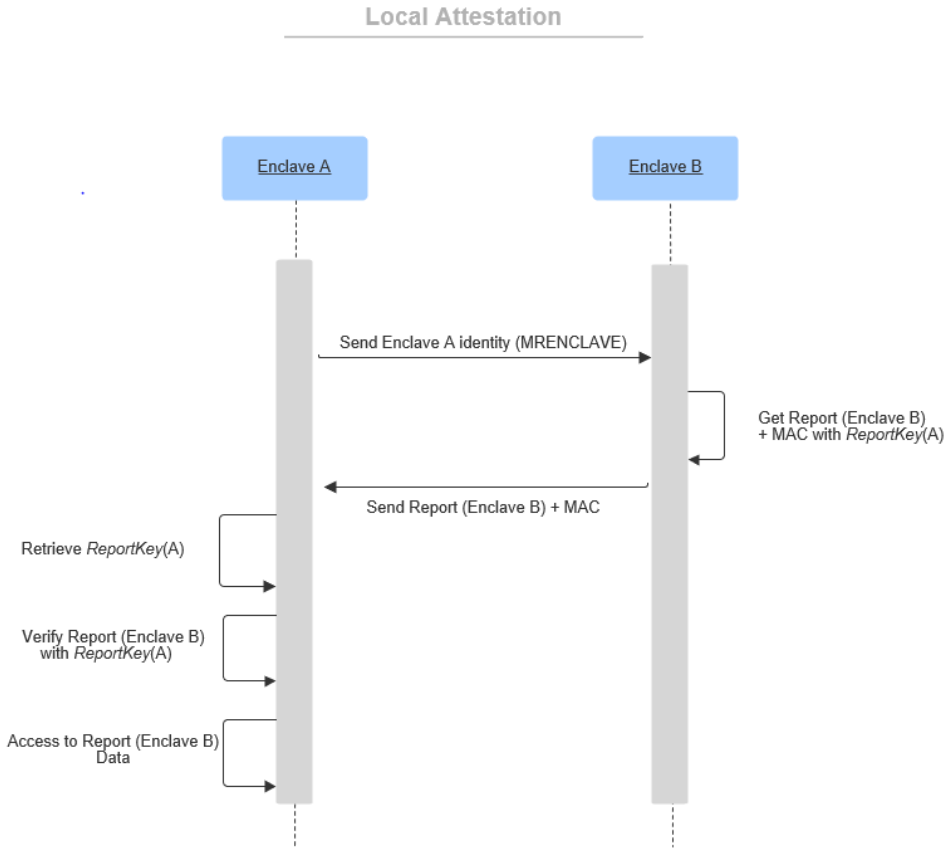


Figure 1 Local Attestation workflow

In the flow diagram above, Enclave A wants to ensure the authenticity of Enclave B.

- Enclave A transmits its identity (the value of the hash `MRENCLAVE`) to Enclave B. The transmission channel does not need to be secure and can be done through an untrusted part of an application.
- Enclave B requests to the processor, through the `EREPORT` instruction, to generate its `REPORT` targeting Enclave A by providing the value of `MRENCLAVE` identifying the latter. The `EREPORT` instruction uses a symmetric key linked to enclave A, the `ReportKey` (A), to calculate a hash on the `REPORT` structure. More specifically, it is a Message Authentication Code (MAC also known as keyed hash) on the `REPORT` structure. By design, the `EREPORT` instruction executed from the target enclave (Enclave B) has access to the symmetric key of the source enclave (Enclave A).
- Then the `REPORT`, including the MAC, is retransmitted to enclave A. Enclave A recover its own `ReportKey` (A) through the `EGETKEY` instruction.
- The `REPORT` is then verified by recalculating the associated MAC and using the symmetric shared key `ReportKey` (A).
- If the verification is successful, Enclave A has certified the authenticity of Enclave B
    - Enclave B is a genuine enclave that runs on the same processor.
    - The characteristics of Enclave B have not been modified since its creation and more particularly the code that runs in the enclave.
- Enclave A can access the data included in the `REPORT` including the report_data field which allows the transmission of a few bytes of information coming from Enclave B.

Note Unlike a simple hash that does not use a key, a Message Authentication Code (MAC also known as a keyed-hash) uses a symmetric key which guarantees not only the integrity of the message on which it is calculated but also its authenticity (that is to say the assurance that the sender is in possession of the symmetric key). For more information, see Message authentication code[15] on Wikipedia.

Note The `ReportKey` is a key derived from the Root Seal Key though the `EGETKEY` instruction. This key is specific to the enclave because it is derived using characteristics of the enclave such as the value of `MRENCLAVE` or its attributes. For more information on the key hierarchy, see section § *A first look at the Root of Trust and the Key hierarchy*.

It is important to note that the local attestation process as described above must take place between two enclaves. In fact, the client part who wants to ensure that it is communicating with a genuine enclave, must be able to use instructions that are only accessible from an enclave. In the Intel SGX implementation, only one enclave can generate a Report, so the attestation of a target enclave can only be done from an enclave.

Knowing that this scenario typically relates to two enclaves that want to cooperate, it would be logical that Enclave B can ensure the authenticity of enclave A using the same attestation process.

A first look at remote attestation

The local attestation scenario remains limited to certifying enclaves which run on the same platform (the same processor). This scenario is interesting because it allows several enclaves to interconnect or leverage each other to perform different and isolated tasks.

[15] Message Authentication Code: https://en.wikipedia.org/wiki/Message_authentication_code

In addition, Intel has defined in its design, so-called "architectural" enclaves that can perform processing related to the implementation of more complex scenarios. Among these processing natively included in the platform; is the *Quoting Enclave* whose role is to implement the remote attestation mechanism.

Remote attestation makes it possible to cover a common scenario where a client side addresses a server application which implements a service relying on one or more processes executing in enclaves. This service must be able to prove to the client, through the remote attestation mechanism, that the specific processing operations are carried out inside authentic enclaves.

This scenario is particularly interesting in the context of a processing which must be executed in a secure manner in a public cloud. Servers equipped with SGX processors are hosted in the cloud provider's data centers (Microsoft Azure) but will be able to perform opaque processing on encrypted data – that is to say without the cloud provider being able to access data in clear-text, processing or results.

From an implementation point of view, the local attestation relies on the use of a symmetric key shared between the two enclaves (the `ReportKey` generated by the processor). The `ReportKey` is used to sign the `REPORT` and to allow the target enclave to verify the attestation. However, this attestation principle must be extended beyond attesting enclaves on the same processor to implement a remote attestation.

This type of attestation allows a remote client running on another machine (another platform without the need to be an SGX processor) to communicate with the machine running the enclaves by asking it to prove that it is a genuine SGX platform running authentic enclaves.

The implementation is based on the use of the Quoting Enclave whose role is to check the REPORTs provided by the other local enclaves and to make them "exportable" by signing them with a private key linked to the processor. The result is a "Quote", which can be verified by the requestor by leveraging an attestation service in possession of the public key.

The private key, called the EPID key (Enhanced Privacy ID, registered trademark by Intel), is linked to the physical processor as well as its firmware version. Only the Quoting Enclave has access to the EPID Key and can sign the quotes.

The magic about the EPID key is that, although unique for each processor, it belongs to a group of other private EPID keys that are associated with the same public key named EPID Group Public key. This means that the attestation service, with the sole indication of the group, can verify the "Quote" with the group's public key while respecting the privacy of the platform that signed the Quote (see also section § *A first look at the Root of Trust and the Key hierarchy* below).

> **Note** Intel has made the choice to not publish public keys directly but to make verification available through its attestation service, Intel Attestation Service.

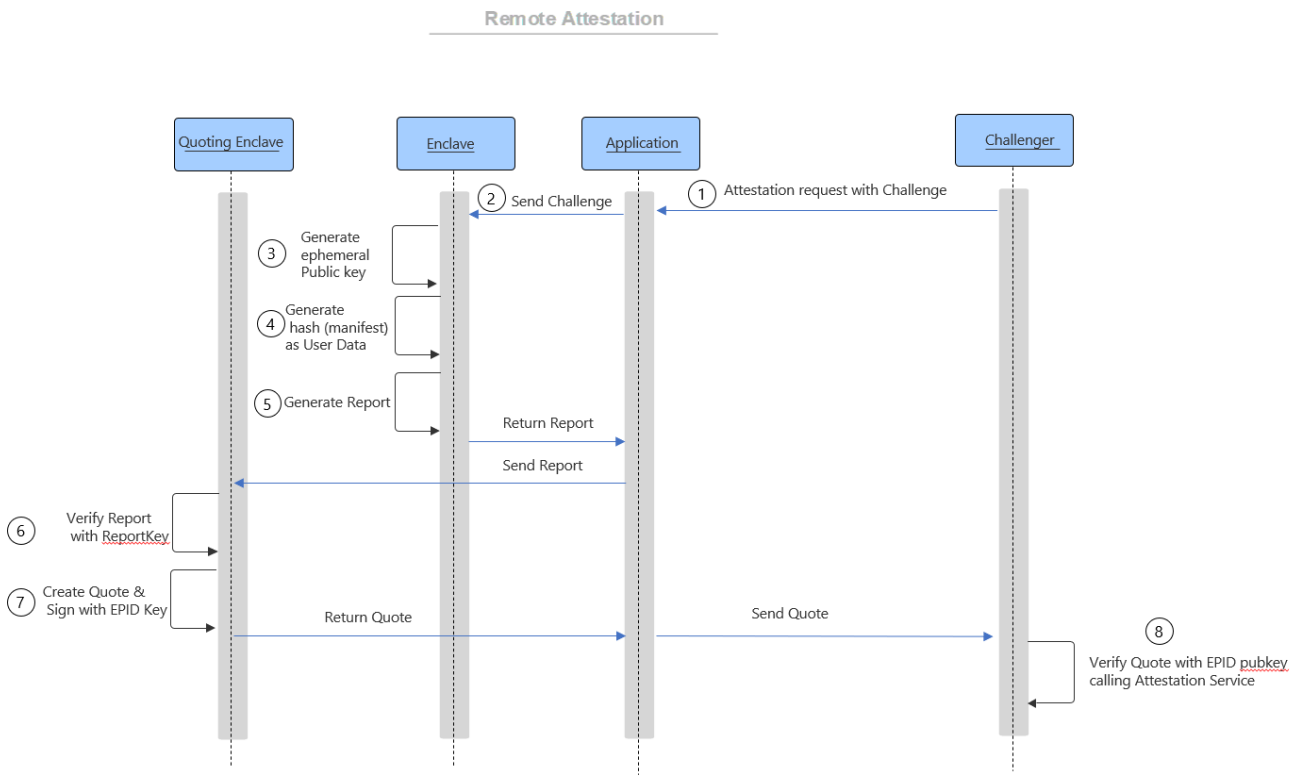


Figure 2 Remote Attestation workflow

The above schema details the flow of the remote attestation process. The challenger is the remote client who wants to ensure the authenticity of the application before sending data for processing. The application runs on the server equipped with the SGX processor on which the enclave is created and will have to execute secure processing. The Quoting Enclave is the architectural enclave that generates the Quote.

1. The challenger establishes a connection with the application and sends an attestation request in which a nonce can be added to avoid replay attacks.
2. The application transmits the request to the enclave.
3. The enclave generates a key pair (ephemeral) whose public key will be returned to the challenger to allow the transfer of secrets (for example a symmetric key used to encrypt data)
4. The enclave generates a hash of the manifest to include in the `report_data` field of the `REPORT`.
5. The enclave calls the `EREPORT` instruction to generate a `REPORT` including the hash of the manifest and targeting the Quoting Enclave (the application has passed the identity of the Quoting Enclave).
6. The application retrieves the `REPORT` structure and sends it to the Quoting Enclave which retrieves its `ReportKey` to check the `REPORT`.
7. The Quoting Enclave creates the Quote structure including the `REPORT` information and signs it with its private EPID key before returning it to the application. Then the application returns the Quote to the challenger.

8. The challenger calls the attestation service to verify the validity of the Quote using the public key EPID. If the answer is positive, the challenger is guaranteed that it is indeed communicating with enclaves on an SGX platform and that information in the Quote can be trusted.

Note that this is a description of a simple implementation example to explain the principle of remote attestation.

Note For more information, see article Innovative Technology for CPU Based Attestation and Sealing[16],

A first look at the Root of Trust and the Key hierarchy

The previous section described the local certification mechanism which relies on a symmetric key shared between the two enclaves, the `ReportKey`. It is the platform in the broad sense – the processor and the architectural enclaves – which is responsible for providing this key to the two enclaves:

1. to allow the enclave which needs to generate the attestation for the target enclave, to sign the Report through the `EREPORT` instruction (the `EREPORT` instruction has access to the `ReportKey` of the target enclave),
2. to make it possible for the target enclave to verify the Report by retrieving its own `ReportKey` through the `EGETKEY` instruction.

The question is: "how is this key generated and why can we trust it?". It is not difficult to guess that this key is a derived key since it is unique for each enclave. To understand how is established the trust in the hardware platform, we must go up the chain of trust which is detailed in the diagram below.

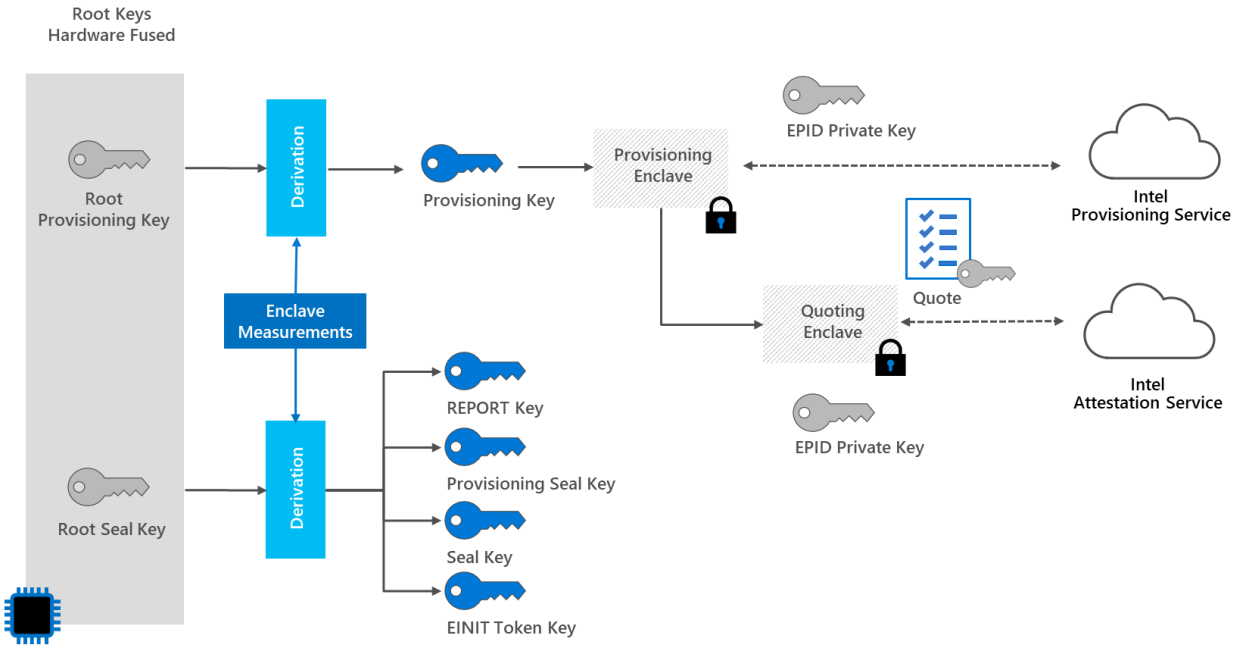


Figure 3 Key Hierarchy

[16] Innovative Technology for CPU Based Attestation and Sealing: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing

Note A subtlety has crept into this hierarchy: the Provisioning Seal Key derives from the 2 root keys: the Root Seal Key but also the Root Provisioning Key.

The chain of trust hierarchy begins with 2 root keys: i) the Root Provisioning Key (RPK), and ii) the Root Seal Key (RSK) which are both integrated into the hardware during the processor manufacturing.

The RPK is a randomly generated key and is kept in a HSM of the Intel company: it constitutes the root of trust of the SGX processor which can prove that it is referenced as Intel hardware. The RSK is also created during manufacturing but is not stored by Intel. The RSK is unique to the processor and could be compared to a serial number.

The Root Provisioning Key is used to support the remote attestation mechanism (upper part of the diagram).

The Root Seal Key is used mainly in the sealing functions (i.e. the protection of the data of an enclave when it dies, and when it is re-instantiated later on) and to generate the Report Key which, as we have seen, is the basic building block of local attestation.

Key derivation mechanism

The derivation mechanism is supported by the `EGETKEY` instruction which makes it possible to produce versions of each type of key specific to each enclave by taking into account the software characteristics specific to each enclave. These characteristics, or Enclave Measurements, include for example the `MRENCLAVE` and `MRSIGNER` hashes or the CPU Security Version Number (CPUSVN).

All keys are derived from the two root keys, the Root Provisioning Key and the Root Seal Key.

Note Not all keys are derived with the same characteristics or attributes of the enclave and software versions of the processor. For a precise correspondence table, see table 3: SGX Key Properties of the document Intel Software Guard Extensions : EPID Provisioning and Attestation Services[17].

Derived keys

The derived keys linked to each enclave have different functions. Four keys derive from the Root Seal Key:

- The Report Key which is used for the local attestation mechanism.
- The Seal Key which allows to protect secrets outside the enclave when it stops executing and to be able to recover them when it is recreated.
- The Provisioning Seal Key used during the EPID key generation process to encrypt it before its transmission to the Intel Provisioning Service (EPID private Key Escrow).
- The EINIT Token Key (also referenced as Launch Key) used to authenticate the initialization token during the creation process of the enclave.

Finally, the Provisioning Key derives only from the Root Provisioning Key to implement the remote attestation process.

It is interesting to note that all keys except the Provisioning Key derive from the Root Seal Key, which implies that they cannot be known to Intel.

[17] Intel Software Guard Extensions: EPID Provisioning and Attestation Services: https://software.intel.com/en-us/sgx/attestation-services

EPID private key and remote attestation

The EPID key is not a derived key but the private key of a key pair generated by the Provisioning Enclave during a dialogue with the Intel Provisioning Service in an initialization phase of the platform.

The EPID key is then transmitted to the Quoting Enclave. The Quoting Enclave uses this key to sign the Quotes submitted to the Intel remote Attestation Service, which uses the associated EPID public key to validate the Quotes.

This EPID private key has the particularity of being associated with a group public key (referenced as EPID Group Public key) which provides anonymization in the attestation request. Indeed, the EPID key generation protocol used makes it possible to associate a set of private EPID keys with a single public key of the EPID group. As a result, the attestation service can validate the quotes it receives with the group public EPID key, but is not able to know which private key and therefore which processor signed the quote.

> **Note** For more information, see section § 3.2.1 Intel Enhanced Privacy ID (EPID) of the document Innovative Technology for CPU Based Attestation and Sealing[18], and section § 3 Intel Enhanced Privacy ID of the document Intel Software Guard Extensions : EPID Provisioning and Attestation Services[19].

> **Note** The document published on the Blackhat website by Yogesh Swami describes in detail the use of the different keys and the mechanism linked to the EPID key. See document Intel SGX Remote Attestation is not sufficient[20].

Interacting with the Intel Attestation Service

For the moment, we have approached the mechanism linked to attestation from the SGX platform point of view on which enclaves are executed, and more particularly how local attestations can be extended to remote attestations through two elements:

- The Quoting Enclave which is responsible for verifying local attestations and transforming them into quotes
- The Attestation service, which role is to certify to the calling entity it that the quotes it sends for verification are valid.

For example, we saw in the previous scenario that the quote was transmitted by the Quoting enclave to the challenger who will submit it to the attestation service for validation.

If we refer to the hierarchy of keys, we remember that the signature of the quote is made by the private key EPID and that this key is associated with a public group EPID key in the possession of Intel. This key pair is generated when the platform is initialized by a dialog between the Quoting Enclave and the Intel Provisioning Service[21].

---

[18] Innovative Technology for CPU Based Attestation and Sealing: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing

[19] Intel Software Guard Extensions: EPID Provisioning and Attestation Services: https://software.intel.com/en-us/sgx/attestation-services

[20] Intel SGX Remote Attestation is not sufficient: https://www.blackhat.com/docs/us-17/thursday/us-17-Swami-SGX-Remote-Attestation-Is-Not-Sufficient-wp.pdf

[21] See previous section § *EPID private key and remote attestation* in the document.

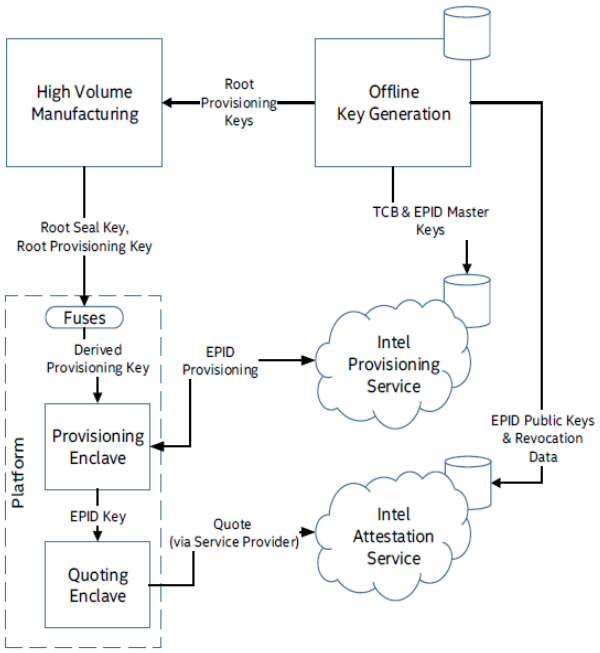


Figure 4 Intel SGX Infrastructure Services[22]

The diagram above, extracted from the Intel documentation, highlights the two infrastructure services provided by Intel:

- The Intel Provisioning Service responsible for the dialogue with SGX platforms to generate the EPID key pairs.
- The Intel Attestation Service responsible for validating quotes during remote attestation requests, in addition to revocation data.

Note that the EPID group public keys are transferred to the Attestation Service in charge of checking the integrity of the quotes which are submitted to it.

Registering and connecting to the Intel Attestation Service

The attestation service is a service provided by Intel in the form of a web service in a cloud environment. It is accessible from the internet but requires prior registration from the portal before it can be requested. Registration is done from the Intel Registration Portal[23].

Two types of subscription are possible depending on whether you want to use the service for testing purposes or for a production platform. In the first case, the subscription is free, in the second case, a paid user license is required.

During registration, a Service Provider ID is assigned (Service Provider ID or SPID) as well as an authentication key (Subscription key). The SPID must be indicated in each Quote structure submitted for validation, and the

[22] Intel Software Guard Extensions: EPID Provisioning and Attestation Services, Chapter 4 SGX Infrastructure Services: https://software.intel.com/en-us/sgx/attestation-services

[23] Explore EPID Attestation to Enhance Enclave Security: https://api.portal.trustedservices.intel.com/EPID-attestation

subscription key, which must appear in the header of each request, will be used as a shared secret key to authenticate with the Intel attestation service.

All communications with the service are encrypted using the HTTPS protocol (TLS) with client and server authentication.

Sending quote attestation request

The web service endpoint is different depending on whether you are working with the service with:

- A development subscription: https://api.trustedservices.intel.com/sgx/dev

-or-

- A production subscription: https://api.trustedservices.intel.com/sgx

The dialogue with the attestation service is done through 2 REST APIs:

1. One to verify the attestation (Verify Attestation Evidence).
2. And the other to retrieve the revocation list for a particular EPID group (Retrieve SigRL).

The call to the attestation verification API takes as input the structure corresponding to the Quote which was transmitted to the challenger by the Quoting Enclave. The attestation service verifies the integrity of the attestation signed by the private key EPID of the platform executing the enclave, checks that this key is not revoked and returns an Attestation Verification Report including the initial Quote structure.

This gives the challenger the confidence that a particular code is performing in a real enclave of an Intel certified processor. It remains, however, for the challenger to check the identity of the enclave with which it then wants to order a processing.

> **Note** For more information, see Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation[24], and section § 4.5 The Intel Attestation Service (IAS) of the document Intel Software Guard Extensions : EPID Provisioning and Attestation Services.

Using Data Center Attestation Primitives (DCAP)

The remote attestation mechanism described above is supported by the IAS service provided by Intel which requires that each platform (processor) has an internet access to Intel services: with the Intel Provisioning Service for negotiation leading to the creation of the EPID key, then with the Intel Attestation Service to validate the quotes issued by the Quoting Enclaves.

This solution, initially deployed by Intel, is not suitable for several scenarios and more particularly in the case of the cloud, for example Azure, where it is not desirable for security reasons to impose on each machine hosting a SGX platform an outbound internet connection to external services. In addition, the cloud provider may want to limit the impact on availability linked to a possible loss of connection to Intel services.

By deploying its own attestation service based on a cache principle, the cloud provider will be able to continue to validate attestations for platforms hosted in its datacenter while relying on Intel's trusted roots. The cloud provider may also want to have more control over the attestation mechanism and more freedom in the evolution and

[24] Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation: https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf

capabilities to integrate Intel SGX attestations into a global attestation system. Finally, the principle of the EPID key under Intel control can go against the confidentiality needs requested by the cloud provider or its customers.

To meet these expectations, Intel has made available an evolution of its remote attestation process to open it to third-party solutions mainly for cloud providers. This involves defining a more open architecture model with the availability of open source components to allow cloud providers to build their own attestation system.

This evolution involves changes on the local platform with impacts on the "architectural" enclaves, but also requires building and deploying services in the data center to ensure the functions of cache, internal attestation and registration of SGX platforms deployed in the data center.

To understand it is necessary to go deeper into the new principle of attestation. Let's start with the impacts on the Intel SGX platforms hosted in the data center.

Using ECDSA-based Quoting Enclave

The role of the Quoting enclave is to transform a local attestation into a remote attestation which can be verified outside the machine on which the application enclave is executed by using the attestation service. In the previous chapter, we detailed how the private EPID key is used by the Quoting enclave to sign the Report of the application enclave, and how the associated EPID group public key is used by the IAS service to validate the quotes.

The new generation of Quoting enclave no longer relies on the use of an EPID key – which required a dialogue with the Intel Provisioning Service – but generates its own attestation key pair, a signature key generated by an algorithm of type Elliptic Curve Digital Signature Algorithm (ECDSA) compliant with FIPS 186-4 and RFC 6090. For the sake of simplicity, we will call it "ECDSA attestation key" or simply attestation key.

This ECDSA attestation key is derived from the Seal Key, which gives it the advantage of being able to be regenerated identically (as long as the TCB of the platform does not evolve), to be able to survive to reboots (which does not need to make it persistent) and to be unique for the whole platform.

An "architectural" Quoting Enclave developed and signed by Intel is now provided in its ECDSA Quoting library[25].

Provisioning Certification Enclave and PCK

A new enclave "architectural" enclave has emerged to support this new mode of attestation: the *Provisioning Certification Enclave*. Indeed, the ECDSA attestation key is generated locally by the Quoting Enclave but, for the moment, there is nothing to prove from an external point of view, that it was actually created in an enclave of an SGX processor.

The purpose of the Provisioning Certification Enclave (or PCE) is to certify that the ECDSA attestation key has been created by an SGX processor. To do this, the PCE must both:

- make sure, when it receives a request, that this request comes from a Quoting Enclave hosted on the same platform.
- be able to sign with a key which proves that the platform on which it is running – like the other enclaves – is a genuine Intel SGX platform.

[25] Intel(R) Software Guard Extensions Data Center Attestation Primitives
https://github.com/intel/SGXDataCenterAttestationPrimitives/blob/master/README.md

The PCE generates a key, *the Provisioning Certification Key* (PCK), derived from hardware keys of the processor. This key is unique to each SGX processor. Intel also makes the associated public key available in a certificate to be used by attestation services.

An identifier is also generated, the *Platform Provisioning ID* (PPID) which makes it possible to serve as a key (in a database sense) to reference and access the PCK certificate associated with the platform and containing the PCK public key.

The way in which the keys will be used allows to retrace the chain of trust. The Quoting enclave asks the Provisioning Certification Enclave to sign with its PCK key the ECDSA attestation key which is itself used to sign the REPORT structure of the application enclave. As a result, the trust in the hardware carried by the PCK is transferred to the ECDSA key, which itself proves that the REPORT is indeed issued from an enclave running on an SGX processor.

This is the principle, but the reality is a little bit more complex and is detailed in the attestation workflow below.

Introducing the Cloud Provider Attestation Service

The last component of the chain is the attestation service deployed in the cloud provider datacenter. The service is responsible for validating the attestation presented by client applications which want to ensure that they initiate a dialogue with real enclaves.

To do this, the service must have at its disposal the list of PCK certificates for SGX machines that registered during their deployment in the datacenter. These certificates will have been previously downloaded from the *Provisioning Certification Service* for Intel SGX accessible on the internet along with other information that is necessary for validation (revocation list of the certificate or intermediate CAs, versions of CPU and PCE, etc.).

Upon request for validation of a quote, the Attestation Service uses the platform identifier (PPID), provided in encrypted form to find the associated PCK certificate. It then proceeds to verify the certification chain and information concerning the requesting Quoting Enclave before rendering a verdict.

Understanding the Remote Attestation Workflow

The remote attestation workflow is represented on the diagram below.

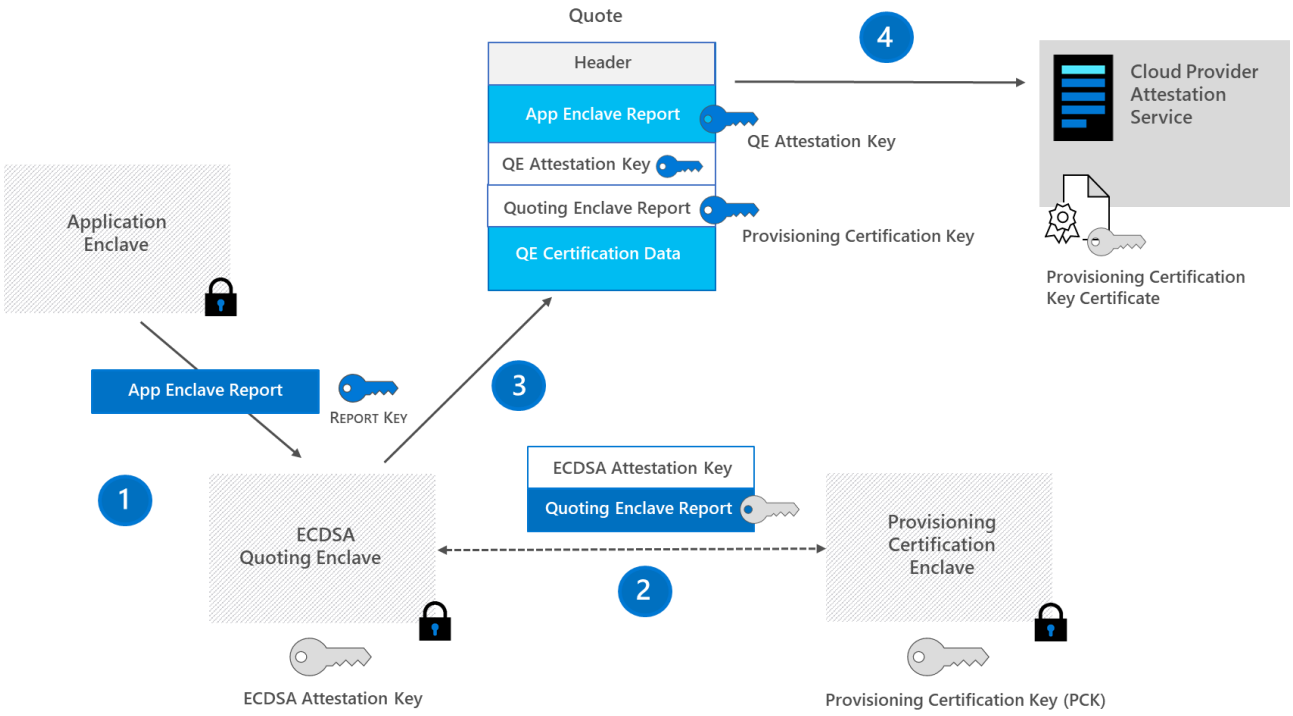


Figure 5 Detailed Remote Attestation workflow

The remote attestation workflow begins with the request from a remote application (challenger) to the application enclave to provide an attestation. This challenger (not shown in the diagram) wants to establish a dialogue with the enclave but requests this attestation to ensure that the enclave is a true SGX enclave and execute an uncompromised code.

- **Step 1**: Creation of the Application Enclave REPORT and request for signature:
    - The Application enclave creates its REPORT targeting the Quoting Enclave (QE) by having it signed with the ReportKey of the latter. The REPORT is sent to the Quoting Enclave.
- **Step 2**: Signature of the ECDSA attestation key of the Quoting Enclave[26]:
    - The Quoting Enclave generates its ECDSA attestation key. It then generates its REPORT structure targeting the Provisioning Certification Enclave (PCE). It calculates a hash of its attestation key and includes it in the ReportData field of its REPORT. This will allow later to check the integrity of the attestation key.
    - The ECDSA attestation key and the QE REPORT are sent to the PCE for signature.
    - The PCE signs the REPORT structure of the QE with its Provisioning Certification Key (PCK) and returns it to the QE.
- **Step 3**: Creation of the Quote:
    - The QE creates the Quote which includes:
        - the REPORT of the application enclave signed with the ECDSA attestation key.
        - the public key part of the ECDSA attestation key.

[26] Note that this step may have taken place previously and the REPORT structure of the QE already available.

        - the REPORT of the Quoting Enclave signed with the PCK key.
        - additional data to verify the REPORT of the QE (including for example the encrypted PPID).
    - Once created, the Quote is returned to the application enclave so that it can return it to the challenger.

- **Step 4**: Verification of the certificate with the Attestation Service
    - The challenger received the Quote and requests the cloud provider's Attestation Service to validate it.
    - The Attestation Service looks for the certificate containing the PCE public key associated with the platform using the PPID identifier[27].
    - The Attestation Service can validate the Quote by checking[28]:
        - The signature of the REPORT by going up the chain of trust from the PCK key, by relying on the certificate containing the public key PCE (validating that no key has been revoked).
        - The identity of the Quoting Enclave and its update status.
    - If the result of the verification is correct, the attestation service sends a positive response to the challenger.

Note For a more complete description of the certification verification process, see document Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives[29]. For a precise description of structures and APIs, see document Intel® Software Guard Extensions (Intel® SGX) Data Center Attestation Primitives: ECDSA Quote Library API[30].

[27] In the implementation of the DCAP library of Intel SGX, the User Data field is used to store a QE identifier to make the link between the encrypted PPID and the PCK certificate.

[28] For the sake of simplicity, certain points of verification relying to possible versions of the TCB have been omitted.

[29] Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives: https://software.intel.com/sites/default/files/managed/f1/b8/intel-sgx-support-for-third-party-attestation.pdf

[30] Intel® Software Guard Extensions (Intel® SGX) Data Center Attestation Primitives: ECDSA Quote Library API: https://download.01.org/intel-sgx/dcap-1.1/linux/docs/Intel_SGX_ECDSA_QuoteGenReference_DCAP_API_Linux_1.1.pdf

Using attestations with Microsoft Azure Attestation

As cover in the previous section, enclave attestation is a process for verifying that an enclave is secure and trustworthy, and thus can be attested.

Let's consider what Microsoft Azure Attestation (MAA) provides in this space.

An overview of the Microsoft Azure Attestation

Microsoft Azure Attestation (MAA) is a newly introduced regional service on the Azure platform. As of this writing, MAA is currently in Public Preview as announced at the //Build'2020 Conference.

As such, MAA is a unified customer-facing service and framework for attestation. As its name indicates, attestation is the central tenet of MAA.

As already illustrated in the previous section, attestation is a process for demonstrating that software binaries were properly instantiated on a trusted platform. Remote parties can then gain confidence that only such intended software is running. The attestation can include detail on the mode of operation, as well as any other data associated by the attested software.

MAA supports Azure's growth and expansion by enabling cutting-edge security paradigms such as Confidential Computing (with Azure Confidential Computing[31] (ACC), and Intelligent Edge protection.

These innovative approaches take cloud security to a new level by introducing data encryption in use. Such capabilities allow high-value customers to minimize their trust in the Cloud Service Provider (CSP) and paves their path to the public cloud.

ACC protects the confidentiality and integrity of customer's data and code while it is processed in the public cloud. ACC is aimed to protect data from the following threats:

1. Malicious insiders with administrative privilege or direct access to hardware on which it is being processed.
2. Hackers and malware that exploit bugs in the operating system, application, or hypervisor.
3. Third parties accessing it without their consent.

ACC ensures that the data is protected inside a TEE. Protection of data using ACC[32] is accomplished in two ways:

1. **Hardware**. Azure can offer hardware-protected families of virtual machines (VMs) that run on the Intel SGX technology at the center of this paper. See section § *Using a DC_series VM in Azure* below.
2. **Hypervisor**. Virtualization-based security (VBS), a.k.a. Virtual Secure Mode (VSM), is a software-based TEE that's implemented by Hyper-V in Windows Server 2016 and above (as well as Windows 10). Hyper-V

[31] Azure Confidential Computing: https://azure.microsoft.com/en-us/solutions/confidential-compute/

[32] Protect Your Data with Azure Confidential Computing: https://docs.microsoft.com/en-us/archive/msdn-magazine/2019/february/azure-protect-your-data-with-azure-confidential-computing

prevents administrator code from running on the computer or server, as well as local administrators and cloud service administrators from viewing the contents of the enclave or modifying its execution.

As a result, customers can tap into disruptive business models which hitherto required scalable compute resources and uncompromising trust, an often-impossible combination. Concretely, customers have been requesting the ability to independently verify the location of a machine, the posture of a VM, and the environment within which TEE are running. MAA empowers these and many additional customer requests.

As a unified service, MMA enables customers to attest their trusted execution environments (TEEs). MAA is a single extendable service for attesting diverse TEE's types to certify: off course the Intel SGX enclaves widely discussed so far, with specific caching capabilities, the already introduced Open Enclave SDK (OESDK) as well - OESDK standardizes specific requirements for verification of identity. This qualifies Open Enclave as a highly fitting attestation consumer of MAA -, but also above software-based VBS/VSM enclave on Microsoft Hyper-V, Cyber Resilient Security (CyReS) TrustZone enclaves for ARM.

Thanks to this coverage, **MAA provides customers and their workloads security assurances that specific sensitive code environments running in TEEs, both in the cloud, on the edge, and on-premises, are trustworthy.**

You can attest a TEE using a Microsoft provided attestation default provider's service endpoint or can create your own a MAA instance, i.e. a so-called attestation provider if you have specific attestation policy requirements.

As such, an attestation provider is service endpoint that provides the MAA REST contract, see section § *Using the attestation provider REST* API below. Each provider honors a specific, discoverable policy. To configure MAA for attesting enclaves, customers would create an attestation provider, see section § *Creating your own attestation provider* below.

They or their workloads can then use in turn the related service endpoints to get attestation for the above-mentioned TEEs.

MAA is critical to Confidential Computing scenarios, as MAA:

- Validates if a trusted execution environment (TEE) is a valid one.
- Evaluates the TEE against a customer policy (evaluates identity/properties of the TEE).
- Manage and store tenant specific policies.
- Generates and signs a token that is used by relying parties to interact with the TEE.

Last but not least, being an integrated Azure service, MAA will provide azure level SLA.

Let's see how this works.

Understanding how Microsoft Azure Attestation works

Below is a high-level view of Microsoft Azure Attestation (MAA) validating a client TEE.

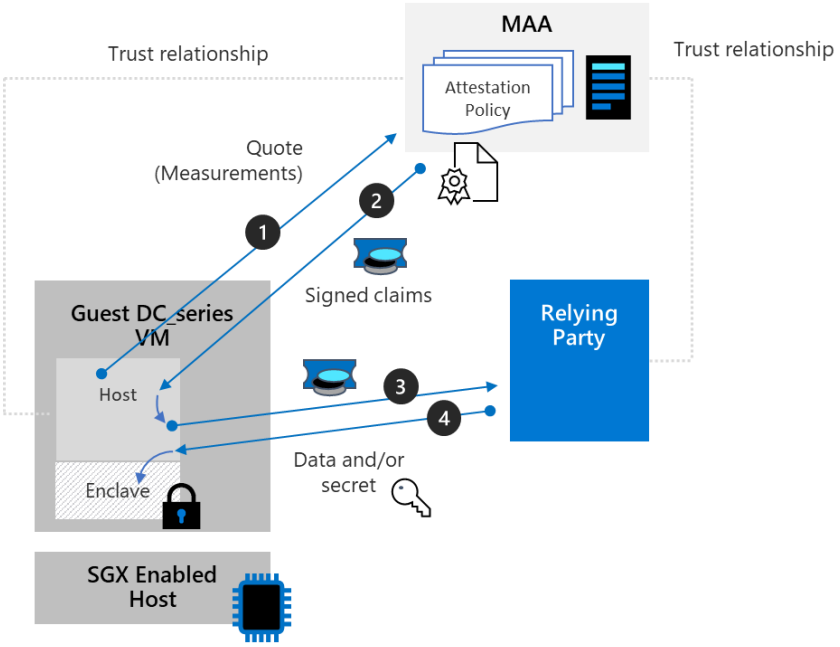


Figure 6 Microsoft Azure Attestation workflow

Here are the general steps in a typical TEE attestation validation workflow using a MAA:

1. The client TEE based application has an attestation URL which references its attestation provider, i.e. a MAA instance.
    a. The TEE creates a "quote" which expresses the state of the enclave.
    b. The Host authenticates to Azure and obtains an access token for the attestation provider, referenced in the attestation URL (from Azure Active Directory). (The attestation provider runs here in the AAD trust mode, see section § *Understanding the MAA's trust model* below.)
    c. The Host collects the attestation quote about the portion of the application running inside the TEE, as well as the TEE environment.
    d. The Host sends this information to the attestation provider. Exact information submitted to the provider depends on the TEE's type.
2. The attestation provider verifies the submitted information against the attestation policy in place for that TEE's type (SGX, OESDK, VBS, etc.) (defined when creating the provider).
    a. If the verification succeeds, i.e. all the required conditions as per defined policy are fulfilled, the provider emits the provider a set of claims in an attestation token and returns the token to client application:
        - The series of claims to include in the attestation is dictated by the above policy that applies.

- The attestation token generated by MAA is formatted as [RFC 7800 JSON Web Token (JWT)][33].

Note JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret or a public/private key pair.

- The JWT token is finally signed so that the client TEE base application or any relying party can validate that the token is originated from a trusted attestation provider.

b. If this step fails, the attestation provider reports an error to the client TEE based application.

Note Customers are in full control of their security policies. Beyond the default policies in place for the each supported TEE's types, custom attestation policy requirements in terms of authorization and issuance can be expressed, uploaded to the attestation provider, and then enforced as part of the evaluation process of any attestation request that comes in and drive the release of any attestation token. Besides the conditions to fulfill to issue a token, such a policy also dictates the claims to include in the related JWT token. See section § *Managing the attestation policies of your own attestation* **provider** below.

3. The client TEE based application sends in turn the attestation token to the relying party if any.
4. The relying party can encrypt secrets/data with the asserted public key of the client TEE and send it back to the enclave.
    a. First, the relying party, whatever it is in terms of application or service, can verify the signature of the attestation token using the public signing key of the attestation provider. To verify the signature, the Relying party can contact the public key endpoint of the attestation provider, referenced in the attestation URL, to retrieve a public signing key for the provider, see section § *Using the OpenID Metadata endpoint* below.
    b. Then the relying party uses the public key contained in the attestation (i.e. the `aas-ehd` field) to encrypt the data to be sent to the enclave. The relying party can be certain that the key was in fact a key known to the TEE because the attestation service verified that the TEE was valid and the TEE held data was known to the portion of the application in the TEE.

As outlined above, one of the important steps in this TEE attestation validation workflow is the validation of the returned attestation token.

Fortunately, because it follows the standard JWT, it can be verified and parsed by various existing libraries. For example, the following code in C# will validate the JWT returned by the attestation provider (using the APIs in the [Microsoft.IdentityModel.Tokens namespace][34]).

For simple validation of the JWT, it can be done as easily as:

```
var jwtHandler = new JsonWebTokenHandler();

var validatedToken = jwtHandler.ValidateToken(encodedJwt, tokenValidationParams);
Assert.IsTrue(validatedToken.IsValid);
```

[33] RFC 7800 Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs): https://tools.ietf.org/html/rfc7800

[34] MICROSOFT.IDENTITYMODEL.TOKENS NAMESPACE: https://docs.microsoft.com/en-us/dotnet/api/microsoft.identitymodel.tokens?view=azure-dotnet

For the attestation provider, the certificate which is used to sign the quote is a self-signed certificate with a subject name whose name matches the attestUri value for the tenant, see section § *Creating a new attestation provider* below.

Furthermore, to establish trust in the above workflow, it is critical to ensure that Microsoft entities such as VM admins, Host admins and Microsoft developers cannot modify attestation requests, policies and MAA-issued tokens. To achieve this, features of MAA like quote validation, token generation, token signing, and policy management are moved into a SGX enclave.

MAA is built to run in two types of environments:

- TEE implementation of MAA running in SGX enclaves.
- Non TEE implementation of MAA with Business continuity and disaster recovery (BCDR)[35] capability, which enables to mitigate service disruptions resulting from significant availability issues or disaster events in a region.

in order to validate that a particular quote was evaluated in the MAA SGX enclave, the client TEE based application or any relying party will check that the signing key used to sign the attestation token was generated in an SGX enclave. The client TEE based application or any relying party will fetch the token signing cert from the MAA OpenID Metadata endpoint, see section § *Using the OpenID Metadata endpoint* below.

As far as the above former type of environment is concerned, the token signing certificate contains the MAA SGX Quote Enclave that the client can verify to ascertain that the signing key was generated in the SGX enclave. (The SHA256 hash of the public key used to sign the attestation token is put in the `report_data` field of the quote's `REPORT`.)

Furthermore, regarding more specifically the Intel SGX technology at the core of this paper, one should note that MMA implements a *Provisioning Certificate Key* (PCK) Caching Service.

Understanding the role of PCK Caching service

The *Provisioning Certificate Key* (PCK) Caching Service is an Azure caching service for the Intel® SGX provisioning certificate service. It's used by MAA and exposed through REST APIs.

As extensively covered in section § *Interacting with the Intel Attestation Service* above, Intel publishes the PCK structures through a hosted service called the SGX provisioning certificate service.

These include PCK certificates, certificate revocation lists, TCB Information, and Quoting Enclave identity structures. Azure collects periodically and caches these structures from Intel's certificate service. Additional information can be found on Intel's product brief[36] on Datacenter Attestation Primitives.

The primary objectives of the PCK caching service are to operate at scale, reduce dependencies on externally hosted services, and limit access to the Internet to perform runtime operations. These cached primitives are typically valid for a period measured in days or more.

It's high time to put all of these concepts in practice. This will provide you with the occasion of deepen your understanding and knowledge of MMA.

---

[35] Business continuity and disaster recovery (BCDR): Azure Paired Regions: https://docs.microsoft.com/en-us/azure/best-practices-availability-paired-regions

[36] Intel® SGX Data Center Attestation Primitives (Intel® SGX DCAP): https://download.01.org/intel-sgx/dcap-1.1/linux/docs/Intel_SGX_DCAP_ECDSA_Orientation.pdf

Preparing your environment for interacting with MAA

To setup the configuration, you will need access to an Azure subscription and at least one server with Intel SGX extension, such as in the rest of this document a DC_series VM instance in Azure (see section § *Using a DC_series VM in Azure* below).

First, you will need to configure your Azure subscription(s).

Preparing your local PowerShell environment on Windows 10

Let's start by installing Azure PowerShell on your local machine if you haven't done so yet.

Installing Azure PowerShell

See Install Azure PowerShell[37] as well as Overview of Azure PowerShell[38] for information on how to install and run Azure PowerShell.

If Azure PowerShell isn't already installed on your Windows 10 local machine, start by doing so. Perform the following steps:

1. Open an elevated PowerShell console and install Azure PowerShell:

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSignedLocation
```

   a. When invited, press A.

```
PS C:\> Install-Module -Name Az -AllowClobber -Scope CurrentUser
```

   b. When invited, press Y.
   c. When invited, press A.

2. If Azure PowerShell is already installed, check that you are using the latest version of AzureRm:

```
PS C:\> Update-Module -Name Az
```

Note Az and AzureRm modules cannot be imported in the same session or used in the same script or runbook. If you are running PowerShell in an environment, you control you can use the 'Uninstall-AzureRm' cmdlet to remove all AzureRm modules from your machine. See here[39] for more information.

Let's continue with the Azure Attestation cmdlets for Azure PowerShell.

[37] Install Azure PowerShell: https://docs.microsoft.com/en-us/powershell/azure/install-az-ps?view=azps-4.4.0

[38] Azure PowerShell documentation: https://docs.microsoft.com/en-us/powershell/azure/?view=azps-4.4.0

[39] Uninstall the Azure PowerShell module: https://docs.microsoft.com/en-us/powershell/azure/uninstall-az-ps?view=azps-4.4.0

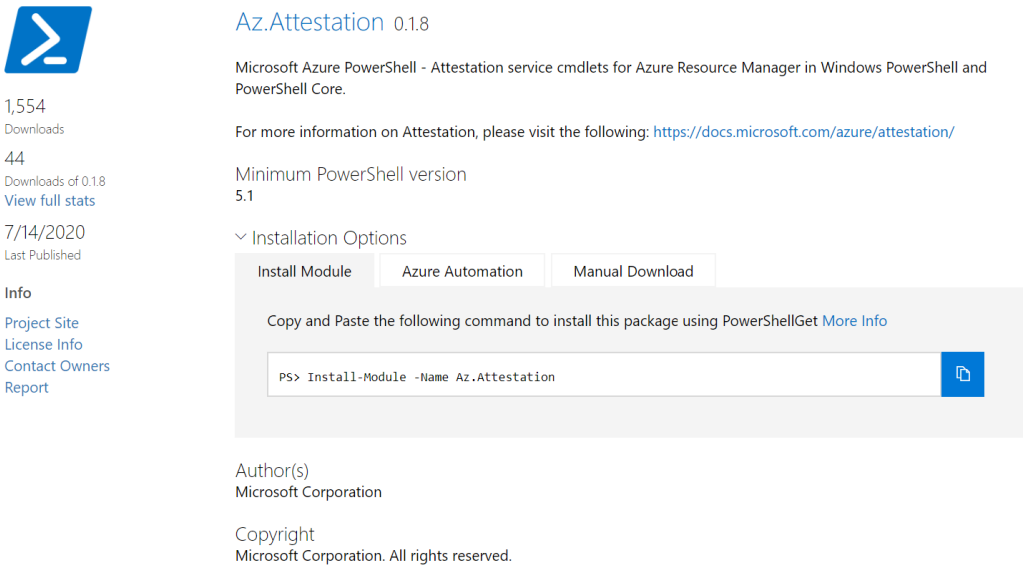
Installing Azure Attestation Cmdlets for Azure PowerShell

See Az.Attestation[40].

You will now Install the Az.Attestation PowerShell module containing cmdlets for MAA.

To install for all users, perform the following steps:

1. From the above elevated PowerShell console, install the latest version of Microsoft Azure Attestation cmdlets for Azure Resource Manager (ARM) in Windows PowerShell from PSGallery[41].



As of this writing, the current version is 0.1.8.

```
PS C:\> Install-Module -Name Az.Attestation -AllowClobber -Scope AllUsers
```

2. Show the Az.Attestation module:

```
PS C:\> Import-Module -Name Az.Attestation
PS C:\> Get-Module -Name Az.Attestation| Format-Table -Property Version, ExportedCommands
```

[40] Az.Attestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/?view=azps-4.4.0

[41] Az.Attestation (PSGallery): https://www.powershellgallery.com/packages/Az.Attestation

At this stage, the following Azure Attestation Cmdlets are available in your environment:

Name of the cmdlet	Description
Create operations	
Add-AzAttestationPolicySigner[42]	Adds a trusted policy signer for a tenant in an attestation provider. Specifies the RFC7519 JSON Web Token containing a claim named "aas-policyCertificate" whose value is an RFC 7517 JSON Web Key which contains a new trusted signing key to add. The RFC 7519 JWT must be signed with one of the existing trusted signing keys.
New-AzAttestation[43]	Creates an attestation provider in a specified resource group.
Read operations	
Get-AzAttestation[44]	Gets information about an attestation provider.
Get-AzAttestationPolicy[45]	Gets the policy from an attestation provider and for a specific TEE.
Get-AzAttestationPolicySigners[46]	Gets the trusted policy signers from an attestation provider.
Update operations	
Reset-AzAttestationPolicy[47]	Resets the user defined attestation policy from an attestation provider.
Set-AzAttestationPolicy[48]	Sets the user defined policy on a specific attestation provider for a specific TEE.
Delete operations	
Remove-AzAttestation[49]	Deletes the specified attestation provider.
Remove-AzAttestationPolicySigner[50]	Removes a trusted policy signer on a specific Microsoft Azure Attestation instance.

You will use some of the cmdlet later in this section. For example, `New-AzAttestation` to create a Microsoft Azure Attestation, a.k.a. an attestation provider, see section *§ Creating your own attestation provider* below.

Note For more information on Attestation cmdlets, please visit this website[51].

3. Close the elevated PowerShell console.

[42] Add-AzAttestationPolicySigner: https://docs.microsoft.com/en-us/powershell/module/az.attestation/add-azattestationpolicysigner?view=azps-4.4.0

[43] New-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/new-azattestation?view=azps-4.4.0

[44] Get-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestation?view=azps-4.4.0

[45] Get-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestationpolicy?view=azps-4.4.0

[46] Get-AzAttestationPolicySigners: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestationpolicysigners?view=azps-4.4.0

[47] Reset-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/reset-azattestationpolicy?view=azps-4.4.0

[48] Set-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/set-azattestationpolicy?view=azps-4.4.0

[49] Remove-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/remove-azattestation?view=azps-4.4.0

[50] Remove-AzAttestationPolicySigner: https://docs.microsoft.com/en-us/powershell/module/az.attestation/Remove-AzAttestationPolicySigner?view=azps-4.4.0

[51] Az.Attestation (PSGallery): https://www.powershellgallery.com/packages/Az.Attestation/

Preparing your Azure subscription

Let's now install all the prerequisites that pertains to your Azure subscription(s).

Registering the required Microsoft.Attestation resource provider

To use Microsoft Azure Attestation (MAA), you must register the Microsoft.Attestation required resource provider.

Note that registering a resource provider is required only once for a subscription.

Perform the following steps:

1. Open a new PowerShell console (without elevated access privileges).
2. Import the Az.Attestation module.

```
PS C:\> Import-Module -Name Az.Attestation
```

3. Sign in to Azure.

```
PS C:\> Connect-AzAccount
```

4. If needed, switch to the subscription you to use for MAA.

```
PS C:\> Set-AzContext -Subscription <your_SubscriptionId>
```

5. Register the Microsoft.Attestation resource provider in your subscription.

```
PS C:\> Register-AzResourceProvider -ProviderNamespace Microsoft.Attestation

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registering
ResourceTypes     : {attestationProviders, defaultProviders, locations, locations/defaultProvider...}
Locations         : {East US 2, Central US, UK South}
```

You can also register the above Resource Providers using the Azure portal.

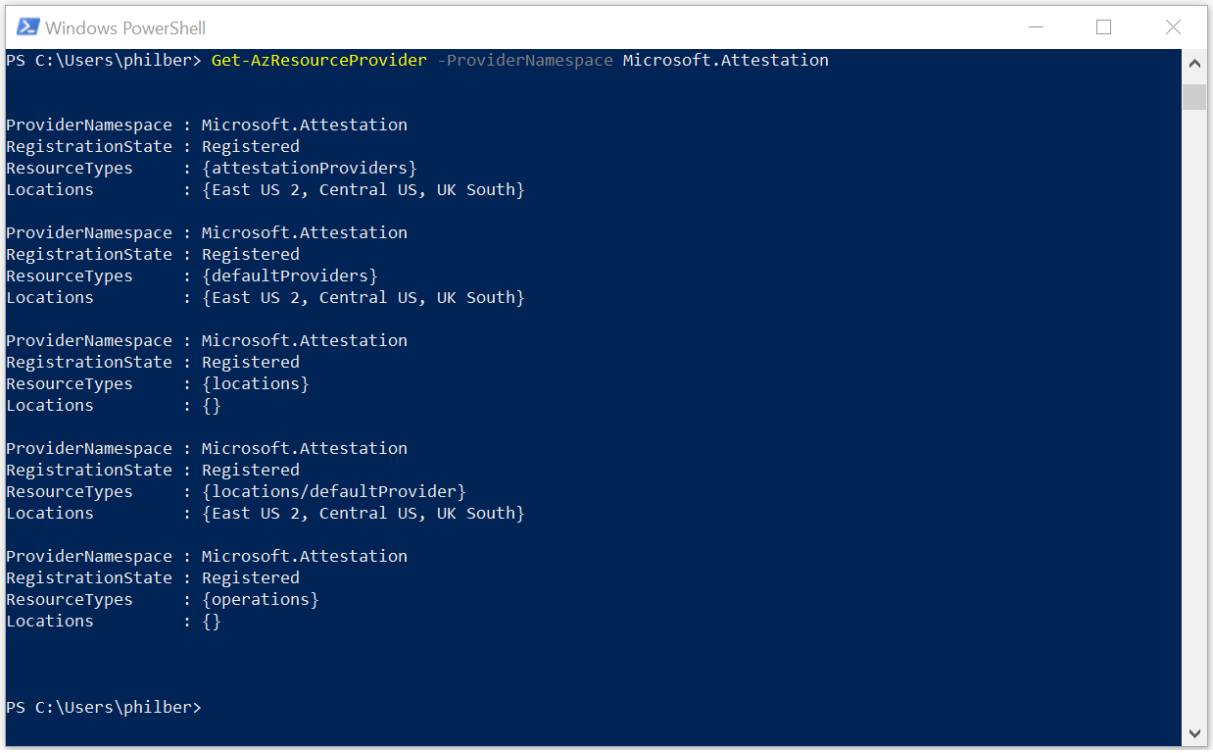
> **Note** For more information about Azure resource providers and how to configure and manage resources providers, see article Azure resource providers and types[52].

[52] Azure resource providers and types
: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-manager-supported-services

Listing the Azure region(s) in which MAA is available

To list the Azure region(s) in which the Microsoft Azure Attestation provider is available, use the following command:

```
PS C:\> Get-AzResourceProvider -ProviderNamespace Microsoft.Attestation

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registered
ResourceTypes     : {attestationProviders}
Locations         : {East US 2, Central US, UK South}

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registered
ResourceTypes     : {defaultProviders}
Locations         : {East US 2, Central US, UK South}

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registered
ResourceTypes     : {locations}
Locations         : {}

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registered
ResourceTypes     : {locations/defaultProvider}
Locations         : {East US 2, Central US, UK South}

ProviderNamespace : Microsoft.Attestation
RegistrationState : Registered
ResourceTypes     : {operations}
Locations         : {}
```



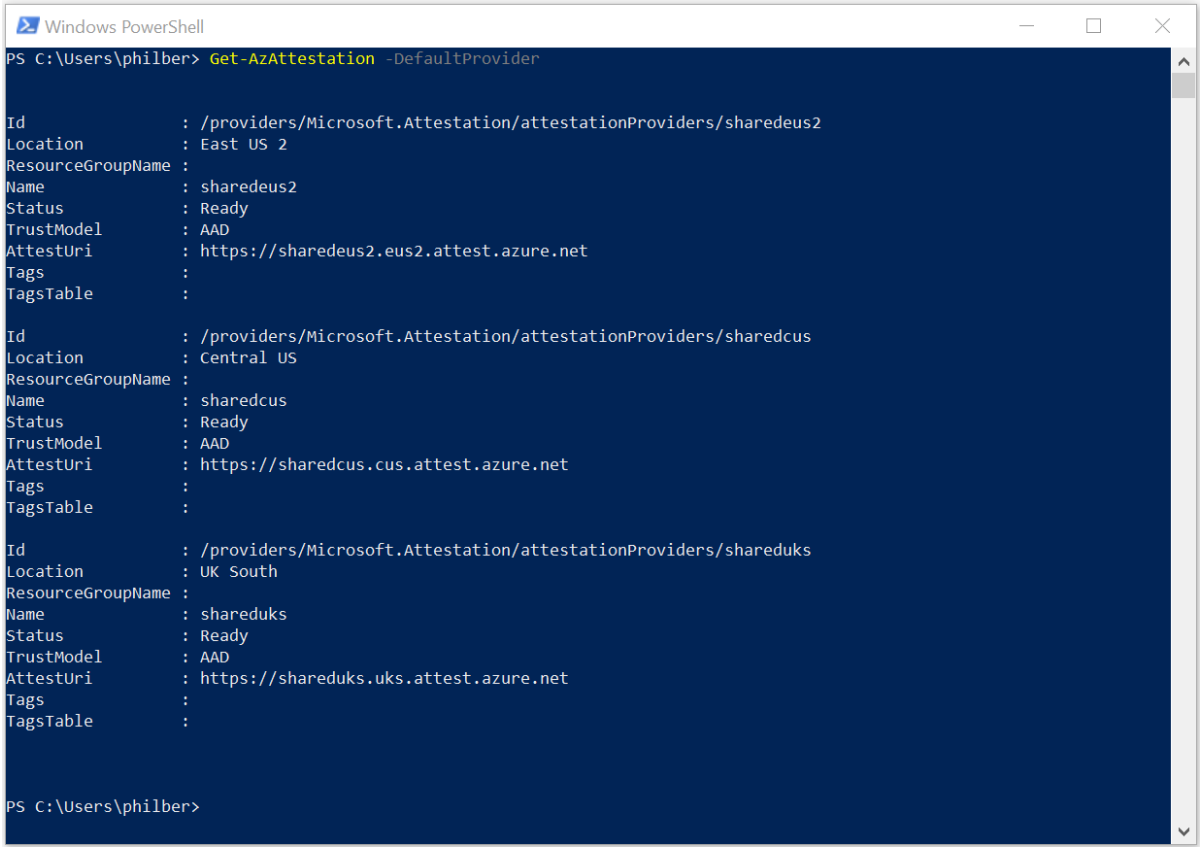
Currently, with the Public Preview, an attestation provider can be instantiated only in the East US 2, Central US, and UK South Azure regions.

Discovering Microsoft provided attestation default providers

Microsoft runs a set of regional attestation default providers that meet the needs of many customers.

To discover Microsoft provided attestation default providers, run the following command:

```
PS C:\> Get-AzAttestation -DefaultProvider

Id                : /providers/Microsoft.Attestation/attestationProviders/sharedeus2
Location          : East US 2
ResourceGroupName :
Name              : sharedeus2
Status            : Ready
TrustModel        : AAD
AttestUri         : https://sharedeus2.eus2.attest.azure.net
Tags              :
TagsTable         :

Id                : /providers/Microsoft.Attestation/attestationProviders/sharedcus
Location          : Central US
ResourceGroupName :
Name              : sharedcus
Status            : Ready
TrustModel        : AAD
AttestUri         : https://sharedcus.cus.attest.azure.net
Tags              :
TagsTable         :

Id                : /providers/Microsoft.Attestation/attestationProviders/shareduks
Location          : UK South
ResourceGroupName :
Name              : shareduks
Status            : Ready
TrustModel        : AAD
AttestUri         : https://shareduks.uks.attest.azure.net
Tags              :
TagsTable         :
```



Note the `DefaultProvider` parameter switch to retrieve the complete set of regional attestation default providers.

Note For more information on the command and its parameters, see Get-AzAttestation[53].

You can then use the results of this enumeration (e.g. ResourceGroup, Name) to call the `Get-*` cmdlets for MAA, as per above section § *Installing Azure Attestation Cmdlets for Azure PowerShell* above, to determine the attest URI, examine the trust model, and, examine their policies.

The attestation default providers use the AAD trust model and provide default policy for each TEE type, see section § *Understanding the MAA's trust model* below.

The below table describes MAA environment type as per availability in different regions.

Region	MAA Environment Type	Supported services in Azure
US East 2	Non TEE	SGX, OESDK, and VBS TEE attestation
Central US	Non TEE	SGX, OESDK, and VBS TEE attestation
UK South	TEE	SGX and OESDK TEE attestation

[53] Get-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestation?view=azps-4.4.0

The regions that are supported by BCDR are as follows:

- East US 2 => Paired with Central US
- Central US => Paired with East US 2

Creating your own attestation provider

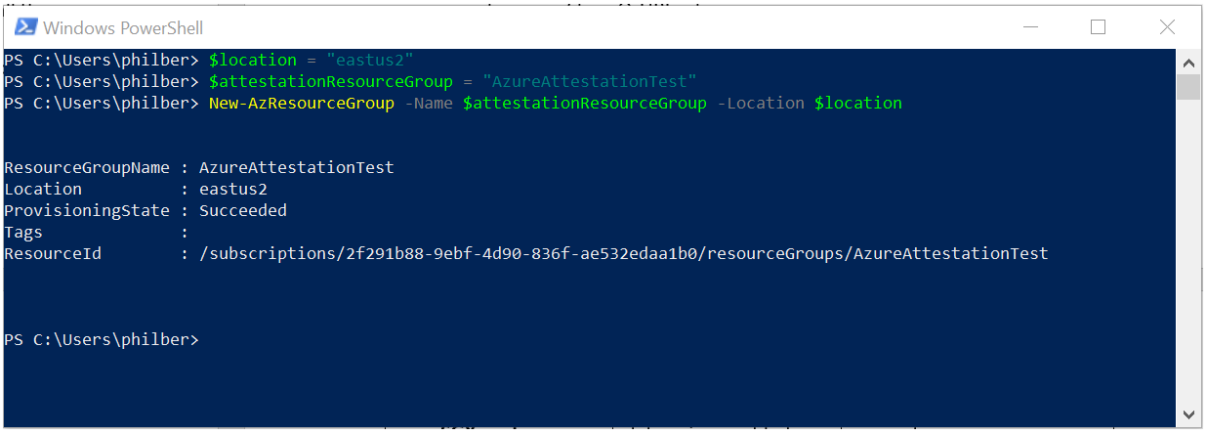
As already mentioned, If the TEE policy in the attestation default provider doesn't meet your own specific needs, you can instead create your own attestation provider in any of the regions supported by MAA.

Unlike the attestation default providers, you will be then able to specify the policy values in your own attestation providers, see section § *Managing the attestation policies of your own attestation* provider below.

Creating a resource group for your own attestation provider

First, create a resource group to hold the required resources, from the previous PowerShell console (without elevated access privileges):

```
PS C:\> $location = "eastus2"
PS C:\> $attestationResourceGroup = "<your_ResourceGroupName>"
PS C:\> New-AzResourceGroup -Name $attestationResourceGroup -Location $location

ResourceGroupName : AzureAttestationTest
Location          : eastus2
ProvisioningState : Succeeded
Tags              :
ResourceId        : /subscriptions/2f291b88-9ebf-4d90-836f-
ae532edaa1b0/resourceGroups/AzureAttestationTest
```



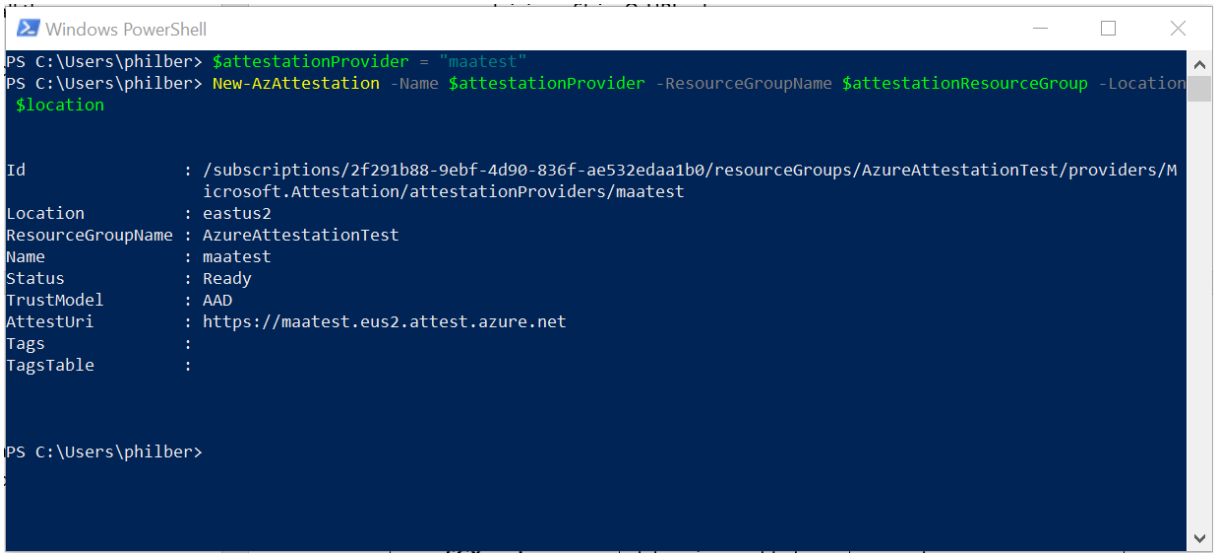
- Replace <*your_ResourceGroupName*> by the name of your choice for the resource group. For example, **AzureAttestationTest** in our illustration.

You can put other Azure resources (including a DC_series VM with your client application instance, see section § *Further illustrating the scenario* 3 below) in the same resource group.

Creating a new attestation provider

From the previousPowerShell console (without elevated access privileges), create your own specific attestation provider, i.e. Microsoft Azure Attestation (MAA) instance:

```
PS C:\> $attestationProvider = "<your_AttestationProviderName>"
PS C:\> New-AzAttestation -Name $attestationProvider -ResourceGroupName $attestationResourceGroup -
Location $location

Id                : /subscriptions/2f291b88-9ebf-4d90-836f-
ae532edaa1b0/resourceGroups/AzureAttestationTest/providers/M
                    icrosoft.Attestation/attestationProviders/maatest
Location          : eastus2
ResourceGroupName : AzureAttestationTest
Name              : maatest
Status            : Ready
TrustModel        : AAD
AttestUri         : https://maatest.eus2.attest.azure.net
Tags              :
TagsTable         :
```



- Replace <*your_AttestationProviderName*> with the name of your choice for your new attestation provider. For example, **maatest** in our illustration.

Note For more information on the command and its parameters, see New-AzAttestation[54].

[54] New-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/new-azattestation?view=azps-4.4.0

Discovering the newly created attestation provider

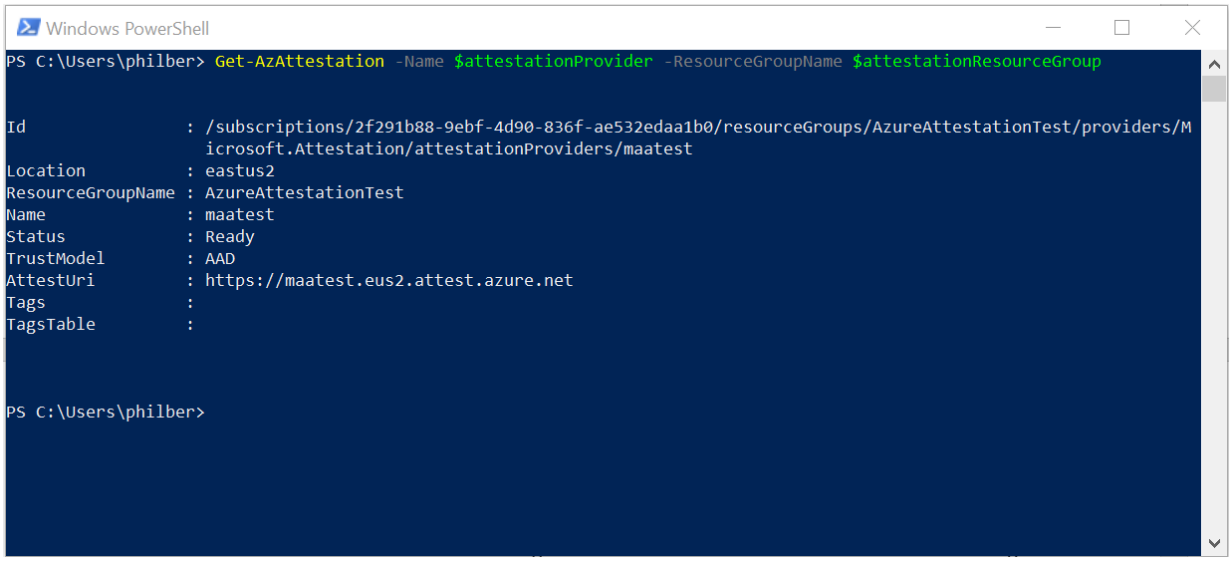
Perform the following steps:

1. Discover the attestation provider you've previously created:

```
PS C:\> Get-AzAttestation -Name $attestationProvider -ResourceGroupName $attestationResourceGroup
```

The results of this command include the following information:

```
Id                : <ARM_ResourceId>
Location          : <your_AzureRegion>
ResourceGroupName : <your_ResourceGroupName>
Name              : <your_AttestationProviderName>
Status            : Ready
TrustModel        : AAD
AttestUri         : https://<your_AttestationProviderName>.<your_AzureRegion>.attest.azure.net
Tags              :
TagsTable         :
```



As such, the creation of an attestation provider is an asynchronous process. Retrieve the **Status** and **AttestURI** properties of your newly created attestation provider.

You may notice that initially the status value is **Not Ready**. It can take up to 10 minutes for it to become **Ready**. Therefore, you may run the below command a few times to confirm the new attestation provider is ready to use.

Note For more information on the command and its parameters, see Get-AzAttestation[55].

[55] Get-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestation?view=azps-4.4.0

2. Take a note of the **AttestURI** property, as you will need it later. For example, https://maatest.eus2.attest.azure.net in our illustration.

Understanding the MAA's trust model

Trust model defines the authorization model used to define and update policy on a resource provider.

Microsoft Azure Attestation defines two trust models to choose from. The first model named **AAD** is based on the Azure Active Directory (Azure AD) authorization model and leverages the Azure AD tenant trusted by your Azure subscription(s). In this mode, MAA is integrated into the Azure AD's authentication framework. This enables MAA to support policies that permit token issuance based on identity and ownership.

This is the one used above and, which is the default.

As such, this model supports the following two roles (see section § *Managing the attestation policies of your own attestation provider* below):

1. **Attestation Reader**. This role has granted rights to perform operation such as get access token for valid Quotes and get the policy definition set for the tenant. This role can also read the attestation provider properties.
2. **Attestation Contributor**. This role has permission to set the Policy. This role can also read, write or delete the attestation provider instance.

The second trust model is named **Isolated**. Policy signing keys are the cryptographic keys used to authorize this Isolated trust model.

If you require this trust model for your development, you will be required to specify your own specific policy signing keys at creation time of your attestation provider, so that it will be configured for an isolated trust, and thus not relying on your Azure AD tenant. For example:

```
PS C:\> New-AzAttestation -Name <your_AttestationProviderName>
               -ResourceGroupName <your_ResourceGroupName> `
               -PolicySignersCertificateFile <your_SignersCertificatefile>
               -Location $location
```

- Replace <*your_SignersCertificatefile*> with the path to a PEM Privacy-Enhanced Mail[56] file for your new attestation provider.

The results of this command include the following information:

```
Id                : <ARM_ResourceId>
Location          : <your_AzureRegion>
Name              : <your_AttestationProviderName>
Status            : Ready
TrustModel        : Isolated
AttestUri         : https://<your_AttestationProviderName>.<your_AzureRegion>.attest.azure.net
Tags              :
TagsTable         :
```

[56] Privacy-Enhanced Mail: https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail

Important note If your attestation provider is configured in Isolated Mode with a Policy Signer Certificate file, then it will reject all unsealed policy JWTs.

Using the attestation provider REST API

Microsoft Azure Attestation is multi-tenant service that supports AAD authentication or certificate authentication (isolated mode). Each customer is expected to create its own MAA instance. Each MAA instance provides a tenant specific URL to access the MAA instance. It's service endpoint that provides the MAA REST contract that honors a specific, discoverable policy.

Sending requests to the attestation provider

Depending on the trusting mode, Azure AD credentials of the tenant or certificate credentials in (PEM) file are used control access to attestation REST APIs.

If the attestation provider's endpoints are Azure AD protected, the client will typically present a bearer token in an Authorization header as part of the request to leverage specific operations:

```
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJ…
```

Note For information about retrieving an Azure AD bearer token, see blog post Accessing Azure App Services using Azure AD Bearer token[57]. (See the portion of this blog post that discusses "Create code to get a Bearer token from Azure AD and use this token to call the Target app").

Note The resource URI for the MAA required for the bearer token should be set to "https://attest.azure.net".

Each attestation provider has REST service endpoints that provides multiple features, like attestation of various TEE types. Each of the APIs is a RESTful API and is accessed via HTTPS URIs.

For example, in our above illustration, we have our own attestation provider resource with the URL https://maatest.eus2.attest.azure.net.

[57] Accessing Azure App Services using Azure AD Bearer token: https://docs.microsoft.com/en-us/archive/blogs/jpsanders/accessing-azure-app-services-using-azure-ad-bearer-token-2

The tenant will use its tenant specific URL to access MAA for both data plane and control plane operations.

Name of the URI	HTTP Verb	Authorization Required?	Description
Attestation			
/attest/Tee/OpenEnclave	POST	Yes	Attests OpenEnclave generated quotes
/attest/SgxEnclave	POST	Yes	Attests an SGX Enclave
/attest/VBSEnclave	POST	Yes	Attests a VBS enclave
/attest/Cyrescomponent	POST	Yes	Attests a CyRes enclave
OpenID Metadata			
/.well-known/openid-configuration	GET	No	Retrieves the OpenID Metadata document for the considered attestation provider. Returns an OpenID Configuration response as per OpenID Connect Discovery 1.0[58] protocol.
Certificate			
/certs	GET	No	Gets information on signing keys and returns an RFC 7517 JSON Web Key Set that contains the signing keys used to sign the attestation response.
Others			
/revoke	POST	Yes	This endpoint enables clients to inform an authorization server that a specified token is no longer used and must be revoked.

Using the OpenID Metadata endpoint

Every attestation provider follows the Open Connect 1.0 standard to publish the signing certificate. The standard specifies using a standard OpenID Metadata endpoint for metadata about token signing. The metadata specifies the algorithm and the keys used to sign the token.

An attestation provider publishes its metadata on the following endpoint:

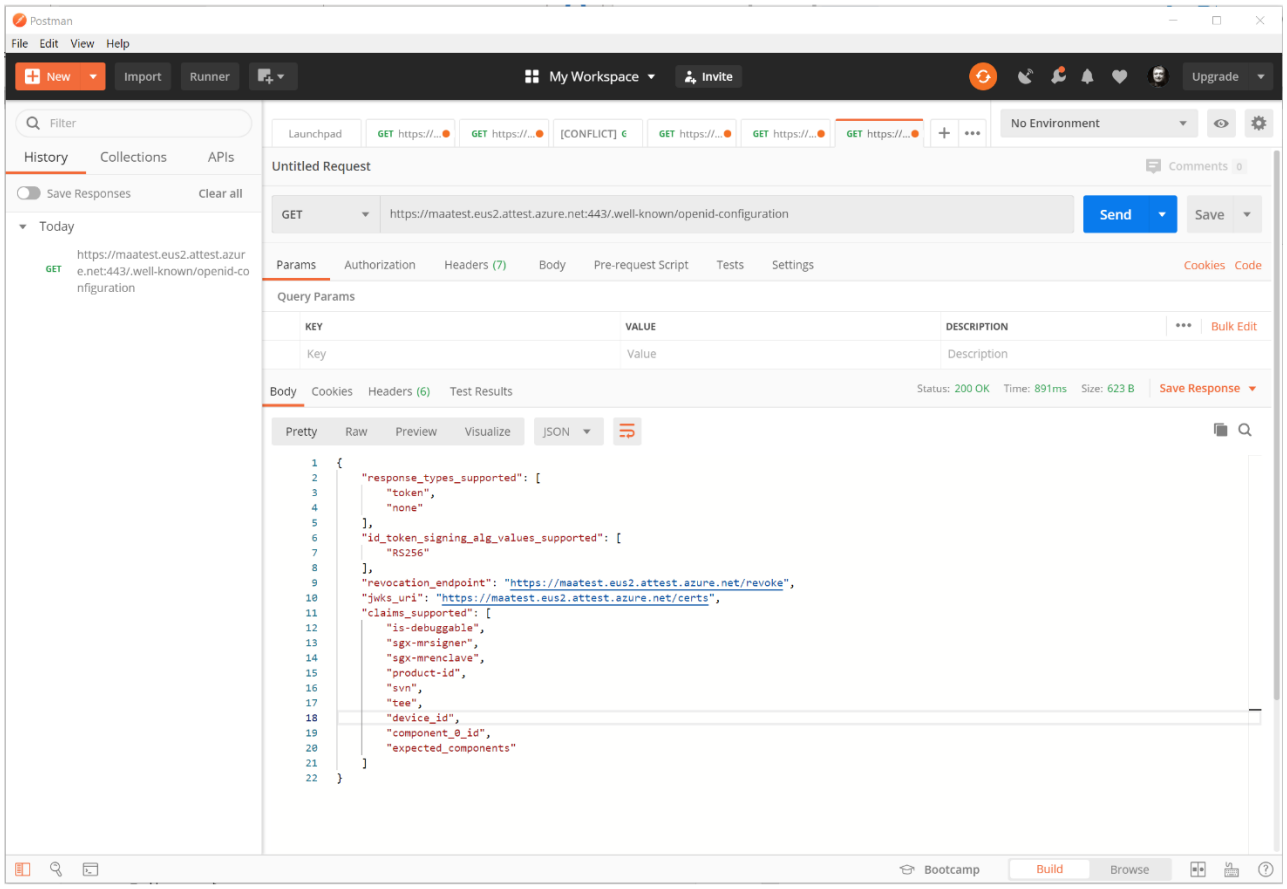
https://<*your_AttestationProviderName*>.<*your_AzureRegion*>.attest.azure.net/.well-known/openid-configuration

For example, in our illustration:

https://maatest.eus2.attest.azure.net:443/.well-known/openid-configuration

Use Postman to get the OpenID Configuration from the OpenID Metadata endpoint. Send a GET command to this URL.

[58] OpenID Connect Discovery 1.0: https://openid.net/specs/openid-connect-discovery-1_0.html




```
{
    "response_types_supported": [
        "token",
        "none"
    ],
    "id_token_signing_alg_values_supported": [
        "RS256"
    ],
    "revocation_endpoint": "https://maatest.eus2.attest.azure.net/revoke",
    "jwks_uri": "https://maatest.eus2.attest.azure.net/certs",
    "claims_supported": [
        "is-debuggable",
        "sgx-mrsigner",
        "sgx-mrenclave",
        "product-id",
        "svn",
        "tee",
        "device_id",
        "component_0_id",
        "expected_components"
    ]
}
```

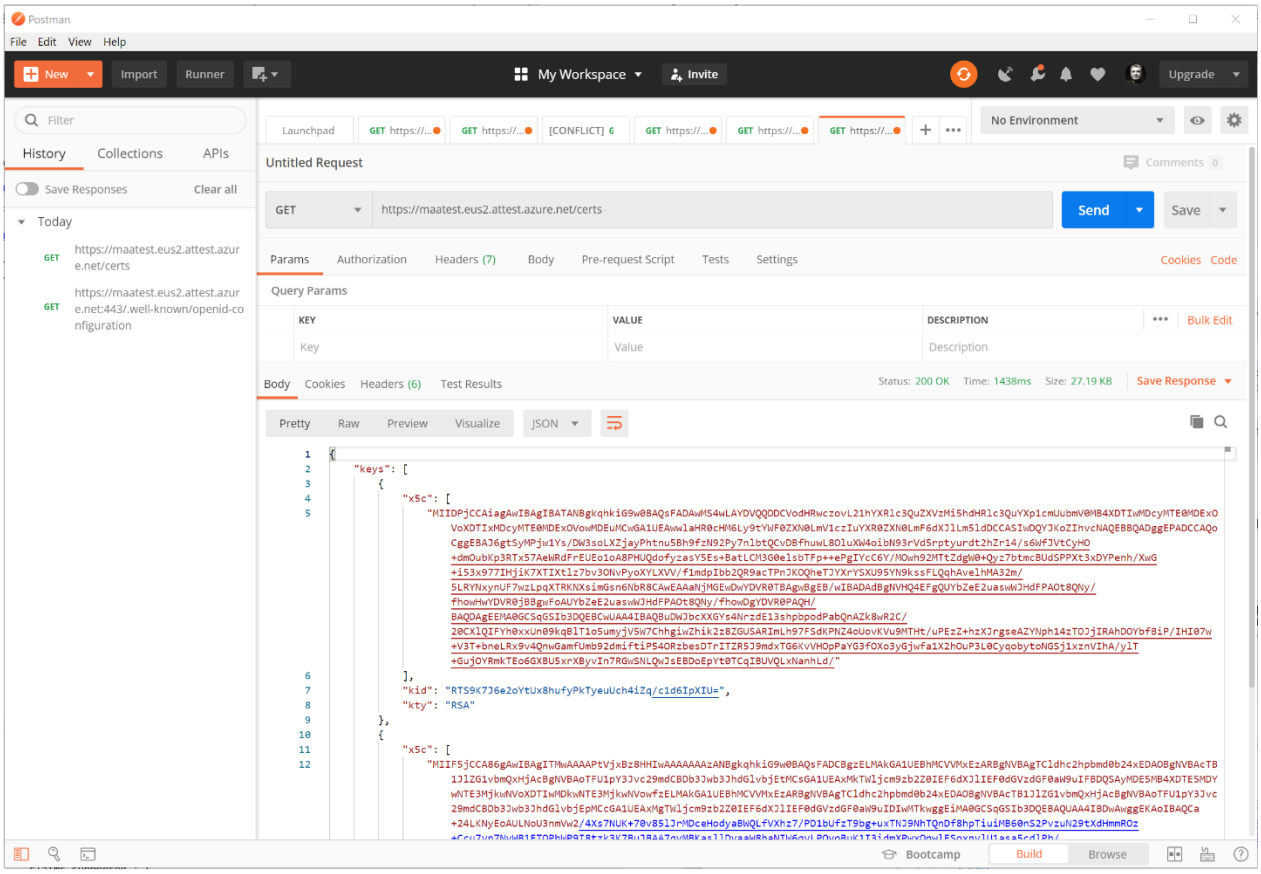
As illustrated above, the metadata specifies the following fields:

Field	Description
response_types_supported	JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports.
id_token_signing_alg_values_supported	The signing algorithm supported.
revocation_endpoint	This endpoint enables clients to inform an authorization server that a specified token is no longer used and must be revoked.
jwks_uri	The URL location of the signing certs
claims_supported	JSON array containing a list of the claims that the MAA instance MAY be able to supply values for.

As outlined in the above table, the `jwks_uri` filed contains the URL location of the signing certs, i.e. the attestation token signing keys.

Now get the signing keys used to sign the attestation response. Send a GET command to this URL. For example, in our illustration:

https://maatest.eus2.attest.azure.net/certs



This should return an RFC 7517 JSON Web Key Set[59] that contains the signing keys used to sign attestation responses.

Below is an example of an RSA signing certificate at the URL.


```
{
  "keys":[
  {"x5c": [

"MIIDPjCCAiagAwIBAgIBATANBgkqhkiG9w0BAQsFADAwMS4wLAYDVQQDDCVodHRwczovL21hYXRlc3QuZXVzMi5hdHRlc3QuYXp1
cmUubmV0MB4XDTIwMDcyMTE0MDUyMFoXDTIxMDcyMTE0MDUyMFowMDEuMCwGA1UEAwwlaHR0cHM6Ly9tYWF0ZXN0LmV1czIuYXR0Z
XN0LmF6dXJlLm5ldDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJ6gtSyMPjw1Ys/DW3soLXZjayPhtnu5Bh9fzN92Py
7nlbtQCvDBfhuwL8OluXW4oibN93rVd5rptyurdt2hZr14/s6WfJVtCyHO+dmOubKp3RTx57AeWRdFrEUEo1oA8PHUQdofyzasY5E
s+BatLCM3G0elsbTFp++ePgIYcC6Y/MOwh92MTtZdgW0+Qyz7btmcBUdSPPXt3xDYPenh/XwG+i53x977IHjiK7XTIXtlz7bv3ONv
PyoXYLXVV/f1mdpIbb2QR9acTPnJKOQheTJYXrYSXU95YN9kssFLQqhAvelhMA32m/5LRYNxynUF7wzLpqXTRKNXsimGsn6NbR8CA
wEAAaNjMGEwDwYDVR0TBAgwBgEB/wIBADAdBgNVHQ4EFgQUYbZeE2uaswWJHdFPAOt8QNy/fhowHwYDVR0jBBgwFoAUYbZeE2uasw
WJHdFPAOt8QNy/fhowDgYDVR0PAQH/BAQDAgEEMA0GCSqGSIb3DQEBCwUAA4IBAQBFloXGhVHSKZB6iWCh7swQc7nS3C961J+CDiA
gEILwF3dAnr9pgYTA3BrKG5w7kHtQjXUPSvSoArKZcmUCNckGkBqNOV7yy1NLc9x+rZnAEpPtShdWqEXSF3KZp/OB1SPPBvH4sKfy
ZXyl2DVjXJV0uE0IdjTuT/ysQxJm7ZWFNV1QXKkFysPow7Hg6jGUlCak5gyCEkFLXbH38dQ426iH64jMSNW+ELIC049pjg/FDbw0o
rRrWO7mvK5mZ1HKxaryL5Of1UncF75rXtmnFDweKCZ8rTOIliCQhAQivvXhD5p5XpzvAh6BqE7g/4BRwWbOIqiA4Ta4UIbTAnsV6S
oF"
            ],
            "kid": "RTS9K7J6e2oYtUx8hufyPkTyeuUch4iZq/c1d6IpXIU=",
            "kty": "RSA"
    }
    ]
  }
```


Managing the attestation policies of your own attestation provider

With your own specific attestation provider, you can update the issuance policy for all TEE types. MAA indeed allows the use of user defined attestation policies to validate properties of the portion of an application running inside a TEE and the platform which hosts it. You can specify a user defined attestation policy or reset to the default policy whenever needed.

Policies are used to determine whether the attestation provider shall issue an attestation token based on evidence, and thereby endorse it. The attestation token would contain the set of claims, which match the configured policy. Accordingly, failure to pass the policies means that no JWT will be issued.

In the AAD trusting model, in order to manage policies, an Azure AD user requires the following permissions for **Actions**:

- `Microsoft.Attestation/attestationProviders/attestation/read`
- `Microsoft.Attestation/attestationProviders/attestation/write`
- `Microsoft.Attestation/attestationProviders/attestation/delete`

These permissions can be assigned to an AD user through a role such as **Owner** (wildcard permissions), "**Contributor**" (wildcard permissions) or **Attestation Contributor** (specific permissions for MAA only).

[59] RFC 7517 JSON Web Key (JWK): https://tools.ietf.org/html/rfc7517

In order to read policies, an Azure AD user requires the following permission for **Actions**:

- `Microsoft.Attestation/attestationProviders/attestation/read`

This permission can be assigned to an AD user through a role such as **Reader** (wildcard permissions) or **Attestation Reader** (specific permissions for MAA only).

Understanding the basics of attestation policy management

Policy management are supported by command-line scripts (PowerShell, Azure CLI) available through Azure SDK. You or administrators, if they choose to do so, can sign their attestation policy using their own cryptographic key.

Note	MAA defines a default policy for each TEE type. Default policies are not available through the Azure portal.

Policy management is twofold. It's important to distinguish the policy creation part from the assignment part.

Policy creation is the process of authoring a custom policy and transforming it into a JWT token describing the policy document to set that Microsoft Azure Attestation can act upon.

The process of creating an attestation policy consists of the following steps:

1. Administrators write a plaintext policy based on the policy language (using their editor of choice, like Visual Studio Code[60]).
2. JSON policy templates are provided for the different TEE types.
3. If administrators want the Isolated trust model (see section § *Understanding the MAA's trust model* above), they will prepare a credential, which will be used to sign the attestation policy
4. Administrators can use a command-line script to prepare the attestation policy for MAA.
    a. The input to the script is the plaintext attestation policy, and the optional credential for signing it.
    b. The script shall sign the policy JWT using the customer credential (if asked).

Policy assignment is the process of associating an attestation policy with an existing MAA instance, see section § *Creating your own attestation provider* above. The attestation policy will be uploaded to the attestation provider, as encoded payload, in a RFC 7515 JSON Web Signature (JWS)[61].

One attestation policy can be configured for each supported TEE type.

The process of uploading an attestation policy consists of the following steps:

1. Administrators use a command-line script to assign the policy on an attestation provider.
2. The attestation provider will parse the attestation policy file and validate its signature (if optionally signed).
    a. For the isolated trust model, the parsing function is run inside a TEE.
    b. If parsing fails, MAA return descriptive error messages.
3. If parsing and assigning the policy succeeds, MAA returns the policy hash, and the credential used to sign it.

[60] Visual Studio Code: https://code.visualstudio.com/

[61] RFC 7515 JSON Web Signature (JWS): https://tools.ietf.org/html/rfc7515

Now that you are equipped with the basics, let's put all of these in practice, and start by getting the policies by default.

Getting the policies by default of your attestation provider

Now download the default policy for the Open Enclave TEE endpoint:

```
PS C:\> Get-AzAttestationPolicy -Name <your_AttestationProviderName> -ResourceGroupName
<your_ResourceGroupName> -Tee <your_TEEType>
```

- Replace <*your_AttestationProviderName*> with the name of your choice for your new attestation provider. For example, **maatest** in our illustration.
- Replace **<***your_ResourceGroupName***>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.
- Replace <*your_TEEType*> with one of the supported TEE types: `SgxEnclave`, `OpenEnclave`, `CyResComponent` and `VBSEnclave`. For example, `OpenEnclave` in our illustration

For example, to get the policy in place for the OpenEnclave endpoint of the attestation provider **maatest** in our illustration:

```
PS C:\> Get-AzAttestationPolicy -Name maatest -ResourceGroupName AzureAttestationTest -Tee
OpenEnclave
```

Note For more information on the command and its parameters, see Get-AzAttestationPolicy[62].

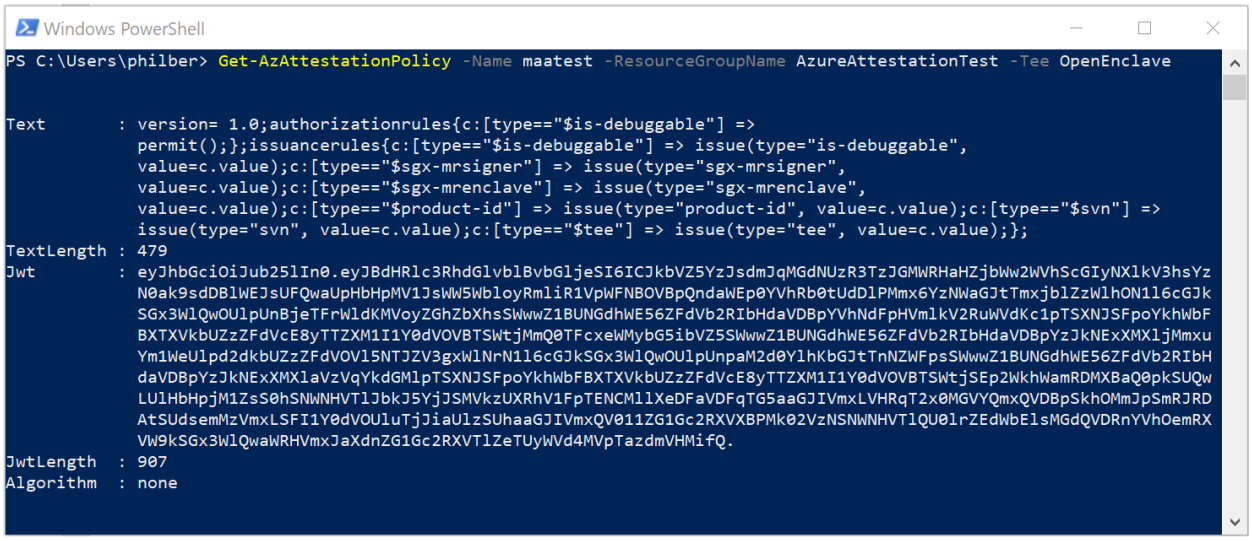
Policies are used to determine whether your attestation provider shall issue an attestation token based on evidence, and thereby endorse it. The token would contain the set of claims, which match the configured policy. Accordingly, failure to pass the policies means that no JWT will be issued.

If success, you will get in return a Base64Url encoded attestation policy file. As already outlined, this file is a RFC 7515 JWS attestation policy file. For example, in our illustration:

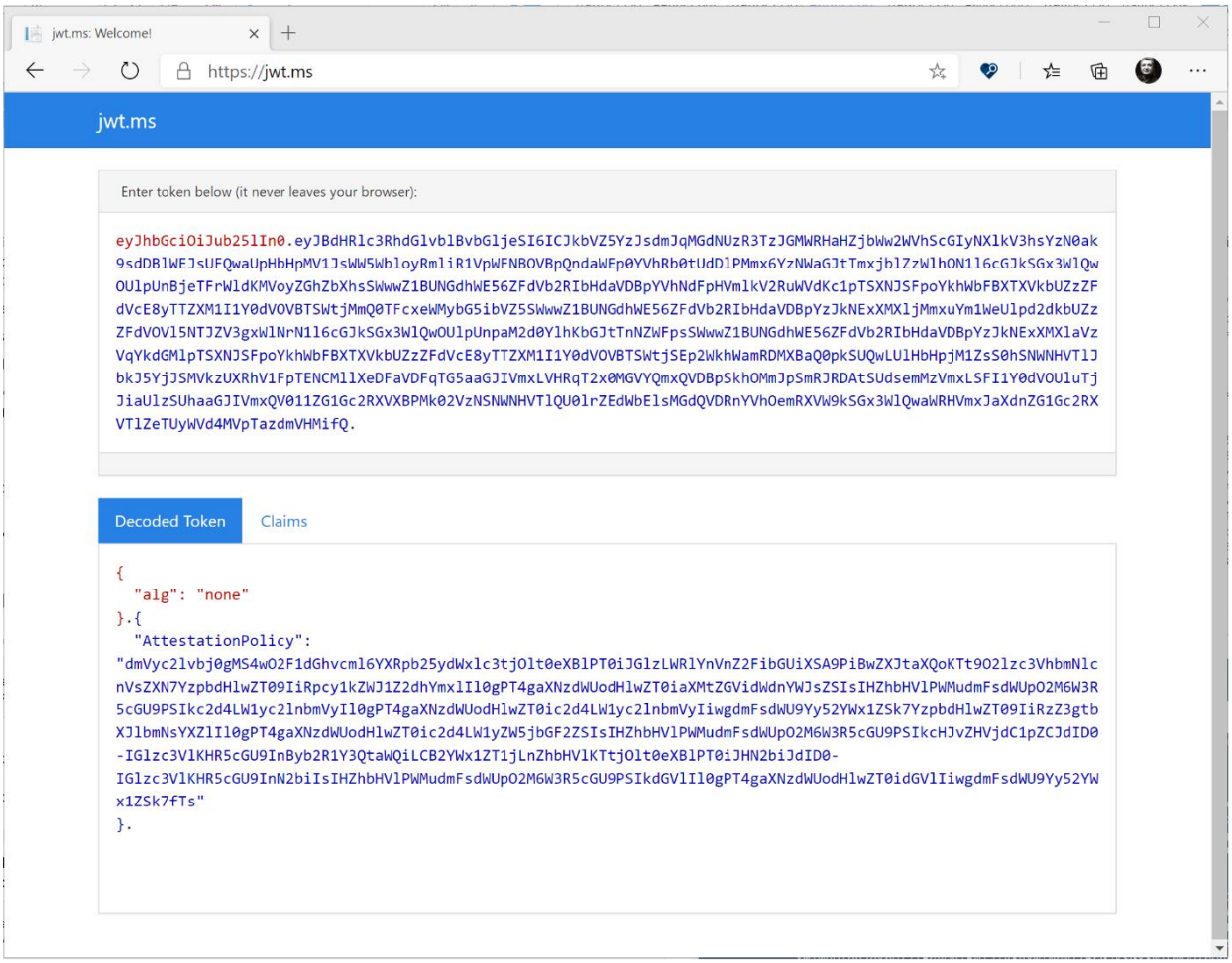
```
Text       : version= 1.0;authorizationrules{c:[type=="$is-debuggable"] =>
             permit();};issuancerules{c:[type=="$is-debuggable"] => issue(type="is-debuggable",
             value=c.value);c:[type=="$sgx-mrsigner"] => issue(type="sgx-mrsigner",
             value=c.value);c:[type=="$sgx-mrenclave"] => issue(type="sgx-mrenclave",
             value=c.value);c:[type=="$product-id"] => issue(type="product-id",
value=c.value);c:[type=="$svn"] =>
             issue(type="svn", value=c.value);c:[type=="$tee"] => issue(type="tee", value=c.value);};
TextLength : 479
Jwt        :
eyJhbGciOiJub25lIn0.eyJBdHRlc3RhdGlvblBvbGljeSI6ICJkbVZ5YzJsdmJqMGdNUzR3TzJGMWRHaHZjbWw2WVhScGIyNXlkV
3hsYzN0ak9sdDBlWEJsUFQwaUpHbHpMV1JsWW5WbloyRmliR1VpWFNBOVBpQndaWEp0YVhRb0tUdDlPMmx6YzNWaGJtTmxjblZzWl
hON1l6cGJkSGx3WlQwOUlpUnBjeTFrWldKMVoyZGhZbXhsSWwwZ1BUNGdhWE56ZFdVb2RIbHdaVDBpYVhNdFpHVmlkV2RuWVdKc1p
TSXNJSFpoYkhWbFBXTXVkbUZzZFdVcE8yTTZXM1I1Y0dVOVBTSWtjMmQ0TFcxeWMybG5ibVZ5SWwwZ1BUNGdhWE56ZFdVb2RIbHda
VDBpYzJkNExXMXljMmxuYm1WeUlpd2dkbUZzZFdVOVl5NTJZV3gxWlNrN1l6cGJkSGx3WlQwOUlpUnpaM2d0YlhKbGJtTnNZWFpsS
WwwZ1BUNGdhWE56ZFdVb2RIbHdaVDBpYzJkNExXMXlaVzVqYkdGMlpTSXNJSFpoYkhWbFBXTXVkbUZzZFdVcE8yTTZXM1I1Y0dVOV
BTSWtjSEp2WkhWamRDMXBaQ0pkSUQwLUlHbHpjM1ZsS0hSNWNHVTlJbkJ5YjJSMVkzUXRhV1FpTENCMllXeDFaVDFqTG5aaGJIVmx
```

[62] Get-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/get-azattestationpolicy?view=azps-4.4.0

```
LVHRqT2x0MGVYQmxQVDBpSkhOMmJpSmRJRDAtSUdsemMzVmxLSFI1Y0dVOUluTjJiaUlzSUhaaGJIVmxQV011ZG1Gc2RXVXBPMk02
VzNSNWNHVTlQU0lrZEdWbElsMGdQVDRnYVhOemRXVW9kSGx3WlQwaWRHVmxJaXdnZG1Gc2RXVTlZeTUyWVd4MVpTazdmVHMifQ.
JwtLength  : 907
Algorithm  : none
```



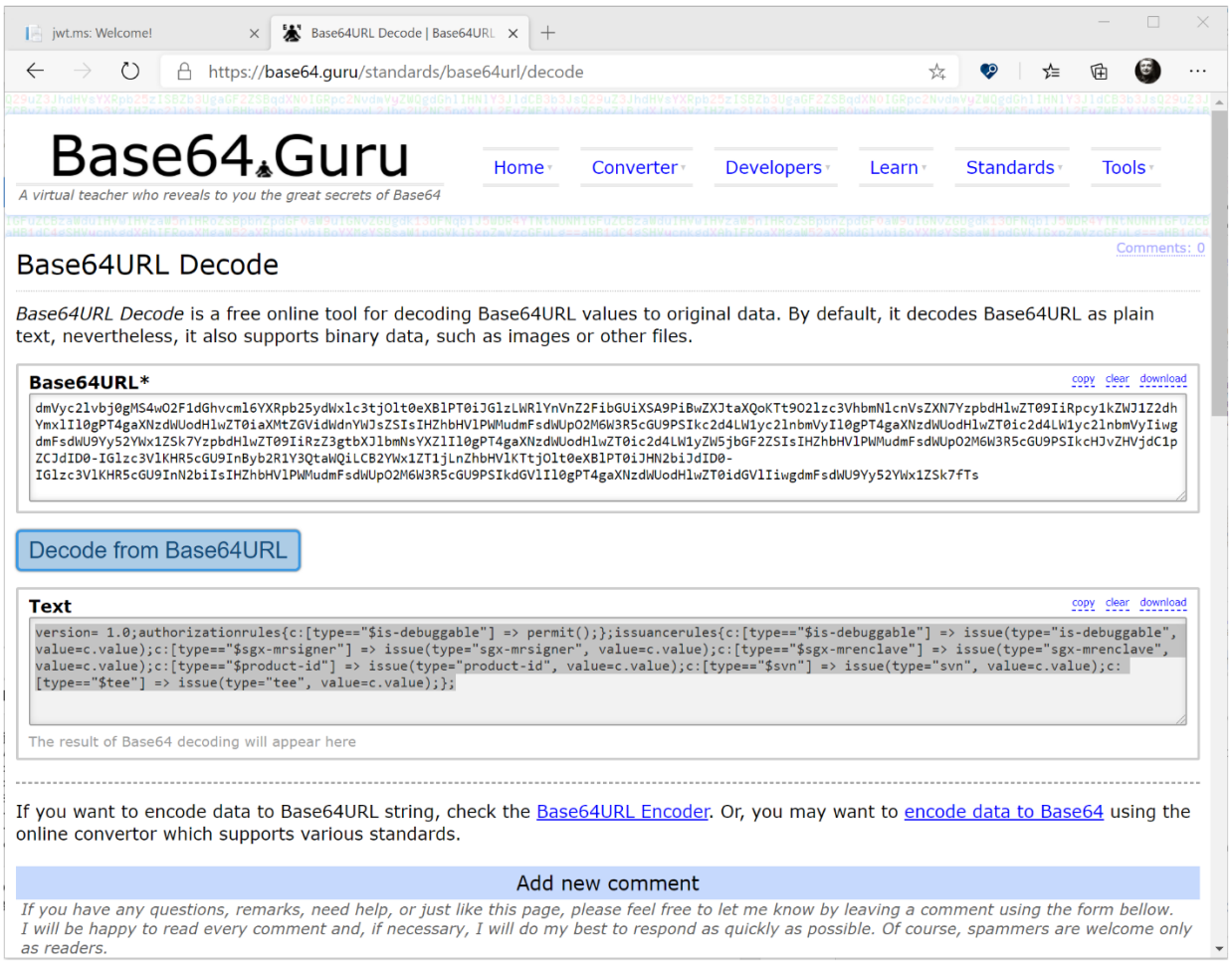
Open a browsing session and navigate to https://jwt.ms.



Copy the value of the `AttestationPolicy` field, which corresponds to the Base64Url decoded attestation policy. For example, in our illustration:

```
dmVyc2lvbj0gMS4wO2F1dGhvcml6YXRpb25ydWxlc3tjOlt0eXBlPT0iJGlzLWRlYnVnZ2FibGUiXSA9PiBwZXJtaXQoKTt9O2lzc
3VhbmNlcnVsZXN7YzpbdHlwZT09IiRpcy1kZWJ1Z2dhYmxlIl0gPT4gaXNzdWUodHlwZT0iaXMtZGVidWdnYWJsZSIsIHZhbHVlPW
MudmFsdWUpO2M6W3R5cGU9PSIkc2d4LW1yc2lnbmVyIl0gPT4gaXNzdWUodHlwZT0ic2d4LW1yc2lnbmVyIiwgdmFsdWU9Yy52YWx
1ZSk7YzpbdHlwZT09IiRzZ3gtbXJlbmNsYXZlIl0gPT4gaXNzdWUodHlwZT0ic2d4LW1yZW5jbGF2ZSIsIHZhbHVlPWMudmFsdWUp
O2M6W3R5cGU9PSIkcHJvZHVjdC1pZCJdID0-
IGlzc3VlKHR5cGU9InByb2R1Y3QtaWQiLCB2YWx1ZT1jLnZhbHVlKTtjOlt0eXBlPT0iJHN2biJdID0-
IGlzc3VlKHR5cGU9InN2biIsIHZhbHVlPWMudmFsdWUpO2M6W3R5cGU9PSIkdGVlIl0gPT4gaXNzdWUodHlwZT0idGVlIiwgdmFsd
WU9Yy52YWx1ZSk7fTs
```

From the browser session, open another tab and navigate to https://base64.guru/standards/base64url/decode. Paste the Base64Url decoded attestation policy, click **Decode from Base64URL**, and then show the decoded policy.



Creating a user defined attestation policy

The attestation policy contains properties that need to be attested, alongside the rules that should apply to these properties.

The client TEE based application making the attestation call sends attestation evidence which the service parses and converts into incoming claims (set of properties, value).

Then the attestation provider processes the claims, based on what's defined in the attestation policy, and returns the computed result.

Understanding the components of an attestation policy

As illustrated in the above screenshot, an attestation policy file is structured as follows:

```
version= 1.0;
authorizationrules{c:[type=="$is-debuggable"] => permit();};
issuancerules{
  c:[type=="$is-debuggable"] => issue(type="is-debuggable", value=c.value);
  c:[type=="$sgx-mrsigner"] => issue(type="sgx-mrsigner", value=c.value);
  c:[type=="$sgx-mrenclave"] => issue(type="sgx-mrenclave", value=c.value);
  c:[type=="$product-id"] => issue(type="product-id", value=c.value);
  c:[type=="$svn"] => issue(type="svn", value=c.value);
```

```
  c:[type=="$tee"] => issue(type="tee", value=c.value);
};
```

A policy file has 3 segments:

1. `Version`,
2. `Authorizationrules`,
3. `Issuancerules`.

Segment	**Description**
Version	`Version=MajorVersion.MinorVersion` The version is the version number of the grammar. **Currently the only version supported is version 1.0.**
Authorizationrules	The authorization rules are a collection of claim rules that will be checked first, to determine if the attention service should proceed to `Issuancerules`. The claim rules apply in the order they are defined.
Issuancerules	The issuance rules are a collection of Claim rules that will be evaluated to add additional information to the attestation result as defined in the policy. The claim rules apply in the order they are defined.

Using the available properties for attesting SGX enclaves / Open Enclave

MAA cryptographically validates multiple properties of the SGX platform, including the microcode, firmware, and software components.

The following client application properties that can be used for attestation are as follows:

Property	**Description**
tee	The type of Trusted Execution Environment
sgx-mrsigner	A hash of the public key used for signing the enclave library
sgx-mrenclave	A cryptographic hash of the enclave log (measurement) as it goes through every step of the build and initialization process
is-debuggable	Indicates whether the enclave library supports debugging.
product-id	An identifier of the enclave library (each library signed with a given key has a unique product-id)
svn	Security version number

The following platform properties that can be used for attestation are as follows:

Property	**Description**
TCB Info	SGX-specific Trusted Computing Base[63] information for a given Intel processor type (e.g. SKU information, PCE identifier).
Quoting Enclave Identity	Identity information for SGX Quoting Enclave issued by Intel
PCK Certificate	Provisioning Certification Key (PCK) certificate for a specific SGX enclave-enabled machine with a specified TCB level (listed in the TCB Info structure.)

[63] Trusted Computing Base: https://en.wikipedia.org/wiki/Trusted_computing_base

PCK Revocation List	Certificate Revocation List (CRL) of all revoked PCK Certificates. This CRL is issued by Intel CA.

Understanding the policy language and the claim rule grammar

MAA provides administrators with the option to define custom rules that they can use to determine the behavior of attestation claims with the claim rule grammar.

You can use the claim rule grammar syntax examples to create a custom rule that enumerates, adds, deletes, and modifies claims to meet the needs of your organization:

- Rules are separated from each other with semicolons.
- Each claim rule needs to be enclosed in [].
- Action needs to be followed by ().
- A prefix(string) followed by ':' can be used to identify a claim in the claim rule, for later usage in the policy. For example:

```
F1:[type==”OSName” , issuer==”CustomClaim”] &&
C2:[type==”OSName” , issuer==”AttestationService”, value== F1.value ]
=> issue(claim=C2);
```

The following are the operators that can be used to check conditions:

Value type	**Operations Supported**
Integer	== (equals), != (not equal), <= (less than or equal), < (less than), >= (greater than or equal), > (greater than)
String	== (equals), != (not equal)
Boolean	== (equals), != (not equal)

The claim rule grammar consists of the following components, separated by the "=>" operator:

- `Authorizationrules`. You can use conditions in a rule to check input claims and determine whether the *Issuancerules* statement of the policy should be executed (permit, deny).

Action Verb	**Description**
Permit	The incoming claim set can be used to compute the `Issuancerules`. Does not take any claim as a parameter.
Deny	The incoming claim set will not be used to compute the `Issuancerules`. Does not take any claim as a parameter.
Add	Add the claim to the incoming claim set. Any claim added to the incoming claim set will be available for the subsequent claim rules.

- `Issuancerules`. In an issuance rule, claims rules are processed based on the issuance statements (issue, add, issueproperty) that you program into the claim rule.

Action Verb	**Description**
Add	Add the claim to the incoming claim set. Any claim added to the incoming claim set will be available for the subsequent claim rules.
Issue	Add the claim to the incoming claim set and to the result.

	Any claim issued to the incoming claim set will be available for the subsequent claim rules.
issueproperty	Special properties on the attestation result can be set using the Issueproperty. The only supported property is *report_validity_in_minutes* which defines for how long the outgoing claim set report is valid for.

Understanding what a claim is

To understand the claim rule grammar, in context of the MAA and your attestation provider, it is important to understand what a claim is.

A claim is a set of properties, that convey information. A claim can be visualized as below.

```
[type="TEE", value="True" , valueType="String", issuer="AttestationService"]
```

A claim contains the following properties:

- `type`. The name of the property.
- `value`. The value of the property.
- `valueType`. The data type of information stored in the value property. Supported types are String, `Integer`, `Boolean`.

Note if not defined the default value is `String`.

- **issuer**. Information(string) regarding the issuer of the claim. The issuer can be one of the following:
    - `AttestationService`. Attestation Service generates a set of claims after parsing the attestation evidence and adds it to the incoming claim set. The issuer in this case is set as `AttestationService`.
    - `AttestationPolicy`. The attestation policy (as defined by the administrator) itself can add claims to the incoming claim set during processing. The issuer in this case is set as `AttestationPolicy`.
    - `CustomClaim`. The attestor (client application) can also add additional claims to the attestation evidence. The issuer in this case is set as `CustomClaim`.

Note If not defined the default value is `CustomClaim`.

Creating the attestation policy file JWS

It is thus of utmost importance that the attestation policy evaluated by the attestation provider for a particular TEE type is in fact the intended user defined attestation policy written by you or the administrators and it has not been tampered or modified by external entities. This protection is achieved by inserting into the attestation report a cryptographic evidence so that the relying party can verify that the policy evaluated by the service is indeed the one that is expected.

The administrator can upload a signed policy object.

After creating an attestation policy file (see previous section), administrators will create an attestation policy file JWS.

The following steps should typically be performed:

1. Generate the JWS with attestation policy file (utf-8 encoded) as the payload. The payload identifier for the base64url[64] encoded policy should be AttestationPolicy.

   Following is a sample JWT:

   Header: `{"alg":"none"}`
   Payload: `{"AttestationPolicy":"base64url(policy file)"}`
   Signature: {}
   Complete JWS Web signature (JWS, RFC 7515): `eyJhbGciOiJub25lIn0.XXXXXXXXX`.

2. Optionally sign the policy, currently MAA supports the following algorithms:

   a. None – When you don't want to sign the policy payload.
   b. RS256 – Supported algorithm to sign the policy payload.

If the trust model is AAD (see section § *Understanding the MAA's trust model* above) meaning that Microsoft is allowed to be operationally *within* the Trusted Computing Base (TCB), customers can specify an unsecured RFC 7515 signature (with a fixed header of '`{"alg":"none"}`). The unsecured signature option allows a customer to specify a JWS without performing any cryptographic operations.

Alternatively, the customer can define a signing key used to sign the certificate and creating an X.509 certificate wrapping that signing key. The customer will then create a JWS using this newly created X.509 certificate. MAA will validate that the signing key was used to sign the JWS.

The signer should be a part of the JWS.

When MAA is operating in "Isolated" trust mode, the service will ensure that the JWS was signed using the key contained in the JWS and that the certificate chain created by the `x5c` field in the JWS header contains one of the configured root policy certificates.

> **Note** This also means that when MAA is configured in "Isolated" mode, it will reject all self-signed certificates and all unsigned JWS documents.

As an illustration, the following Python sample script can be used to perform the above steps:

```
from OpenSSL import crypto
import jwt
import getpass

def cert_to_b64(cert):
              cert_pem = crypto.dump_certificate(crypto.FILETYPE_PEM, cert)
              cert_pem_str = cert_pem.decode('utf-8')
              return ''.join(cert_pem_str.split('\n')[1:-2])

print("Provide the path to the PKCS12 file:")
pkcs12_path = str(input())
pkcs12_password = getpass.getpass("\nProvide the password for the PKCS12 file:\n")
pkcs12_bin = open(pkcs12_path, "rb").read()
pkcs12 = crypto.load_pkcs12(pkcs12_bin, pkcs12_password.encode('utf8'))
ca_chain = pkcs12.get_ca_certificates()
ca_chain_b64 = []
for chain_cert in ca_chain:
              ca_chain_b64.append(cert_to_b64(chain_cert))
```

[64] base64url encoding: https://brockallen.com/2014/10/17/base64url-encoding/

```
 signing_cert_pkey = crypto.dump_privatekey(crypto.FILETYPE_PEM, pkcs12.get_privatekey())
signing_cert_b64 = cert_to_b64(pkcs12.get_certificate())
ca_chain_b64.insert(0, signing_cert_b64)

print("Provide the path to the policy text file:")
policy_path = str(input())
policy_text = open(policy_path, "r").read()
encoded = jwt.encode({'text': policy_text }, signing_cert_pkey, algorithm='RS256', headers={'x5c' :
ca_chain_b64})
print("\nAttestation Policy JWS:")
print(encoded.decode('utf-8'))
```

Uploading the attestation policy file JWS

Once the attestation policy file JWS is created for the user defined attestation policy, it can then be uploaded to the attestation provider.

For the sake of the illustration, let's supposed that you defined an attestation policy for the Open Enclave TEE endpoint, and that the related file JWS is named *userdefined.oe.policy.txt*.

From a PowerShell console, upload your attestation policy file JWS to your attestation provider:

```
PS C:\> $policy = Get-Content -Path .\userdefined.oe.policy.txt
PS C:\> Set-AzAttestationPolicy -Name <your_AttestationProviderName> -ResourceGroupName
<your_ResourceGroupName> -Tee <your_TEEType> -Policy $policy
```

- Replace <*your_AttestationProviderName*> with the name of your choice for your new attestation provider. For example, **maatest** in our illustration.
- Replace **<***your_ResourceGroupName***>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.
- Replace <*your_TEEType*> with one of the supported TEE types: `SgxEnclave`, `OpenEnclave`, `CyResComponent` and `VBSEnclave`. For example, `OpenEnclave` in our illustration

For example, to set the user defined policy for TEE type `OpenEnclave` for the attestation provider maatest in our illustration:

```
PS C:\> $policy = Get-Content -Path .\userdefined.oe.policy.txt
PS C:\> Set-AzAttestationPolicy -Name maatest -ResourceGroupName AzureAttestationTest -Tee
OpenEnclave -Policy $policy
```

Note For more information on the command and its parameters, see Set-AzAttestationPolicy[65].

When the attestation policy file JWS is uploaded, the attestation provider validates the user defined policy:

- If the policy file is free of syntax errors the policy file gets accepted by the provider.

  Once the provider has validated the policy document, it will save the policy in storage and return a JWT signed with the tenant specific signing key which contains a claim named `aas-policyHash` which

[65] Set-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/set-azattestationpolicy?view=azps-4.4.0

contains the hash of the policy file JWS provided request. The value of the aas-policyHash will be the Base64url encoded SHA256 hash of the policy document expressed in the payload request.

If the certificate in the policy file JWS was matched with a configured root certificate, then the JWT will also contain a claim named aas-`policySigningCertificate`, which will contain the policy signing certificate encoded as a JSON Web Key.

- If the policy file contains syntax errors the policy file will be rejected by the provider.

  MAA will reject all policy file JWS that have any header parameters other than `x5c`, `kid`, `jwk` or `alg` – a client MUST provide an `x5c` value for the certificate chain.

Resetting the attestation policies of your attestation provider

Run the following command to reset the attestation policy from a tenant in MAA for an attestation provider:

```
PS C:\> Set-AzAttestationPolicy -Name <your_AttestationProviderName> -ResourceGroupName
<your_ResourceGroupName> -Tee <your_TEEType>
```

- Replace <*your_AttestationProviderName*> with the name of your choice for your new attestation provider. For example, **maatest** in our illustration.
- Replace **<***your_ResourceGroupName***>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.
- Replace <*your_TEEType*> with one of the supported TEE types: `SgxEnclave`, `OpenEnclave`, `CyResComponent` and `VBSEnclave`. For example, `OpenEnclave` in our illustration

For example, to reset the attestation policy for the TEE type `OpenEnclave` for the attestation provider **maatest** in our illustration:

```
PS C:\> Reset-AzAttestationPolicy -Name maatest -ResourceGroupName AzureAttestationTest -Tee
OpenEnclave
```

Note For more information on the command and its parameters, see Set-AzAttestationPolicy[66].

Using attestation with attestation providers

Sending an attestation request

MAA, and more specifically, Microsoft provided attestation default providers or your own attestation providers will accept an input set of properties, evaluate that evidence based on a policy and emit a set of claims in the form of a JWT token.

The JWT token is signed so that a relying party can validate the token. Relying party needs to validate that MAA issued the token.

[66] Set-AzAttestationPolicy: https://docs.microsoft.com/en-us/powershell/module/az.attestation/set-azattestationpolicy?view=azps-4.4.0

Furthermore, the relying party can optionally validate that the expected attestation provider, i.e. the right instance of MAA was used to issue the token. Every attestation provider is tied to a tenant and each attested provider has its own policy for the different TEE types as you now understand.

This way a relying party can confirm that the JWT token was issued by an attestation provider that belongs to Tenant 'A' with Policy 'B'.

For that purpose, the issued JWT tokens by MAA contains `iss` (issuer) field in the payload. The `issuer` field contains the base URI of the attestation provider. The base URI binds the token to an instance of MAA.

Leveraging an attestation's response

Getting and decoding the JWT token

As mentioned, the JWT token is signed so you can also validate that the token is originated from a trusted attestation provider.

Validating the JWT token signature

You, as a developer, or a relying party can typically validate the issued attestation token by:

1. Extracting the value of the `iss` field and appending */.well-known/openid-configuration* to form the URL of the OpenID Metadata endpoint.
2. Retrieving the metadata and extract the `jwks_uri` endpoint, and then using that URL to retrieve the token signing keys
3. Or directly retrieving the keys form `jwks_url` from the `iss` claim as it too is well known endpoint (You will need to append */certs* to base URL from `iss` field).
4. Verifying the signature of the token using one of the signing keys from the `jwks_uri` endpoint.
5. Extracting the MAA enclave quote from the cert extension
6. Validating the MAA enclave quote

This allows to determine that:

1. The attestation token was issued by MAA. OpenID metadata endpoint and `jwks_uri` endpoint are protected using TLS.
2. Optionally, the token was issued by MAA TEE with the signing key generated in the TEE.

Optionally, you, as a developer, or the relying party can also determine if the attestation token was issued by the expected attestation provider, i.e. the right instance of MAA. You can check the tenant information in the base URI to determine the particular instance of the attestation provider that issued the JWT token.

Deleting your own attestation provider

Eventually, when you no longer need a previously created attestation provider (see section § *Creating your own attestation provider* above, you can delete it.

```
PS C:\> Remove-AzAttestion -Name <your_AttestationProviderName> -ResourceGroupName
<your_ResourceGroupName>
```

- Replace <*your_AttestationProviderName*> with the name of your choice for your new attestation provider. For example, **maatest** in our illustration.
- Replace **<***your_ResourceGroupName***>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.

For example, to remove the attestation provider **maatest** in our illustration:

```
PS C:\> Remove-AzAttestion -Name maatest -ResourceGroupName AzureAttestationTest
```

Note For more information on the command and its parameters, see Remove-AzAttestationPolicy[67].

[67] Remove-AzAttestation: https://docs.microsoft.com/en-us/powershell/module/az.attestation/remove-azattestation?view=azps-4.4.0

Developing TEE-based applications using attestations

To build the example scenarios described in the rest of the document, we were inspired by two examples of *file_encryptor* and *remote_attestation* code from the Open Enclave SDK available on the GitHub repo at the URL https://github.com/openenclave/openenclave/tree/v0.8.1.

The primary objective of this document is to deepen the attestation process. We wanted to illustrate this by basically implementing the transfer of encrypted data to an enclave, a necessary step for the implementation of the Multi-party Machine Learning scenario described below.

The attestation process allows a party to validate that it is communicating with a real enclave but also, through the exchange of attestation structures, to transmit information (for example encryption keys) which will then make it possible to transfer encrypted data to the enclave. The three proposed scenarios constitute a progression in the use of attestations to serve the same type of scenario but in cases closer to reality.

Disclaimer

The code style from the two samples has been respected as much as possible for the development of the new scenarios presented below.

Please note, this implementation should only be considered as an example allowing a better understanding in using the remote attestation process in a simplified scenario of data transfer between several parties respecting confidentiality. **These are by no means examples of implementation to be put into production**.

Introducing the Multi-party Machine Learning use case

This scenario, which demonstrates how to share of information between several parties without anyone revealing their own data to the others, is one of the emblematic examples made possible by confidential computing.

The scenario is the following: several hospitals have their own data sets from the patients they treat. They wish to develop a machine learning model to be able, for example, to make analyzes from patients' radiographies or scanners. This model should allow them to detect abnormalities that would be difficult to detect by humans, and thus help doctors in their diagnosis.

However, each hospital has an insufficient data set to effectively train a model and does not wish to share its own data with other hospitals, both for compliance purposes concerning the protection of health data and for competitive aspects such as this may be the case for private hospitals.

The solution, however, would be to share the data from all hospitals to train the model so that everyone can request the model afterwards for future detections. Confidential calculation can meet these needs by authorizing the sharing of data for common processing without disclosing their own data to the other parties.

The implementation is based on the features offered by enclaves to carry out processing on data which is sent to them without the data or the processing being visible from outside.

However, to set up this scenario:

- Each party must be responsible beforehand for encrypting its data before transmitting it to the enclave, to ensure that it cannot be visible either by the other parties or by the cloud provider;
- It must be possible to exchange with the enclave the encryption key used to protect the data : this will allow the enclave to decrypt the data to carry out the training of the model or inference on the proposed data.

The principle of implementation is as follows:

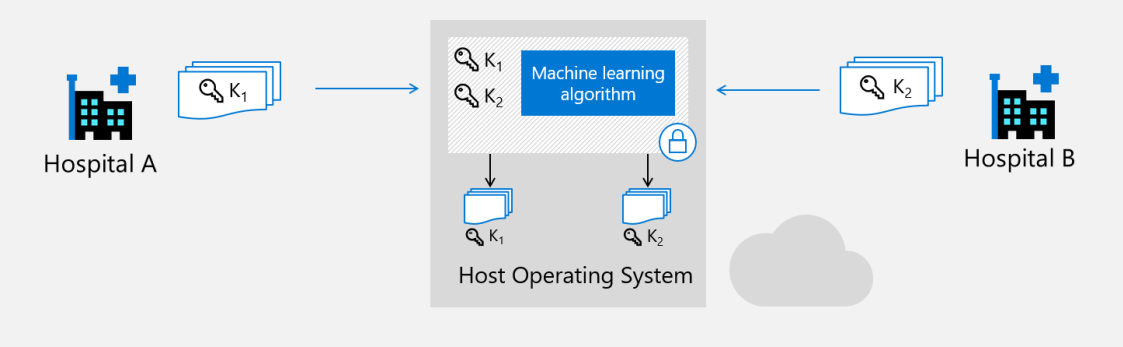


Figure 7 Multi party-Machine Learning use case

Hospital A locally has a set of data that it wishes to submit for training the machine learning model. It encrypts its data locally with a symmetric key (in reality, block encryption will most certainly use a set of symmetric keys). The encrypted data is then copied to a cloud storage accessible by the enclave.

To start the processing, the client part of the hospital A application communicates with the enclave which is running on a server in the datacenter and asks the enclave for an attestation to ensure that it communicates with a genuine enclave. This communication will be the preferred means for the enclave to transfer the public part of a key pair inside the attestation (the asymmetric key pair has been generated previously inside the enclave and does not need to be sealed).

After validation of the attestation by requesting an attestation service, the hospital A application encrypts the symmetric key used to encrypt the data with the public key of the enclave and returns it to the enclave.

The enclave is then able to use its private key to decrypt the symmetric key. The data available on the cloud storage is retrieved by the enclave which can decrypt it with the symmetric key and train the machine learning model running confidentially inside the enclave.

Hospital B follows the same process, allowing to train the model without any party revealing its data.

Once trained, the model can be used in the same way by transferring to the enclave the data to be processed in an encrypted manner and recovering later on the results of the inference in complete confidentiality.

The 3 scenarios

The 3 example scenarios that are implemented bring a certain progression in the use of attestations for the transfer of encrypted data to an enclave:

1. Secure data transfer between enclaves running on the same host based on a remote attestation; it could have been possible to rely on a local attestation but this choice allows to test the remote attestation to prepare the following scenario.

2. Secure data transfer between enclaves running on different hosts with remote attestation and communication between hosts across the network using the gRPC protocol.
3. Data transfer triggered by a client application not running in an enclave and leveraging Azure Attestation Service to validate the enclave and send it an encryption key to decrypt the data to be processed.

Each scenario is explained with pseudo-code based on a diagram. Then sequence diagrams (in format inspired by UML) are proposed to describe the initialization and key exchange part through the attestation process.

Simplified view of the process

The diagram below explains the basic principle used to transfer encrypted data between two enclaves. This same principle will then be used regardless of the nature of the client application (enclave or not) and the type of attestation service used.

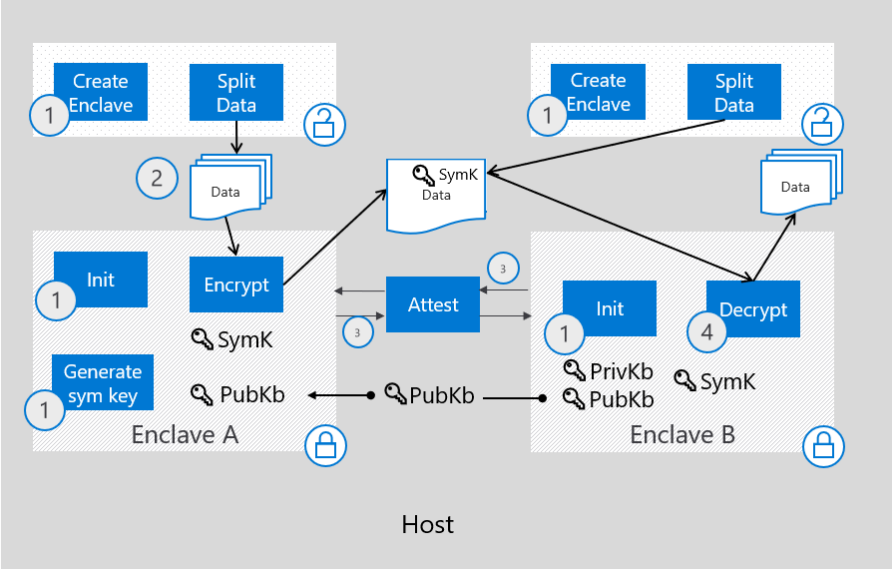


Figure 8 Transferring encrypted data between enclaves

- During step 1 of the scenario, two enclaves, enclave A and enclave B are created on an SGX host and their initialization code is launched.
- Enclave A generates a symmetric key and then retrieves the data to be transferred to encrypt it with this key. If the data is large, it is cut into blocks (step 2).
- For its part, enclave B generates an asymmetric key pair and waits for receiving a request.
- Enclave A establishes a communication channel with enclave B and requests an attestation (step 3).
- The public key of enclave B is transmitted to enclave A which uses it to encrypt the symmetric key previously used to encrypt the data.
- The encrypted symmetric key is transmitted to Enclave B which decrypts it with the RSA private key and uses it to decrypt the data before processing.

Scenario 1: Exchanging encrypted data between local enclaves

Overview of the scenario

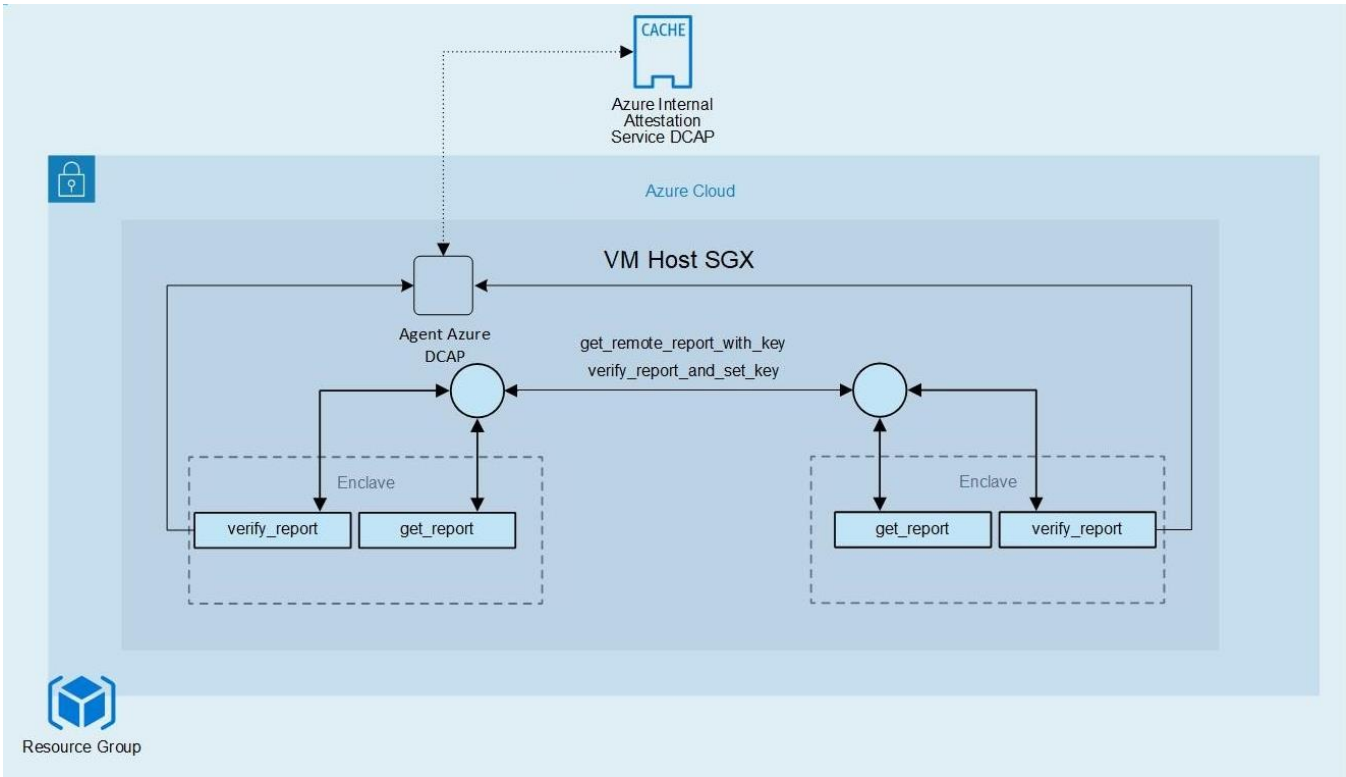


Figure 9 Exchanging encrypted data between enclaves

This first scenario uses the model described above with the implementation of two enclaves on the same host but based on a remote attestation. However, a small additional complexity was added related to a limitation, in the SGX implementation, of the size of the information that can be transmitted in the attestation.

Indeed, the size of the report_data field is limited to 64 bytes, which does not allow to transmit an RSA 2048 key which requires 256 bytes. It is therefore not possible to directly convey the public key generated by the destination enclave in the attestation. The workaround is to generate a hash of the RSA key, specifically a SHA256 that fits in 64 bytes and that will be stored in the report_data field of the attestation.

The RSA public key will therefore be transmitted via the communication channel, without any particular protection. To verify that the transmitted public key was not changed during the transfer, Enclave A will just have to recalculate the hash of the public key and compare it to the hash transmitted in the attestation.

The remote attestation relies on a DCAP client made available by the Open Enclave SDK and installed on the SGX host machines which allows to access to the Azure Internal Attestation Service, without the machines having to be connected to internet.

Note This solution is inspired by what is described in the example Remote Attestation Sample du SDK Open Enclave[68].

1. At the beginning, the application running on the host creates the two enclaves, enclave A and enclave B, and their initialization code is launched.
2. Enclave A generates an AES 256-bit symmetric key, retrieves a file from the disk and encrypts its data with this key. This encrypted data is transferred to the host's memory in the untrusted part (which is not a problem since it is encrypted).
3. For its part, enclave B generates an RSA 2048-bits key pair and waits for a request. Note that the private key part of the key pair remains stored in the enclave's memory.
4. Enclave A establishes a communication channel with enclave B and requests an attestation from it.
5. Enclave B computes a hash (SHA 256) of its RSA public key and generates an attestation (REPORT structure) where this hash is included in the report_data of the attestation. The attestation is returned to enclave A, as well as the public key (outside of the attestation).
6. Enclave A requests a validation of the received attestation. If the return is OK, this confirms the validity of enclave B and guarantees the integrity of the information contained in the attestation structure.
7. Enclave A computes the hash of the public key and compares it with the hash value transmitted in the report_data field of the attestation. Identical values confirm that the public key was not altered during its transfer.
8. Enclave A encrypts the AES private key with the RSA public key of enclave B. It calculates the hash of the encrypted symmetric key and generates an attestation in which this hash is inserted in the report_data field.
9. The attestation is finally returned to enclave B along with the encrypted symmetric key.
10. Enclave B recomputes the hash on the encrypted symmetric key and compares it to report_data value. If there is a match, enclave B decrypts the symmetric key with its private key and then uses it to decrypt the data.

[68] The Remote Attestation Sample https://github.com/openenclave/openenclave/tree/master/samples/remote_attestation#remote-attestation-sample

Scenario 1 workflow diagram

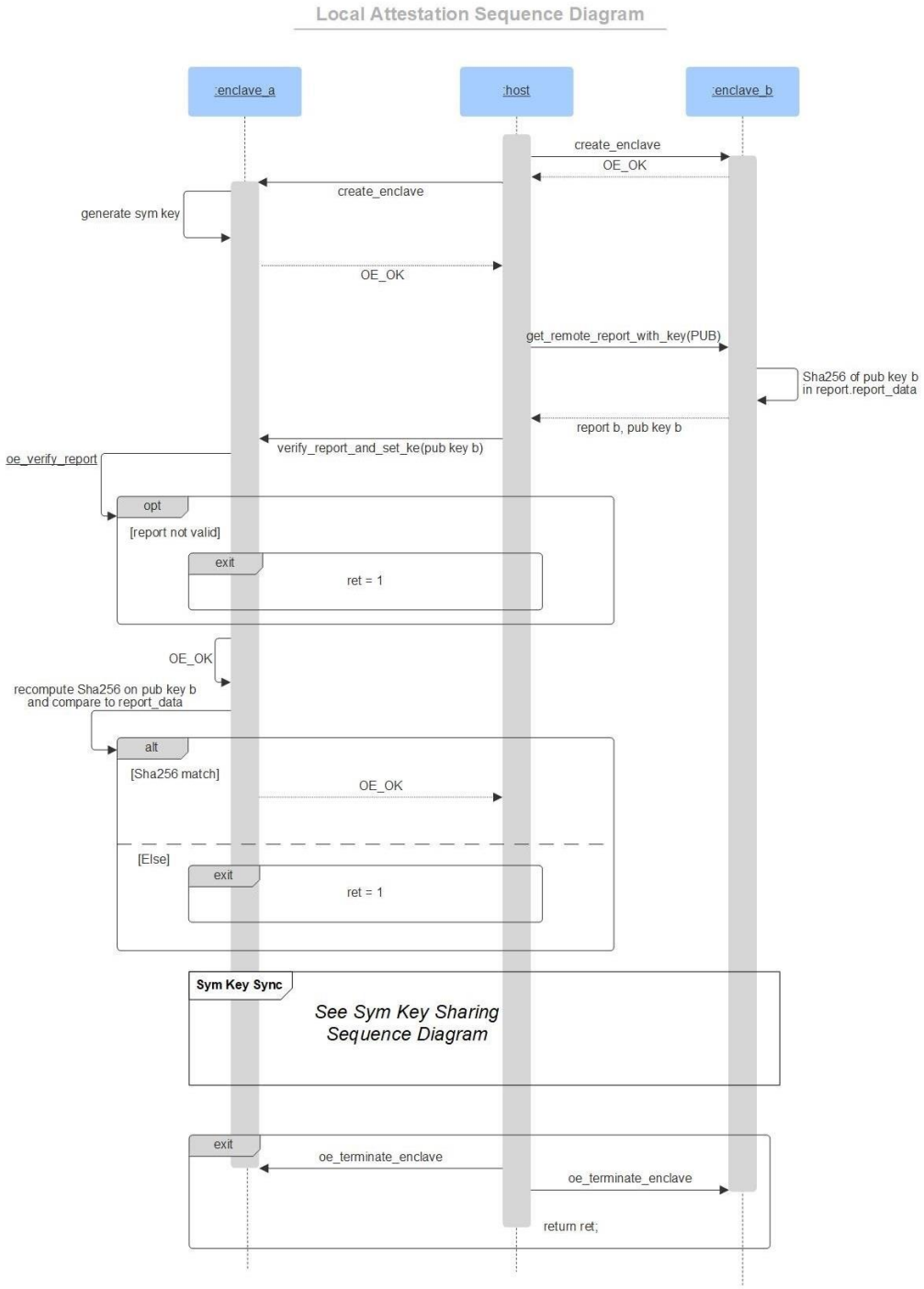


Figure 10 Scenario 1: Local Attestation Diagram

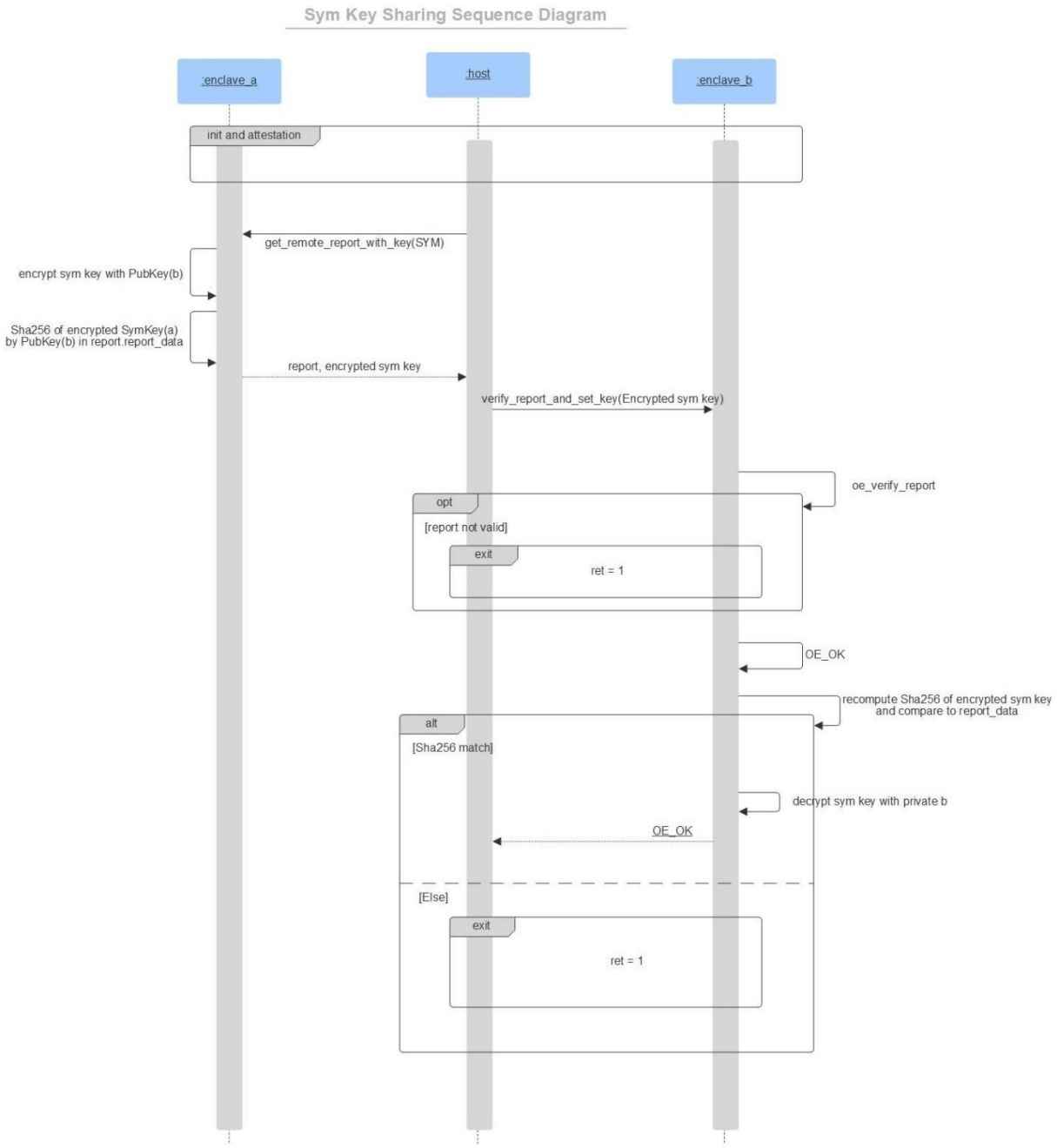


Figure 11 Scenario 1: Symmetric Key Sharing Diagram

A sample code walkthrough for the scenario

In this sample, the host application does the following:

1. Create the two enclaves, enclave_a and enclave_b:

```
oe_create_secretsharinge_enclave( enclaveImagePath, OE_ENCLAVE_TYPE_SGX, OE_ENCLAVE_FLAG_DEBUG, NULL,
0, &enclave);
```

2. Ask enclave_b for a remote report and a public key:

```
int get_remote_report_with_key(
    KeyKind key_kind,
    uint8_t** key,
    size_t* key_size,
    uint8_t** remote_report,
    size_t* remote_report_size);
```

Where:

- `key_kind` set to `KeyKind::PEM`
- key will hold the public key that identifies enclave_b
- `remote_report` will contain a remote report signed by the enclave platform for use in remote attestation

Implementation of `get_remote_report_with_key` calls `oe_get_report` OE_CALL, which generates the report. Then the hash of the public key is computed and stored it in the report_data. The enclave allocates space and copies its public key into the host memory, addressed by key pointer.

3. Ask enclave_a to attest (validate) enclave_b's remote report. This is done through the following call:

```
int verify_report_and_set_key(
    KeyKind key_kind,
    uint8_t* key,
    size_t key_size,
    uint8_t* remote_report,
    size_t remote_report_size);
```

Where:

- `key_kind` set to `KeyKind::PEM`
- key holds the public key that identifies enclave_b
- `remote_report` contains the remote report to verify

Implementation of `verify_report_and_set_key` calls `oe_verify_report` OE_CALL, which verifies the report, and recomputes the hash of the key to compare it to the report data.
The public key is then copied from host to enclave's memory for future use.

Now the secure channel is set. Step 2 and 3 a repeated to ask symmetrically enclave_b to validate enclave_a. This is represented in the *Sym Key Sharing Sequence Diagram*.

4. Ask enclave_a for a remote report and its symmetric key encrypted by enclave_b public key. This is done through the following call: `get_remote_report_with_key`

Where:

- `key_kind` set to `KeyKind::AES`
- key holds the public key that identifies enclave_b
- `remote_report` contains a remote report signed by the enclave platform for use in remote attestation

With `key_kind` set to AES, the symmetric key is encrypted by the previously set public key, then the hash of this data is included into the generated report.

5. Ask enclave_b to attest (validate) enclave_a's remote report. This is done through the following call: verify_report_and_set_key where:

    - `key_kind` set to `KeyKind::AES`
    - key holds the encrypted symmetric key
    - `remote_report` contains the remote report to verify

    With `key_kind` set to AES, on top of verifying the hash from the `report_data`, the key data from host memory is decrypted and stored in enclave's memory.

    Now both enclaves share the same symmetric key and can encrypt/decrypt data.

6. Free the resource used, including the host memory allocated by the enclaves and the enclaves themselves. For example:

```
oe_terminate_enclave(enclave_a);
oe_terminate_enclave(enclave_b);
```

Scenario 2: Exchanging encrypted data between remote enclaves

Overview of the scenario

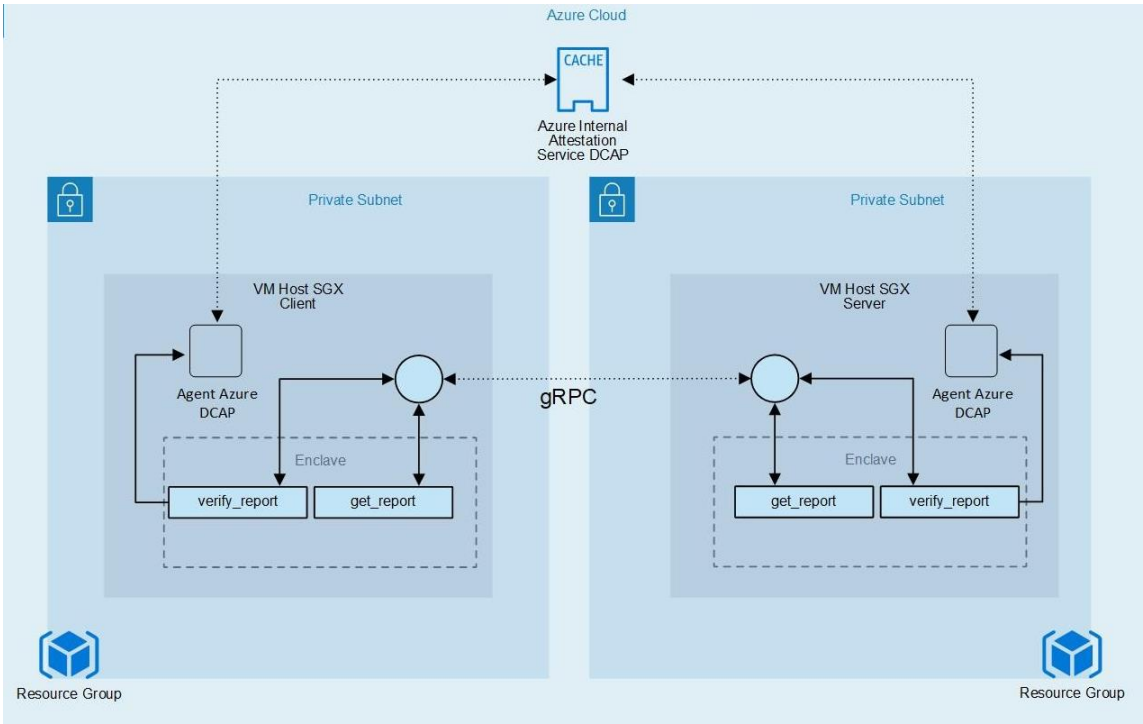


Figure 12 Exchanging encrypted data between remote enclaves

This scenario is like the previous scenario with the difference that the enclaves run on different hosts. This implies that communication between the two enclaves is done through the untrusted parts of the applications that communicate across the network.

The choice was made to use gRPC for its efficiency (it relies on HTTP/2) and the fact it is implemented in many languages which will be useful for the following scenario. In addition, gRPC integrates Protocol Buffers (Protobuf) and its interface description language as a way of serialization.

As for the previous scenario, the remote attestation relies on a DCAP client that provides access to the Azure Internal attestation service for internal hosts.

Note For more information about gRPC and Protobuf, see gRPC[69].

[69] gRPC: https://grpc.io/

Scenario 2 workflow diagram

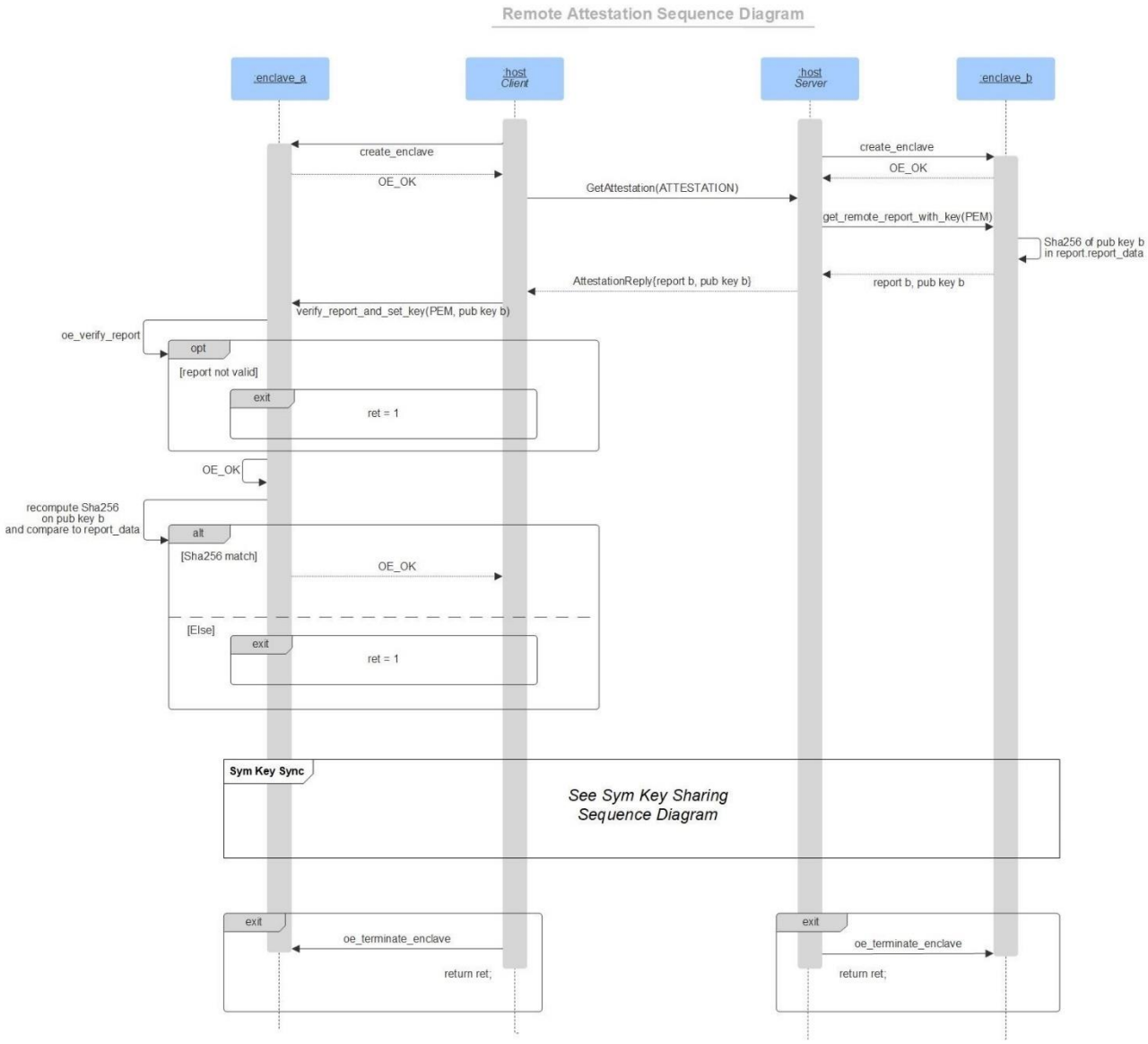


Figure 13 Scenario 2: Remote Attestation Sequence Diagram

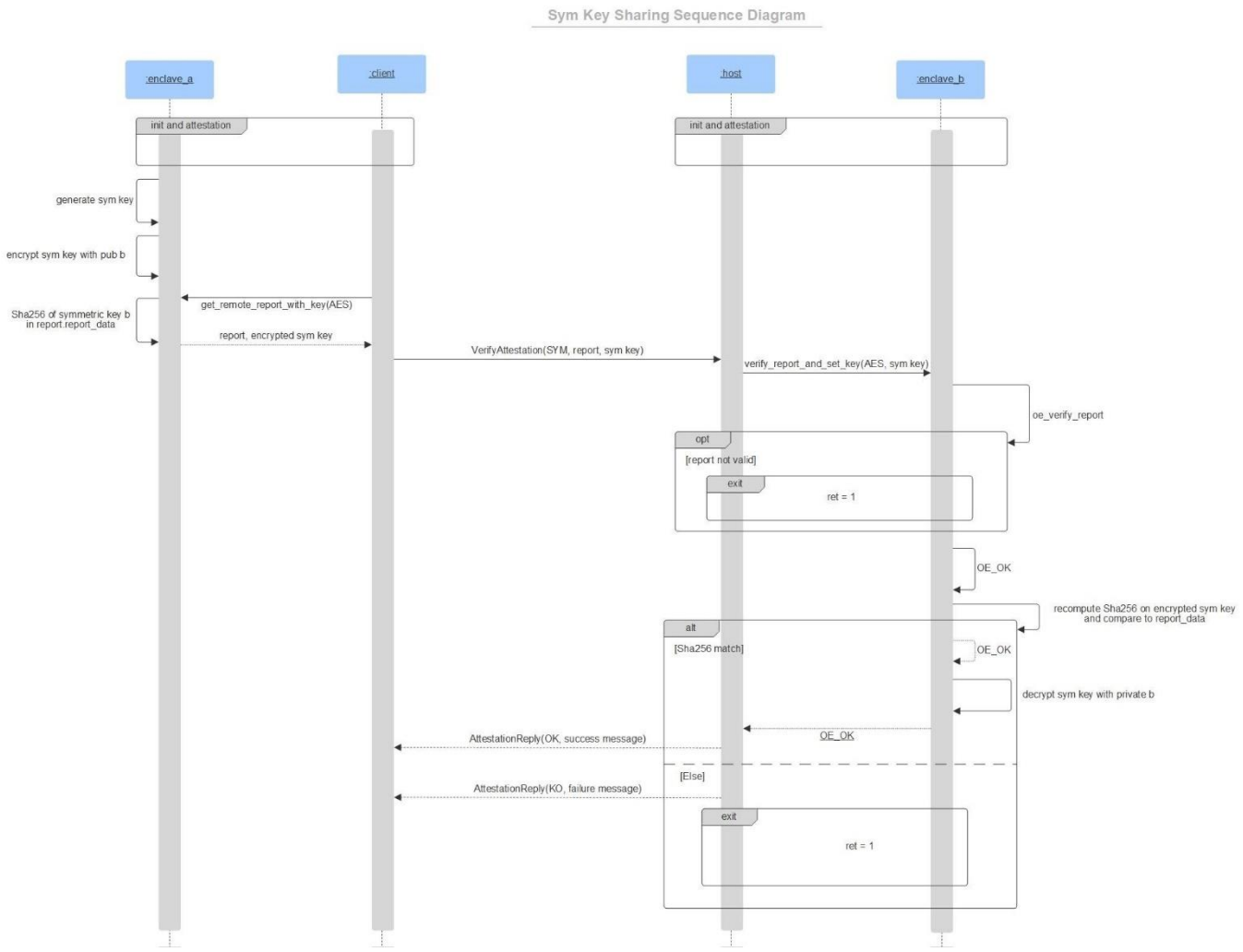


Figure 14 Scenario 2: Symmetric Key Sharing Diagram

A sample code walkthrough for the scenario

The host client and host server do the following in this sample:

1. Client and Server each creates an enclave:

```
oe_create_secretsharinge_enclave( enclaveImagePath, OE_ENCLAVE_TYPE_SGX, OE_ENCLAVE_FLAG_DEBUG, NULL,
0, &enclave);
```

2. Server runs gRPC service and listens for connection:

```
std::string address("0.0.0.0:5000");
SecretSharingServiceImplementation service;

ServerBuilder builder;

builder.AddListeningPort(address, grpc::InsecureServerCredentials());
```

```
builder.RegisterService(&service);

std::unique_ptr<Server> server(builder.BuildAndStart());
server->Wait();
```

3. Client connects to gRPC Server:

```
std::string address("0.0.0.0:5000");
SecretSharingClient client(
  grpc::CreateChannel(
      address,
      grpc::InsecureChannelCredentials()
  )
);
```

Where `SecretSharingClient` client is the gRPC client stub.

4. Client requests server's remote attestation:

```
attestation_data_t at_data{PEM, NULL, 0, NULL, 0};
if (client.GetAttestation(CommandRequest{CommandRequest::ATTESTATION}, at_data))
         return 1;
```

Where:

- `CommandRequest::ATTESTATION` is an enum to ask for an attestation
- `at_data` is attestation structure of server received by network

5. Server gets remote report and public key:

```
if (get_remote_report(enclave, at_data)) {
   return Status(StatusCode::INTERNAL, "get_remote_attestation failed.");
}

reply->set_ok(true);
reply->set_key_kind(at_data.key_kind);
for (int i = 0; i < at_data.remote_report_size; i++) {
   reply->add_report(at_data.remote_report[i]);
}
for (int i = 0; i < at_data.key_size; i++) {
   reply->add_key(at_data.key[i]);
}
reply->set_msg("Remote report generated successfully.");
```

Where:

- `at_data` is the attestation structure to be sent over the network.
- `AttestationRequest* request` is the Protobuf message object for client's request data.
- `AttestationReply* reply` is the Protobuf message object for serialization of attestation data.
- `remote_report` will contain a remote report signed by the enclave platform for use in remote attestation.

Implementation of `get_remote_report` calls `oe_get_report` `OE_CALL`, which generates the report and computes the hash of the public key to be stored in the `attestation_data_t` structure. The enclave allocates space and copies its public key into host memory, addressed by `at_data.key` pointer.

6. Client has received the server's attestation data and verifies the report.

   This is done through the following call:

```
if (verify_remote_report(enclave, at_data))
   return 1;
```

   Where `at_data` is server's attestation structure to be verified.

   Implementation of `verify_report` calls `oe_verify_report` `OE_CALL`, which verifies the report and recomputes the hash of the `at_data.key` to compare it to the report data. The public `at_data.key` is then copied from host to enclave's memory for future use.

   Now the secure channel is set. The client's symmetric key is sent to the Server.

7. Client gets its enclave's remote report. As server did, attestation data is set in a structure.

   This is done through the following call: `get_remote_report`

   Where `at_data.key_kind` set to `KeyKind::AES`

   With `key_kind` set to AES, the symmetric key is encrypted by the previously set public key (from the Server's enclave). Then the hash of the encrypted data is set into the generated report.

8. The Client then requests the Server to verify its report.

   This is done through the following call: `client.VerifyAttestation`

   Where `CommandRequest` is set to `CommandRequest::SYNC_SYM`

   `CommandRequest::SYNC_SYM` is for setting up the symmetric to the server. That way syncing the en/decryption of both hosts.

9. The Server verifies the remote report and sets the AES key.

```
parse_request(at_data, request);
if (verify_remote_report(enclave, at_data)) {
   return Status(StatusCode::INTERNAL, "get_remote_attestation failed.");
}
```

   Where:

   - `parse_request()` parses the protobuf message to `attestation_data_t`
   - `at_data` is the received attestation data

   After validation of the remote report, the attestation data is decrypted with the enclave's private key. The data, which is now the AES key is stored in enclave's memory. Both enclaves share now the same symmetric key and can encrypt/decrypt data.

10. Free the resource used, including the host memory allocated by the enclaves and the enclaves themselves. For example:

```
oe_terminate_enclave(enclave_a);
```

```
oe_terminate_enclave(enclave_b);
```

Scenario 3: Attesting a remote enclave using Microsoft Azure Attestation

Overview of the scenario

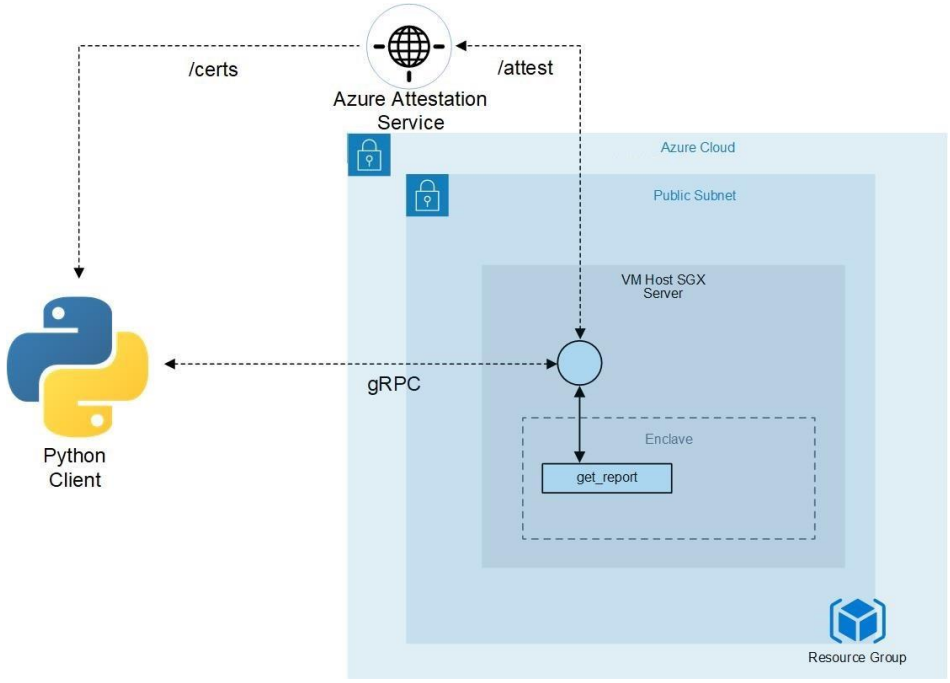


Figure 15 Attesting a remote enclave using Microsoft Azure Attestation

In the previous scenario we have seen how we can transmit reports through the network to be attested between two enclaves. Here we go further by approaching a real case where a client part, which does not require to be executed in an enclave – therefore does not impose an SGX machine – relies on a server part which is responsible for processing data in an enclave.

In addition, the validation of attestations is no longer based on the DCAP client supplied with the Open Enclave SDK, but on the Microsoft Azure Attestation service (MAA) which supports, among other things, the definition of attestation policies which determine if the attestation provider shall issue an attestation token based on particular rules.

We have chosen to write the client application in Python taking advantage of the fact that the communication part supported by gRPC is available for this language.

For the sake of simplicity, the secret sharing mechanism and the encryption of data have not been detailed in this part.

The sample follows these simple steps:

1. The server part creates an enclave.

2. During enclave initialization, an asymmetric key (public and private key) is generated. These keys are kept in the enclave's memory. The server connection is initiated and listens for connection requests.
3. The Python client connects to the server and requests an attestation.
4. The server generates the report, like in scenario 1 and 2, computes the hash of its public key and adds it to report data.
5. The server then prepares the request to the attestation service by filling a JSON object field Quote with the report and EnclaveHeldData with the public key. After getting Azure Active Directory (AD) bearer token, this request is sent over HTTP to AAS, which returns a JSON Web Token (JWT). The token is signed by the service.
6. The server forwards the token to the client.
7. The client collects MAA public certificates to validate the attestation.
8. Upon collection, the JWT signature can be verified as well as the attestation issuer.

To go even more further, one can imagine that symmetric key as used in scenario 1 and 2 could be stored in Azure Key Vault (AKV) and the data to be decrypted stored encrypted in an Azure Storage. In this way, the server could pull data directly from the Azure storage and transferred it to the enclave to be processed securely.

Scenario 3 workflow diagram

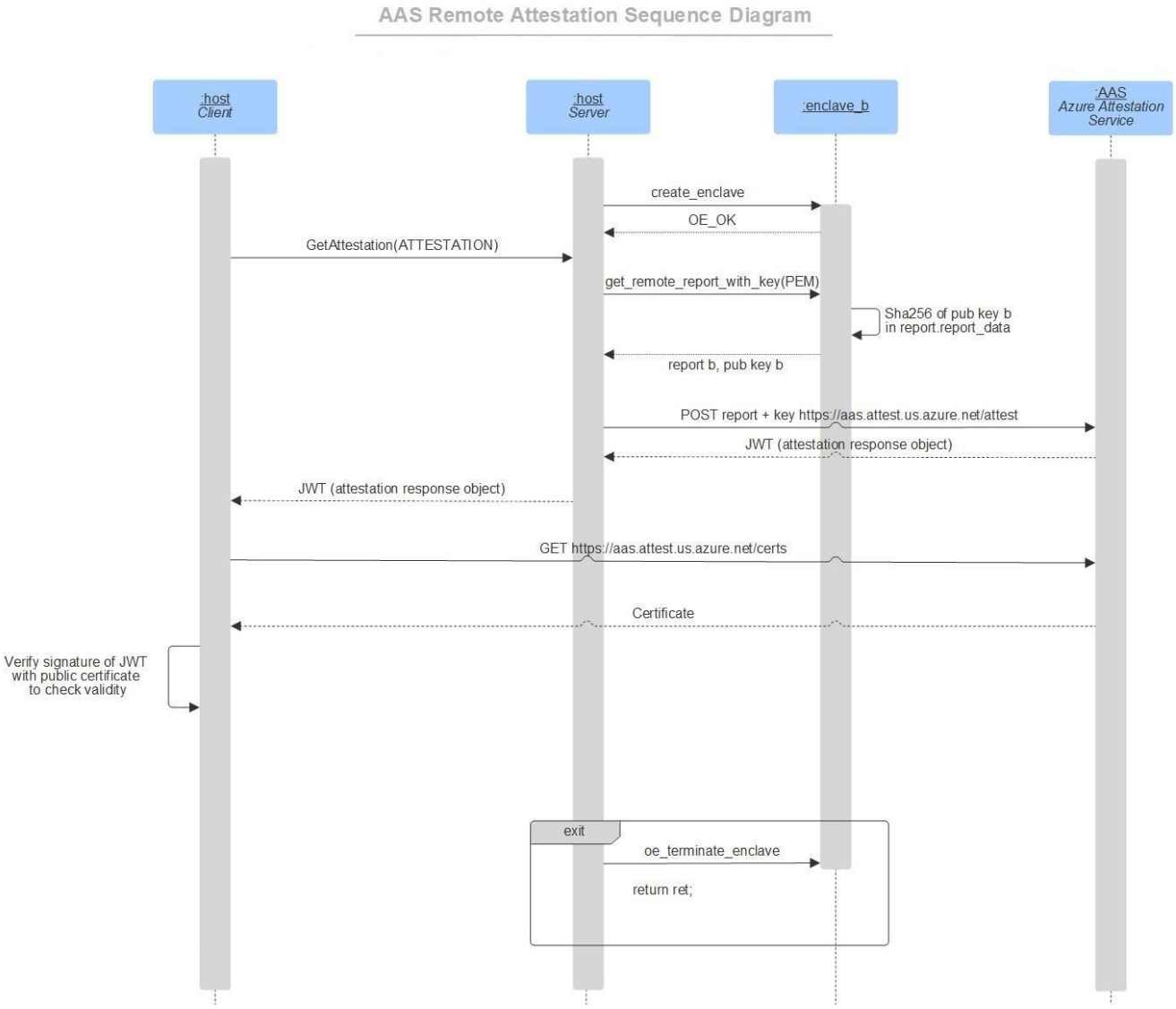


Figure 16 Microsoft Attestation Service Sequence Diagram

A sample code walkthrough for the scenario

The Server is still written in C++, but the client is written in Python.

The server host and the client do the following in this sample:

1. The server creates an enclave:

```
oe_create_secretsharinge_enclave( enclaveImagePath, OE_ENCLAVE_TYPE_SGX, OE_ENCLAVE_FLAG_DEBUG, NULL,
0, &enclave);
```

2. The server runs the gRPC service and listens for connection:

```
std::string address("0.0.0.0:5000");
SecretSharingServiceImplementation service;

ServerBuilder builder;

builder.AddListeningPort(address, grpc::InsecureServerCredentials());
builder.RegisterService(&service);

std::unique_ptr<Server> server(builder.BuildAndStart());
server->Wait();
```

3. The client connects to the gRPC Server:

```
with grpc.insecure_channel(target='localhost:5000',
                           options=[('grpc.enable_retries', 0),
                                    ('grpc.keepalive_timeout_ms', 100)
                                    ]) as channel:
stub = s_shr_grpc.SecretSharingStub(channel)
```

Where `SecretSharingStub` is the gRPC client stub.

4. The client requests a remote attestation to the server:

```
response = stub.GetAttestation(
   s_shr_pb.AttestationRequest(cmd=s_shr_pb.CommandRequest.ATTESTATION))
```

Where:

- `CommandRequest.ATTESTATION` is an enum to ask for an attestation
- `at_data` is an attestation structure of the server received through the network

5. The server gets the remote report and the public key:

```
if (get_remote_report(enclave, at_data)) {
   return Status(StatusCode::INTERNAL, "get_remote_attestation failed.");
}

/* ... init aad_info structure ... */

std::string ad_token;
aad_get_token(aad_info, ad_token);
get_remote_report(enclave, at_data);
std::string aas_token;
aas_request(at_data, ad_token, aas_token);

reply->set_ok(true);
reply->set_token(aas_token);
reply->set_msg("Remote report generated successfully.");
```

Where:

- `at_data` contains attestation data to be sent over http to the Microsoft Azure Attestation (MAA)
- `AttestationReply* reply` is the Protobuf message object for serialization of attestation data

- `at_data` will contain the AAS returned token

Implementation of `get_remote_report` calls `oe_get_report` `OE_CALL`, which generates the report and computes the hash of the public key to store in the `attestation_data_t` structure.

6. The client has received the server's attestation token and verifies its signature. This is done through the following call:

```
check_jwt_signature(response.token)
```

Where `response.token` is the server attestation token to be verified.

To do so, the client gets the publicly accessible certificate of the MAA to verify the JWT signature.

If the token's signature is validated, the attestation is now complete. The client can now use the public key to share data with the enclave.

7. The server at shutdown eventually frees the resources used, including the host memory allocated by the enclaves and the enclaves themselves. For example:

```
oe_terminate_enclave(enclave_b);
```

Now that you may have a good understanding of the code of the samples for the provided scenarios, we will see now how to build and run these samples.

Building and running the sample codes for the scenarios

To successfully build and run the above samples code, you need a computer equipped with the Intel SGX technology running Linux, for example, an Ubuntu 10.04 distro, with the Open Enclave SDK (OESDK) installed on top of it.

One of the easy way to fulfill the above requirement consists in leveraging Azure Confidential Computing[70] (ACC), and the SGX-based v1 DC-series or v2 DCsv2-series families of VMs it provides.

We assume here that you are using a Windows 10 local machine as per section § *Guide prerequisites* above.

Using a DC_series VM in Azure

Installing OpenSSH on Windows 10

The OpenSSH Client and OpenSSH Server are separately installable components in Windows 10 1809 and above.

> **Note** For information about the OpenSSH availability on Windows 10, see here[71].

To install OpenSSH on your Windows 10 local machine, perform the following steps:

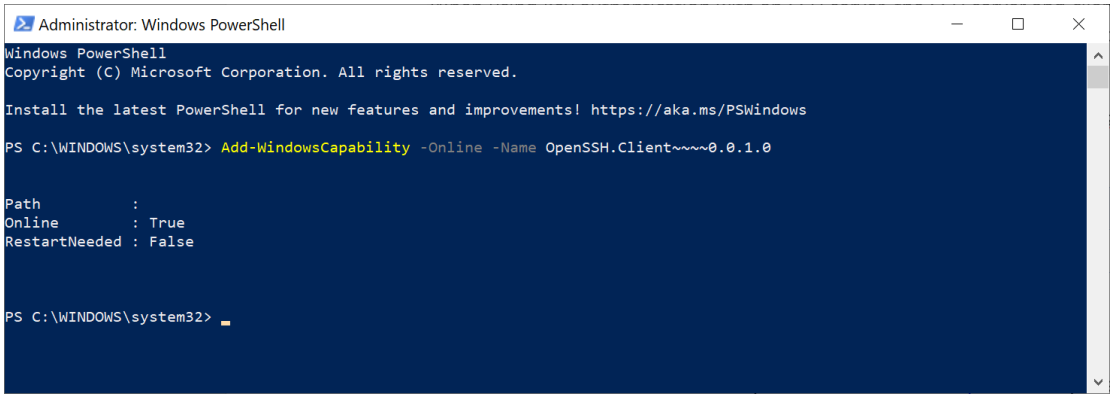
1. Open an elevated PowerShell console.

[70] Azure Confidential Computing: https://azure.microsoft.com/solutions/confidential-compute/

[71] Installation of OpenSSH For Windows Server 2019 and Windows 10: https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse

2. Run the following command:

```
PS C:\> Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```



Once the installation completes, you can use the OpenSSH client from PowerShell or the Windows 10 command shell.

Generating your RSA Key pairs with OpenSSH

OpenSSH includes different tools and more specifically the `ssh-keygen` command for generating secure RSA key pairs, that can be in turn used for key authentication with SSH.

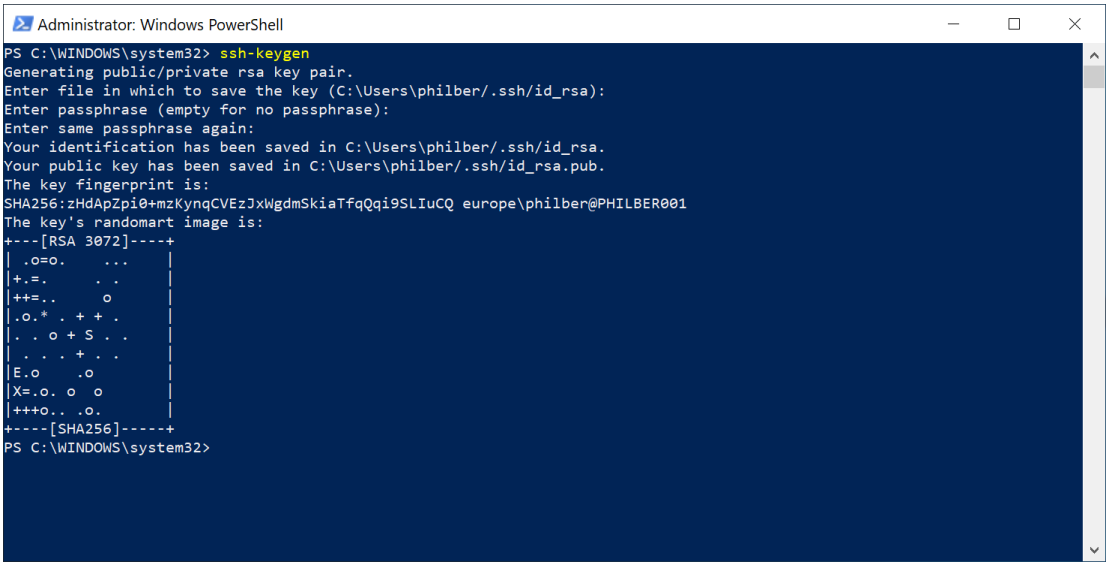
RSA Key pairs refer to the public and private key files that are used by certain authentication protocols.

To generate your RSA Key pairs, perform the following steps:

1. Open an elevated PowerShell console.
2. Run the following command:

```
PS C:\> ssh-keygen
```

You can just hit ENTER to generate them, but you can also specify your own filename if you want. At this point, you'll be prompted to use a passphrase to encrypt your private key files. The passphrase works with the key file to provide 2-factor authentication. For this example, we are leaving the passphrase empty.



Note SSH public-key authentication uses asymmetric cryptographic algorithms to generate two key files – one "private" and the other "public". The **private key** file is the equivalent of a password and should protected under all circumstances. If someone acquires your private key, they can log in as you to any SSH server you have access to. The **public key** is what is placed on the SSH server and may be shared without compromising the private key.

When using key authentication with an SSH server, the SSH server and client compare the public key for username provided against the private key. If the public key cannot be validated against the client-side private key, authentication fails.

By default, the files are saved in the following folder *%USERPROFILE%\.ssh:*

File	Description
%USERPROFILE%\.ssh\id_rsa	Contains the RSA private key
%USERPROFILE%\.ssh\id_rsa.pub	Contains the RSA public key.

Creating a DC_series VM in your Azure subscription

In the Azure Platform, the Open Enclave SDK must be indeed installed on top of a Confidential Compute DC-series[72] or DCsv2-series[73] [74] virtual machine (VM).

[72] PREVIOUS GENERATIONS OF VIRTUAL MACHINE SIZES: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-previous-gen?toc=/azure/virtual-machines/linux/toc.json&bc=/azure/virtual-machines/linux/breadcrumb/toc.json#preview-dc-series

[73] GENERAL PURPOSE VIRTUAL MACHINE SIZES: https://docs.microsoft.com/en-us/azure/virtual-machines/dcv2-series

[74] AZURE LAUNCHES DC-SERIES CONFIDENTIAL COMPUTE VM PREVIEW: https://www.petri.com/azure-launches-dc-series-confidential-compute-vm-preview

As such, both DC-series and DCsv2-series VMs are generation 2 VMs[75] that, besides the supports of the Intel SGX technology, use the new UEFI-based boot architecture rather than the BIOS-based architecture used by generation 1 VMs, along with additional features that are not available in generation 1 VMs, such as increased memory, and virtualized persistent memory (vPMEM).

V1 DC-Series VMs are in Preview and deploy an older version of the DC-Series VMs. They aren't going to be generally available and will remain in preview until deprecation.

For the most up-to-date technology and confidential computing VM, you will need to use DCsv2-series instead that correspond to an Azure Confidential Compute (Virtual Machine) V2 deployment.

The DCsv2-series is indeed a new family of CC VMs in Azure that are backed by the latest generation of the Intel XEON E-2288G processor[76] with the Intel SGX technology. With the above-mentioned Intel Turbo Boost Technology, these machines can go up to 5.0GHz. This family is currently available in UK South and Canada Central only.

DCsv2-series VMs allow for a greater selection of VM sizes, higher EPC (Enclave Page Cache), and a higher level of support.

For the sake of this guide, and to minimize the implied cost, you can opt for the Standard_DC1s_v2 with 1 vCPU and 4 GB of memory, between the two available sizes.

Furthermore, in terms of OS, amongst the three operating systems are supported for the above family of VMs, you will choose Ubuntu Server 18.04 TLS.

You can refer to the section § *Module1: Setting up a confidential computing VM in Azure* of the **guide Building and Executing Trusted Execution Environment (TEE) based application in Azure** in this series of guide to see how to instantiate such VMs.

---

[75] SUPPORT FOR GENERATION 2 VMS ON AZURE: https://docs.microsoft.com/en-us/azure/virtual-machines/linux/generation-2#creating-a-generation-2-vm

[76] Intel XEON E-2288G: https://www.intel.com/content/www/us/en/products/processors/xeon/e-processors/e-2288g.html

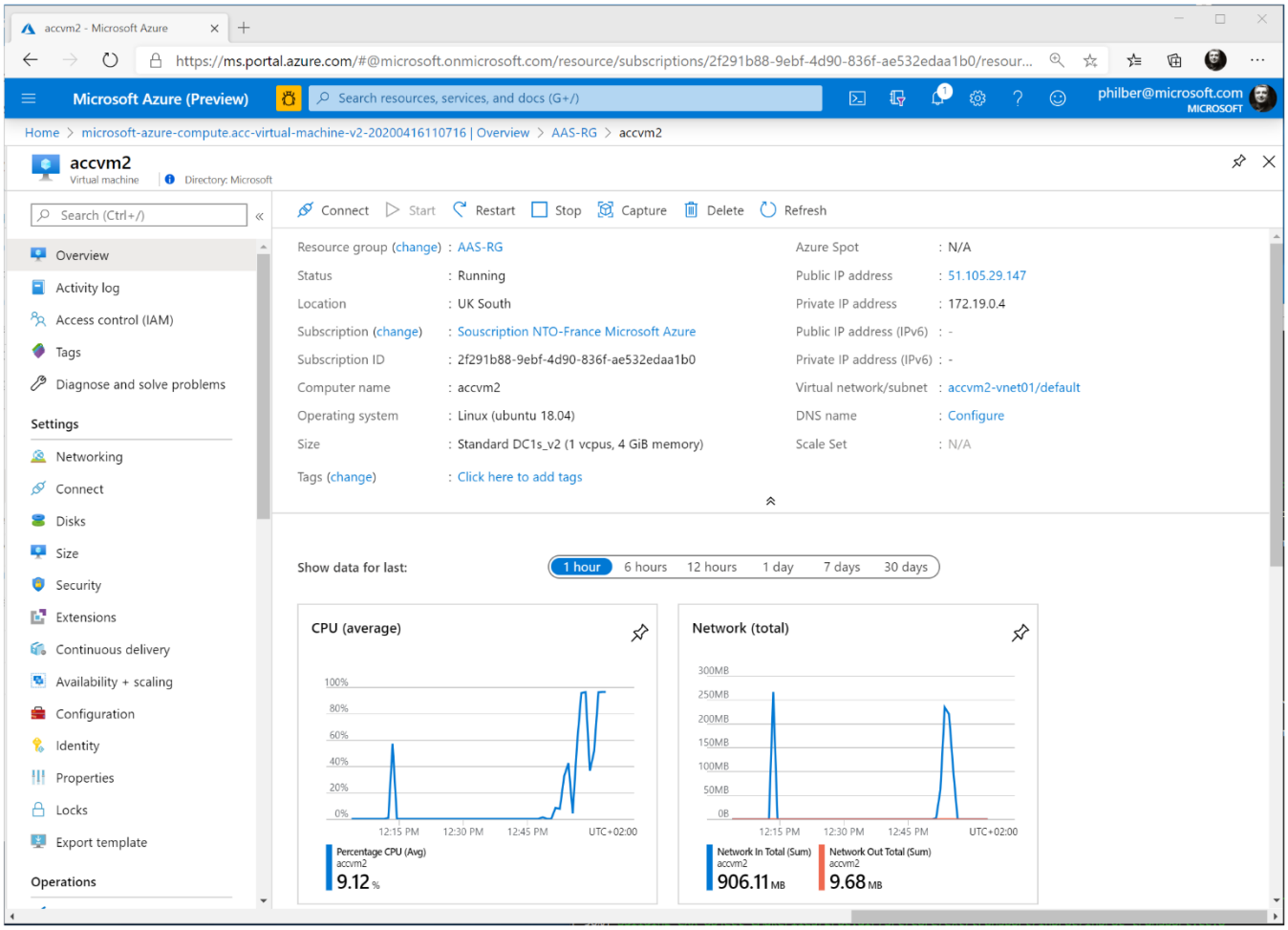
Connecting to your DC_series VM

Once your VM is online, by using a SSH client of your choice, such as OpenSSH, PuTTY, etc. you can test your remote connection to the newly created VM using the administrator credentials provided above. The public IP address of the VM can be found on the VM Networking page.

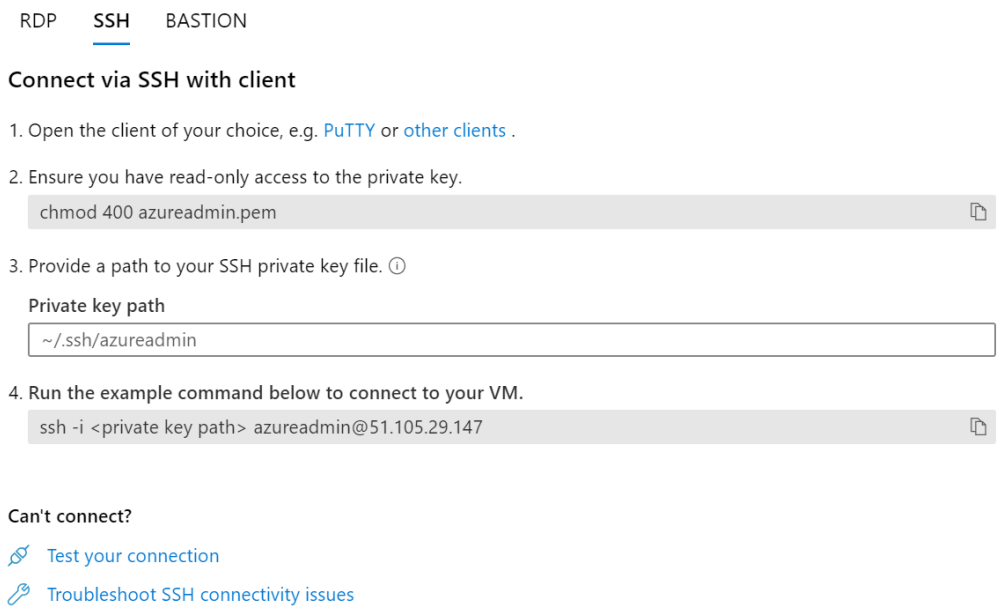
> **Note** Depending on your configuration, you may need to configure a proxy in the SSH client to connect to the virtual machine.

To connect to your VM using OpenSSH on your Windows 10 local machine, perform the following steps:

1. From the Azure portal, search for your VM and click on it to display its menu.



2. Click on **Connect**, select SSH, and then make a note of the public IP address and the SSH connection string.

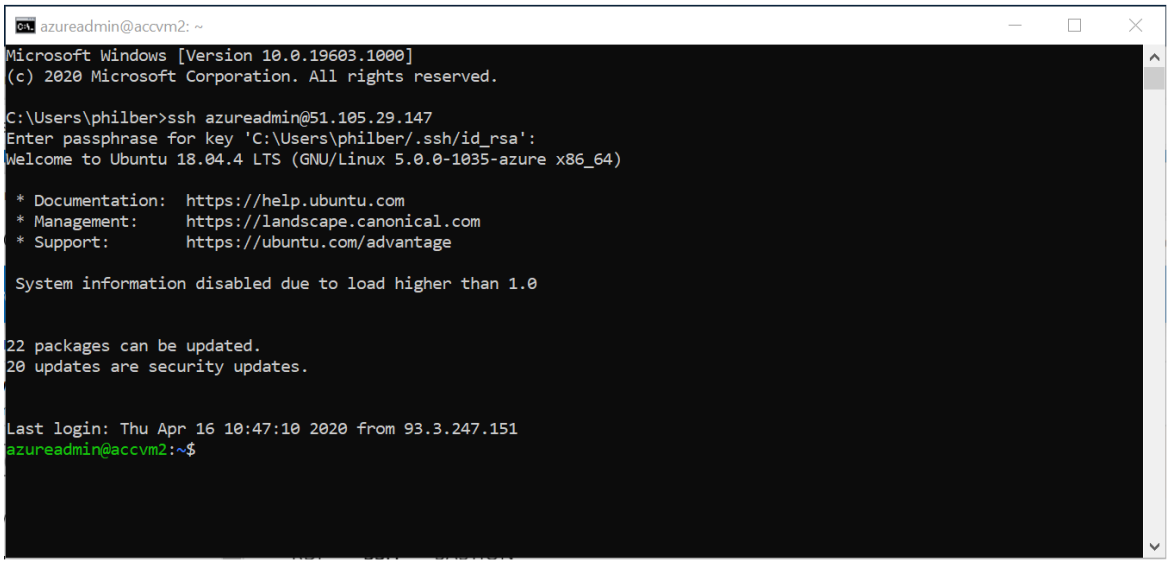


In our illustration, the IP address of the DC_series VM is 51.105.29.147, and the SSH connection string is azureadmin@51.105.29.147.

3. Launch the Windows Terminal by typing "*Win+R*", then "*Windows Terminal*", and then press ENTER. You can instead use a classic PowerShell command prompt.
4. Now SSH to your VM:

```
PS C:\> ssh azureadmin@ 51.105.29.147
```

5. When prompted, type "*yes*". Optionally specify your passphrase if any for your private key.



Et voila! You should now be connected to one of your DC-series VMs. For example, a DCsv2 series in our illustration.

Installing the Open Enclave SDK and other dependencies

Installing the Open Enclave SDK

See Install the Open Enclave SDK (Ubuntu 18.04)[77]

Perform the following steps:

1. From the Bash terminal, configure the Intel and Microsoft APT repositories:

```
$ echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu bionic main' | sudo tee
/etc/apt/sources.list.d/intel-sgx.list
$ wget -qO - https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-deb.key | sudo apt-key add -

$ echo "deb http://apt.llvm.org/bionic/ llvm-toolchain-bionic-7 main" | sudo tee
/etc/apt/sources.list.d/llvm-toolchain-bionic-7.list
$ wget -qO - https://apt.llvm.org/llvm-snapshot.gpg.key | sudo apt-key add -

$ echo "deb [arch=amd64] https://packages.microsoft.com/ubuntu/18.04/prod bionic main" | sudo tee
/etc/apt/sources.list.d/msprod.list
$ wget -qO - https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add –
```

2. Install the Intel SGX DCAP driver:

```
$ sudo apt update
$ sudo apt -y install dkms
$ wget https://download.01.org/intel-sgx/sgx-
dcap/1.5/linux/distro/ubuntuServer18.04/sgx_linux_x64_driver_1.21.bin -O sgx_linux_x64_driver.bin
$ chmod +x sgx_linux_x64_driver.bin
$ sudo ./sgx_linux_x64_driver.bin
```

Note This step also installs the az-dcap-client[78] package which is necessary for performing remote attestation in Azure. A general implementation for using Intel DCAP outside the Azure environment is coming soon.

3. Install the Intel and Open Enclave packages and dependencies:

```
$ sudo apt -y install clang-7 libssl-dev gdb libsgx-enclave-common libsgx-enclave-common-dev
libprotobuf10 libsgx-dcap-ql libsgx-dcap-ql-dev az-dcap-client open-enclave
```

Note This may not be the latest Intel SGX DCAP driver. Please check with Intel's SGX site[79] if a more recent SGX DCAP driver exists.

4. Install CMake

```
$ sudo snap install cmake --classic
$ cmake –version
```

[77] INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04): https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md

[78] az-dcap-client: https://github.com/microsoft/azure-dcap-client

[79] Intel® Software Guard Extensions SDK for Linux: https://01.org/intel-software-guard-extensions/downloads

For compiling and running the above sample code, you must have CMake and protobuf installed.

Installing gRPC

Let's start by installing gRPC. For that purpose, you can follow the guide available at https://github.com/grpc/grpc/blob/v1.27.0/BUILDING.md as illustrated hereafter. Please note that the last tested version with this code sample being 1.27.2.

Perform the following steps:

1. Fulfill the prerequisites:

```
$ apt-get install build-essential autoconf libtool pkg-config
$ apt-get install libgflags-dev libgtest-dev
$ apt-get install clang-5.0 libc++-dev
```

2. Clone gRPC repository with 1.27.2 release tag:

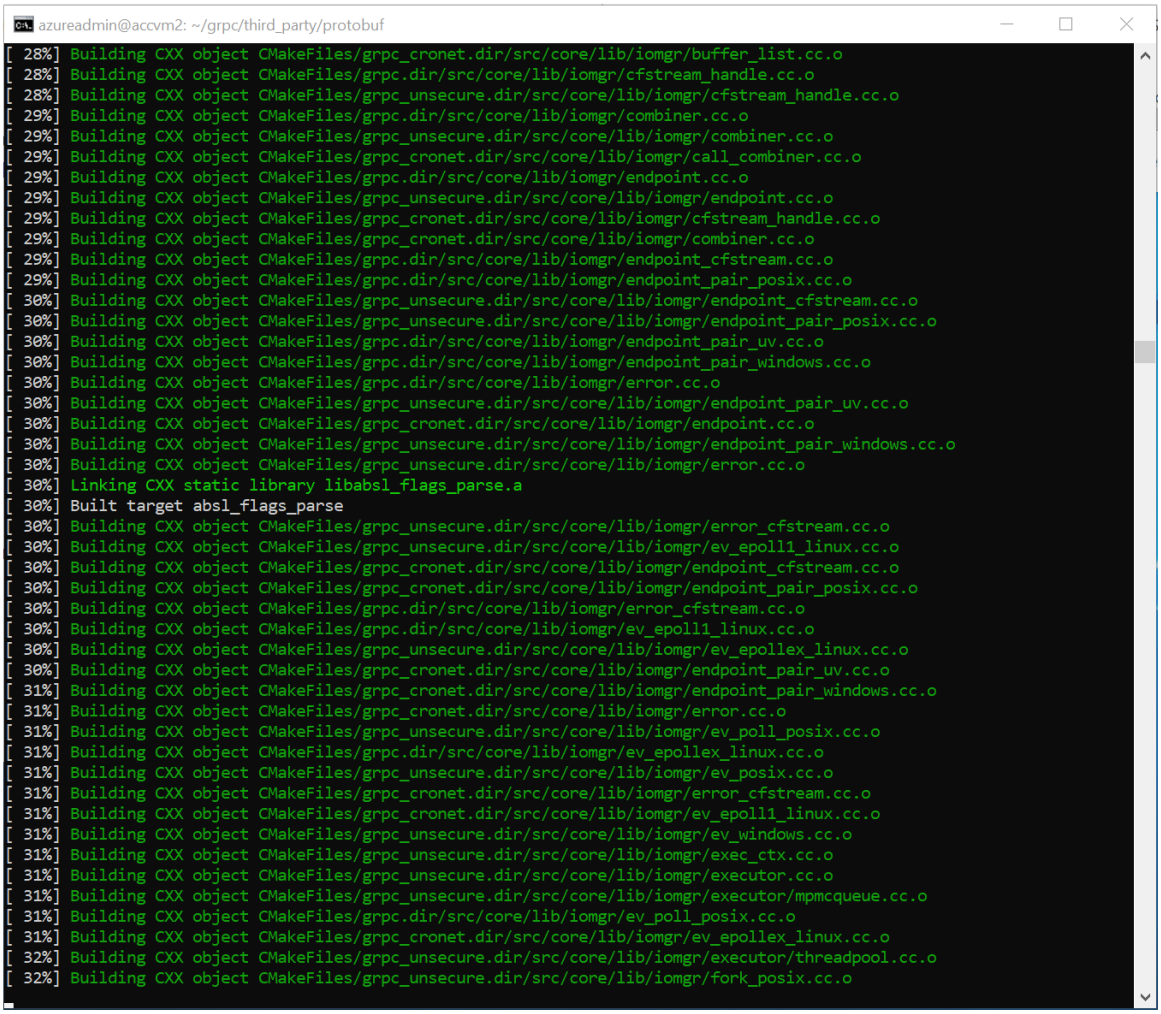
```
$ git clone -b v1.27.2 https://github.com/grpc/grpc
$ cd grpc
$ git submodule update --init
```

3. Install Protocol Buffers (Protobuf) dependency:

```
$ cd third_party/protobuf/cmake
$ mkdir build
$ cd build
$ cmake -Dprotobuf_BUILD_TESTS=OFF -DCMAKE_BUILD_TYPE=Release ..
$ sudo make -j4 install
```

4. Now install gRPC from source (reusing Protobuf previous installation):

```
$ cd ../../ && mkdir -p cmake/build$ cd cmake/build
$ cmake \
  -DCMAKE_BUILD_TYPE=Release \
  -DgRPC_INSTALL=ON \
  -DgRPC_BUILD_TESTS=OFF \
  -DgRPC_SSL_PROVIDER=package \
  -DgRPC_PROTOBUF_PROVIDER=package \
  ../..
$ sudo make -j4 install
```



Eventually, you will need also python3.7 to run the client of `ms_oe_server`.

Installing Python 3.7

Installing Python 3.7 on Ubuntu with apt is a relatively straightforward process.

Perform the following steps:

1. Start by updating the packages list and installing the prerequisites:

```
$ sudo apt update
$ sudo apt install software-properties-common
```

   The `software-properties-common` package gives you better control over your package manager by letting you add PPA (Personal Package Archive) repositories.

2. Next, add the `deadsnakes` PPA with newer releases than the default Ubuntu repositories to your sources list:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

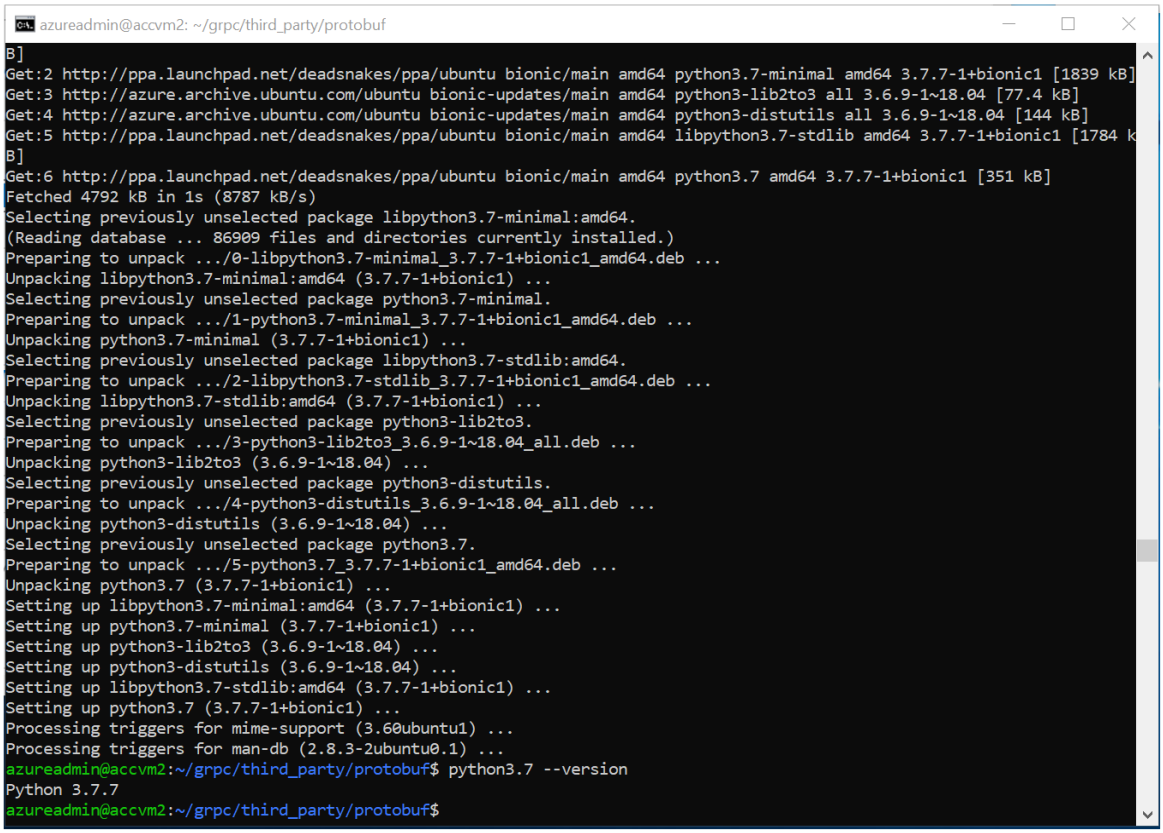
When prompted, press ENTER to continue.

3. Once the repository is enabled, install Python 3.7:

```
$ sudo apt install python3.7
```

4. Allow the process to complete and verify the Python version was installed successfully

```
$ python3.7 --version
Python 3.7.7
```



At this point, Python 3.7 is installed on your DC_series VM and is ready to be used.

Installing Azure CLI

See Install the Azure CLI[80], Install Azure CLI with apt[81]

The Azure CLI is a command-line tool providing a great experience for managing Azure resources. The CLI is designed to make scripting easy, query data, support long-running operations, and more. As of this writing, the current version of the Azure CLI is **2.3.1**.

[80] Install the Azure CLI: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest

[81] Install Azure CLI with apt: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli-apt?view=azure-cli-latest

Note For information about the latest release, see the release notes[82].

For Ubuntu 18.04, there are two ways to install the Azure CLI with distributions that support apt: As an all-in-one script that runs the install commands for you, and instructions that you can run as a step-by-step process on your own as per the above link.

You can proceed with the all-in-one script which runs all installation commands in one step. Run it by using `curl` and pipe directly to `bash`, or download the script to a file and inspect it before running:

```
$ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

To sign in on your DC_series VM, perform the following steps:

1. Run the `login` command.

```
$ az login
```

   Open a browser page at https://aka.ms/devicelogin and enter the authorization code displayed in your PowerShell terminal.

2. Sign in with your account credentials in the browser.
3. If you have multiple Azure Subscriptions, select the default Azure Subscription you want to work with

```
$ az account set --subscription <your_SubscriptionId>
```

Cloning the samples' code repo

Now clone the sample codes' repo:

```
$ cd ~/
$ git clone https://github.com/microsoft/azure-tee-attestation-samples/
```

Building the samples' code

Every project uses CMake to build the samples' code. It is recommended as a good practice to create a local "build" directory and run the commands from there:

```
$ cd azure-tee-attestation-samples
```

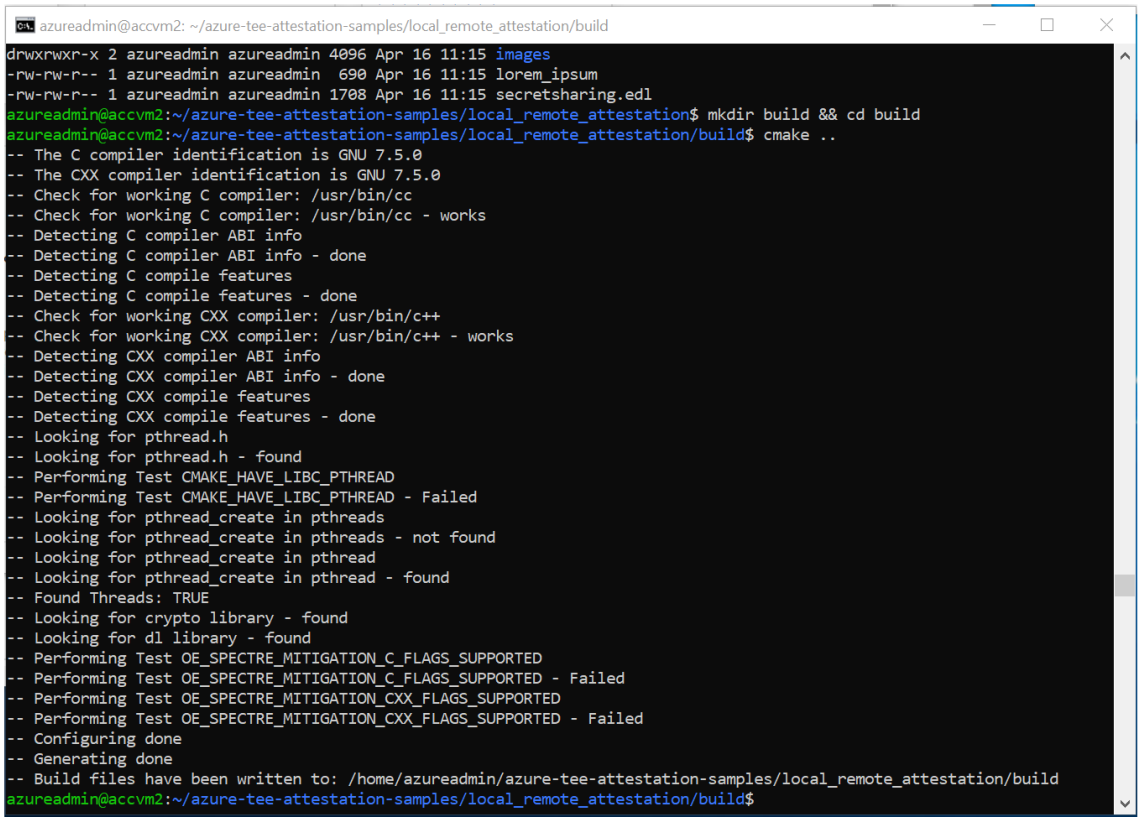
For all samples, you can build project by:

```
$ cd <your_sampleCodeDirName>
```

[82] Azure CLI release notes: https://docs.microsoft.com/en-us/cli/azure/release-notes-azure-cli?view=azure-cli-latest

```
$ mkdir build && cd build
$ cmake ..
```

Where <*your_sampleCodeDirName*> is either `local_remote_attestation`, `remote_client_server`, one_enclave.



`local_remote_attestation` has only one executable to run. Once the cmake completes, simply run this command:

```
$ make run
```

This will run the host executable with the enclave path and every other argument needed, i.e. file data that is encrypted/decrypted.

`remote_client_server` is a client server sample. Therefore, once the cmake completes, you will need to run the following commands:

```
$ make run_server
```

and

```
$ make run_client
```

Those will run the host executables with their enclave path (`enclave_for` client and `enclave_b` for server) and every argument needed, i.e. the (file to encrypt for client).

`one_enclave` is also (yet) another client server sample code. However, the server is built with CMake (since it's written in C++) and the client uses Python that just needs to be run. Therefore, run the server with the following command:

```
$ make run_server
```

And, in turn, the client with this one:

```
$ python3 client.py*
```

> **Note** You might want to use virtualenv or similar to install the clients dependencies (see directory's README.md). Also, in this sample code, we do not show encryption or decryption of data in the enclave, therefore we do not provide one in arguments.

Further illustrating the scenario 3

Configuring an Azure AD identity for the Remote Attestation with local attestation sample code

See Create an Azure service principal with Azure PowerShell[83] and How to: Use the portal to create an Azure AD application and service principal that can access resources[84].

During an attestation workflow, your application will send attestation requests to your attestation provider. For that, it must obtain an authentication token from Azure AD, which requires an identity in Azure AD if your attestation provider is in the AAD trust model.

The following instructions assume you have an active PowerShell session, you have signed into Azure, and you are connected to a subscription containing your attestation provider as per section § *Registering the required Microsoft.Attestation resource provider* above.

From your Windows 10 local machine, perform the following steps:

1. Create a service principal representing a new application in Azure AD with the role **Attestation Reader** on the resource group in which the attestation provider has been created. For example, **AzureAttestationTest** in our illustration.

```
PS C:\> $servicePrincipalName = "<your_ServicePrincipalName>"
PS C:\> $sp = New-AzADServicePrincipal  -DisplayName $servicePrincipalName -Role "Attestation Reader"
-Scope /subscriptions/<your_Subscription_ID>/resourceGroups/<your_ResourceGroupName>
```

[83] Create an Azure service principal with Azure PowerShell: https://docs.microsoft.com/en-us/powershell/azure/create-azure-service-principal-azureps?view=azps-4.4.0

[84] How to: Use the portal to create an Azure AD application and service principal that can access resources: https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-service-principal-portal

- Replace <*your_ServicePrincipalName*> with the name you want to use for your new service principal. For example, **MAAClientApp** in our illustration.
- Replace <*your_SubscriptionId*> with the name you want to use for your new attestation provider.
- Replace **<***your_ResourceGroupName***>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.

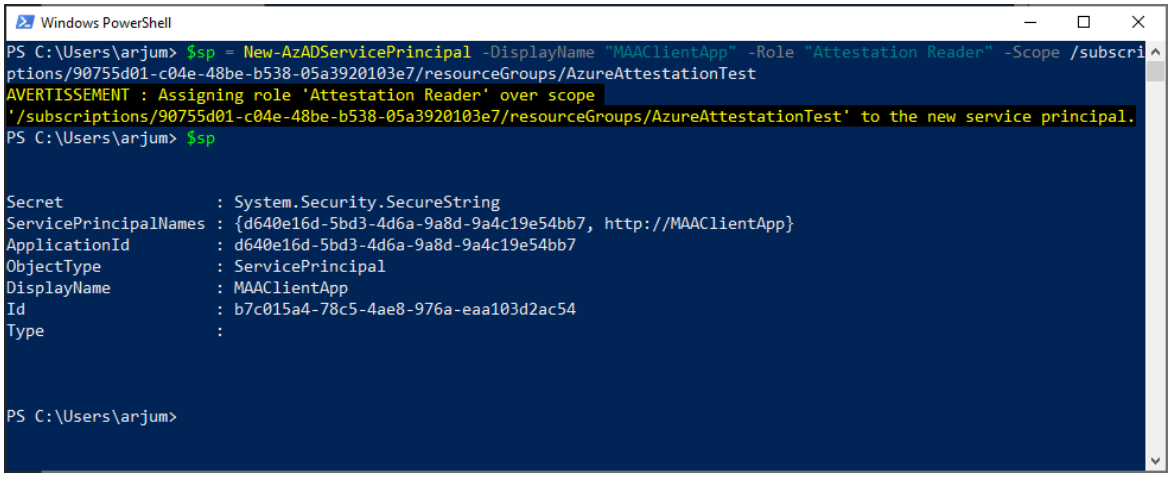
  For example, in our illustration:

```
PS C:\> $servicePrincipalName = "MAAClientApp"
PS C:\> $sp = New-AzADServicePrincipal  -DisplayName $servicePrincipalName -Role "Attestation Reader"
-Scope /subscriptions/90755D01-c04e-b538-05a3920193e7/resourceGroups/AzureAttestationTest
AVERTISSEMENT ! Assigning role 'Attestation Reader'over scope
'./subscriptions/90755d01-c04e-48be-b538-05a3920103e7/resourceGroups/AzureAttestationTest' to the new
service principal.
```

  The above command creates a service principal in Azure AD. Since the application id was not provided, an application was created for the service principal. The service principal was created with **Attestation Reader** permissions over the current subscription.

2. Display the returned object by simply typing:

```
PS C:\> $sp
```



  The object returned from `New-AzADServicePrincipal` contains the **Id** and **DisplayName** properties, either of which can be used for signing in with the service principal.

Note The returned object also contains the Secret (credential) which is a SecureString containing a generated password. Make sure that you store this value somewhere secure to authenticate with the service principal. Its value won't be displayed in the console output. If you lose the password, you can reset the service principal credential.

3. Output the **ApplicationId** of the service principal. Take note of the Application Id, as you will need it in a later step.

```
PS C:\> Write-Host "Application Id:" $sp.ApplicationId.Guid
Application Id: d640e16d-5bd3-4d6a-9a8d-9a4c19e54bb7
```

4. Export the secret of the newly created service principal and save it for a later use:

```
PS C:\> $BSTR = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($sp.Secret)
PS C:\> $UnsecureSecret = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($BSTR)
PS C:\> Write-Host "Secret:" $UnsecureSecret
Secret: fec76…-…-…-…30c877
```

   Note that by default, the secret will expire in two years.

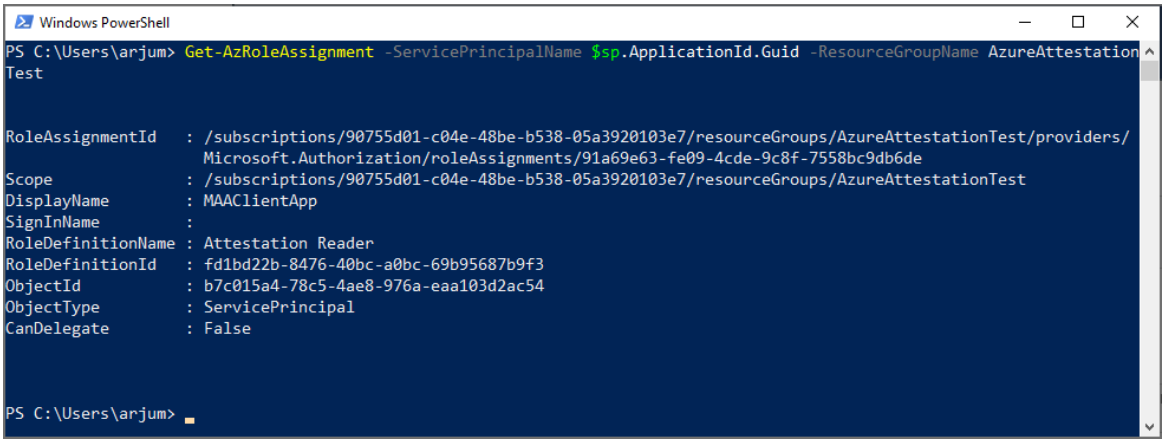
5. Validate that the service principal is assigned to the **Attestation Reader** role.

```
PS C:\> Get-AzRoleAssignment -ServicePrincipalName $sp.ApplicationId.Guid -ResourceGroupName
<your_ResourceGroupName>
```

- Replace **<*your_ResourceGroupName*>** with the name you have chosen before for the previous resource group. For example, **AzureAttestationTest** in our illustration.

  For example, in our illustration:

```
PS C:\> Get-AzRoleAssignment -ServicePrincipalName $sp.ApplicationId.Guid -ResourceGroupName
AzureAttestationTest
```

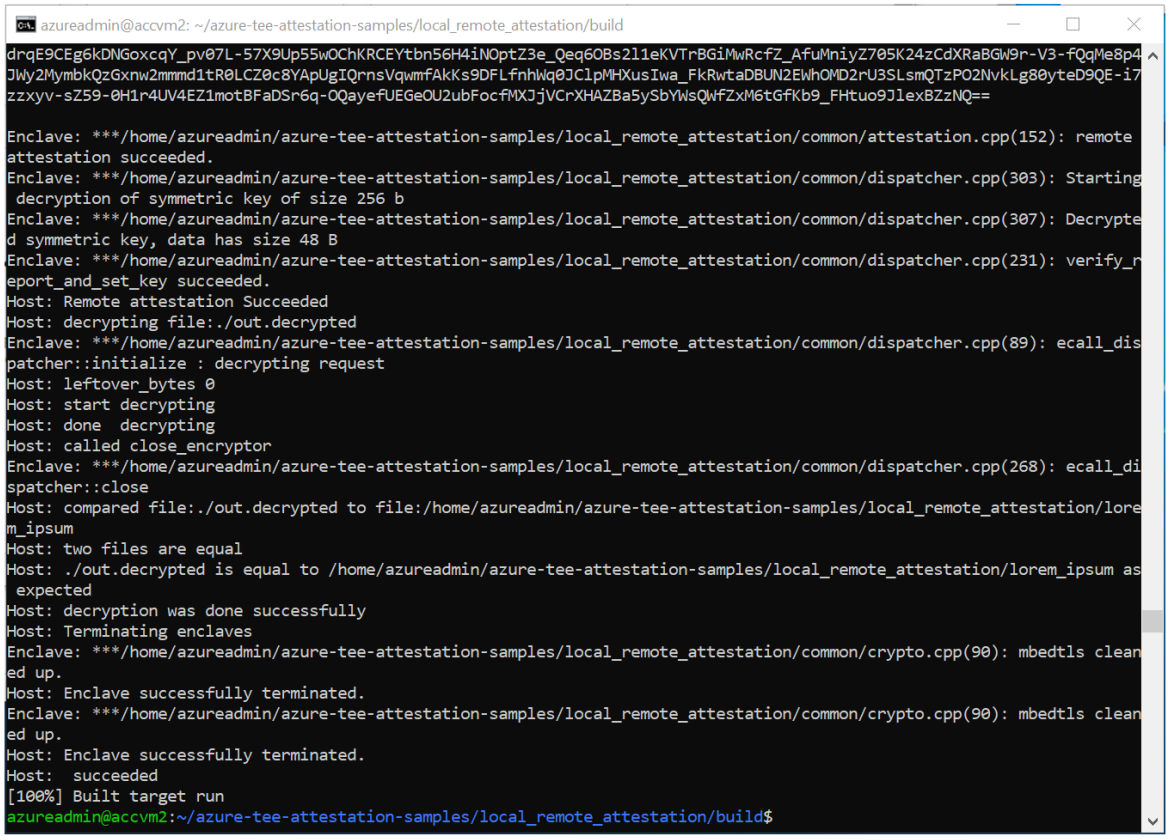


Compiling and running the Remote Attestation with local attestation sample code

SSH to your DC_series VM, from the Bash terminal, log to your Azure subscription as per section § *Installing Azure CLI* above.

Now run the sample by typing:

```
$ cd ~/
$ cd azure-tee-attestation-samples/local_remote_attestation/build
$ make run
```



Getting the enclave A Quote and related Enclave Held Data (EHD)

In general, the overall flow of an SGX (or here Open Enclave) secure key exchange attestation operation is:

1. The "relying party", here the enclave B, would like to communicate with the enclave A. The "relying party" requests a key from the enclave A.
2. The enclave A creates a "Quote", which expresses its state. For the "Enclave Held Data" (EHD), it uses the public key of enclave A (typically the public key is formatted as a RFC 7517 Json Web Key (JWK)[85]). It places the SHA256 of the JWK in the first 32 bytes of the `reportData` field of the quote and returns both the quote and the EHD to the "relying party", i.e. enclave B.
3. The "relying party" sends the quote and EHD to the attestation provider, which validates the token and EHD and returns a signed JWT.
4. The "relying party" validates the resulting JWT and extracts the EHD from the JWT.

The "relying party" then uses the public key contained in the `aas-ehd` field to encrypt the data to be sent to the enclave. The "relying party" can be certain that the key was in fact a key known to the enclave because the attestation service verified that the enclave was valid and the enclave held data was known to the application.

Let's see how the above translate in our illustration with scenario 3.

[85] RFC 7517 JSON Web Key (JWK): https://tools.ietf.org/html/rfc7517

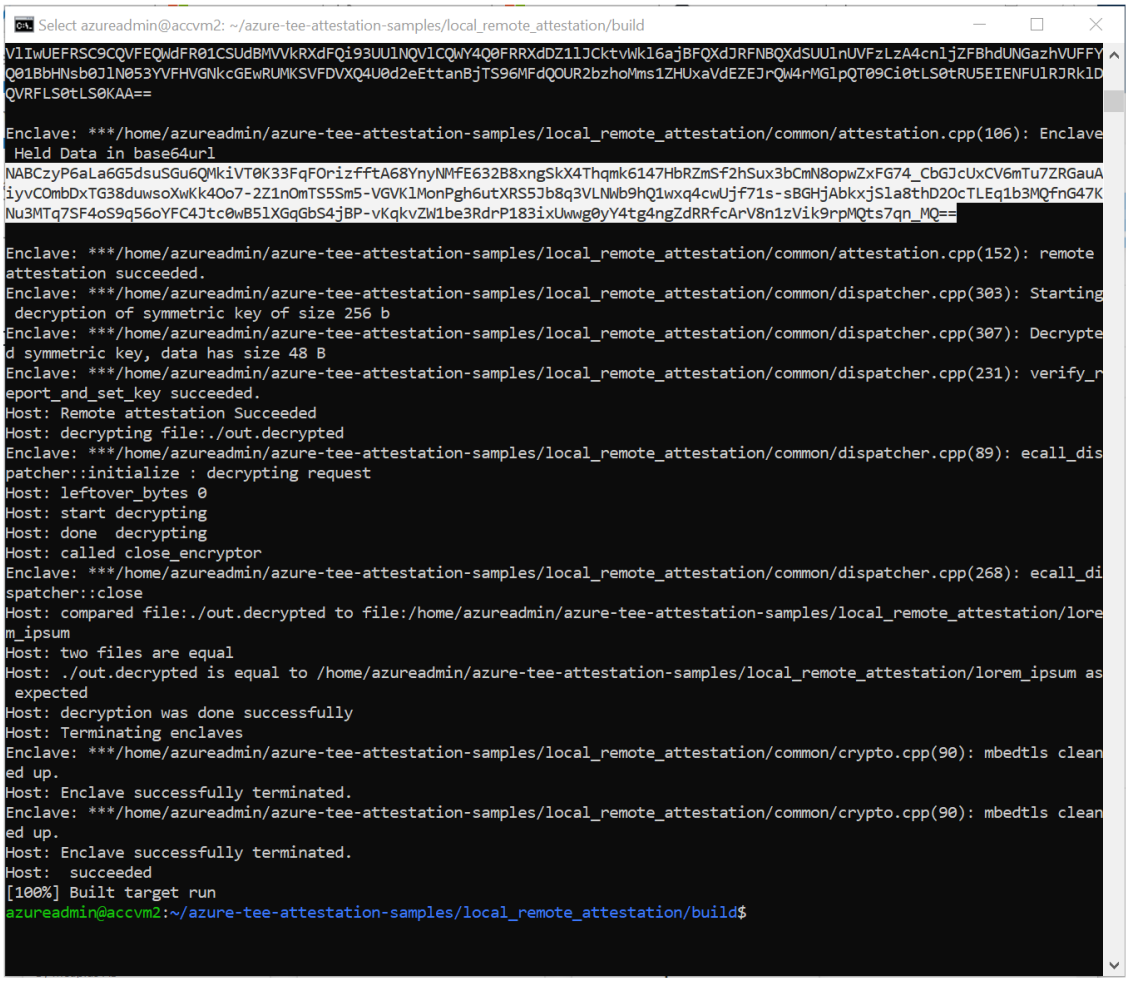
First browse to the following text **"Host: Requesting enclave b to attest enclave a's remote report and set the key type 1====="** and copy the Open Enclave Quote.

For example:

AQAAAAIAAADoEQAAAAAAAAMAAgAAAAAABQAKAJOacjP3nEyplAoNs5V_Bgd8wpTjoGCPzVq9_3MQq_ZOAAAAAAA8PAwX_gAYAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcAAAAAAAAABwAAAAAAAAAD4sflDstlXJ8KsIMqsBwRfOXtwzZtbCt
n_wYQBDM6JRwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA_kO167PNgc9kXnWzf40ymZNeC_nFTP1YAKH5dnYfKcsAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAABAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAS4-t1lV1bJSVkwGL9y5aUm3V-
oAx8EMk9x3AiS1e4JoAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADQQAACxy23AQsZaHyqoKVUJx8ett-
bfmpRi4d8vYcKrOX2xOXs8Y9m5dbGMkWwdR57F1AxrIrJgujgu421SbdIEeABLyJEeqikxDNZ8n2JdYjVXAVplc9ppU0-
zO9W6wH_Z5RNNmaDZU16UDUpUlql5_9jxm9V8jp-ysPeE8-
CtJsEW2Q8PAwX_gAYAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABUAAAAAAAAAABwAAAAAAAAAADNyt-
32yKtpf1gNFXN4b-
folj6XyhNzW4MYzkvYzoRBQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAjE9XddeWUD6WE393xoqCmgBWrI3tcBQLCBsJ
RJDFe_8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABAAUAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAcbEIvJ-
McbyhmWVRjtqFEG1iDl6RQ99DW5SAP64np5QAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGDIPuaNHTf2ATLTxd4EqJ-
knZKp371eKp3S0E6m84_7wfeMIPUGQCJhKPydRKv6iiE0iZUEBva1mq-
LwXBB2VogAAABAgMEBQYHCAkKCwwNDg8QERITFBUWFxgZGhscHR4fBQDMDQAALS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1J
SUVnRENDQkNlZ0F3SUJBZ0lWQUtic2FuMWpKY0xlTEprM0RZUi83Y05yOTd1Z01Bb0dDQ3FHU000OUJBTUMKTUhFeEl6QWhCZ05WQ
kFNTUdrbHVkR1ZzSUZOSFdDQlFRMHNnVUhKdlkyVnpjMjl5SUVOQk1Sb3dHQVlEVlFRSwpEQkZKYm5SbGJDQkRiM0p3YjNKaGRHbH
ZiakVVTUJJR0ExVUVCd3dMVTJGdWRHRWdRMnhoY21FeEN6QUpCZ05WCkJBZ01Ba05CTVFzd0NRWURWUVFHRXdKVlV6QWVGdzB5TUR
BME1UVXhPRFU1TlRGYUZ3MHlOekEwTVRVeE9EVTUKTlRGYU1IQXhJakFnQmdOVkJBTU1HVWx1ZEdWc0lGTkhXQ0JRUTBzZ1EyVnlk
R2xtYVdOaGRHVXhHakFZQmdOVgpCQW9NRVVsdWRHVnNJRU52Y25CdmNtRjBhVzl1TVJRd0VnWURWUVFIREF0VFlXNTBZU0JEYkdGe
VlURUxNQWtHCkExVUVDQXdDUTBFeEN6QUpCZ05WQkFZVEFsVlRNRmt3RXdZSEtvWkl6ajBDQVFZSUtvWkl6ajBEQVFjRFFnQUUKbl
RvVktkeGc3Q2ZpOWd0RkJQZHdiY0lERjh3UWltZmIxTVAyOCs4empPNTlIUFVueVd2MXZoRlRNRkxzTnRSSwpFcnFCTzZyYTkRPdWJ
ISTlxdWhmYTRLT0NBcHN3Z2dLWE1COEdBMVVkSXdRWU1CYUFGTkRvcXRwMTEva3VTUmVZZClBIc1VaZERWOGxsTk1GOEdBMVVkSHdS
WU1GWXdWS0JTb0ZDR1RtaDBkSEJ6T2k4dllYQnBMblJ5ZFhOMFpXUnUKWlhKMmFXTmxjeTVwYm5SbGJDNWpiMjB2YzJkNEwyTmxjb
lJwWm1sallYUnBiMjR2ZGpFdmNHTnJZM0pzUDJOaApQWEJ5YjJObGMzTnZjaklQZGVvhRNEVGZ1FVMi9Sbmo5cW1rSkZLVW1DM1
QvTGRxdWoyTVpFd0RnWURWUjBQQQgFRSC9CQVFEQWdiQU1Bd0dBMVVkRXdFQi93UUNNNQUF3Z2dIVUJna3Foa2lHK0UwUwQkRRRUVnZ0h
GTUlJQndUQWVCZ29xaGtpRytFME5BUUVFRUFkQkJCSTQyQ1BpaWowbWRvbUw0cUJybmRKKzhnNOU1JSUJaQVlLS29aSWh2aE5BUUJCQ
VlF3RUFZTEtvWklodmhOQVEwQkFnRUNBUUV3RUFZTEtvWklodmhOQVEwQkFnSUNBUUV3RUFZTEtvWklodmhOCkFRMEJBZ01DQVFJd
0VBWUxLb1pJaHZoTkFRMEJBZ1FDQVFRd0VBWUxLb1pJaHZoTkFRMEJBZ1VDQVFFd0VRWUwKS29aSWh2aE5BUTBCQWdZQ0FnRnQ0FNQk
FHQ3lxR1NJYjRUUUVOQVFJSEFnRUdNQkFHQ3lxR1NJYjRUUUVOQVFJSQpBZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSUpBZ0VBTUJBR0N5
cUdTSWI0VFFFTkFRSUtBZ0VBTUJBR0N5cUdTSWI0ClRRRU5BUUlMQWdFQU1CQUdDeXFHU0liNFFFTkFRSU5BZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSU5BZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSU5BZ0VBTUJBR0N5cUdTSWI0
U0liNFFFTkFRSU5BZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSUtBZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSUxBZ0VBTUJBR0N5cUdTSWI0VFFFTkFRSUxBZ0VBTUJBR0N5cUdTSWI0

```
URBZkJnTlZIU01FR0RBV2dCUWlaUXpXV3AwMGlmT0R0SlZTdjFBYk9TY0dyREJTQmdOVkhSOEVTekJKCk1FZWdSYUJEaGtGb2RIUn
djem92TDJObGNuUnBabWxqWVhSbGN5NTBjblZ6ZEdWa2MyVnlkbWxqWlhNdWFXNTAKWld3dVkyOXRMMGx1ZEdWc1UwZFlVbTl2ZEV
OQkxtTnliREFkQmdOVkhRNEVGZ1FVSW1VTTFscWROSW56ZzdTVgpVcjlRR3prbkJxd3dEZ1lEVlIwUEFRSC9CQVFEQWdFR01CSUdB
MVVkRXdFQi93UUlNQVlCQWY4Q0FRRXdDZ1lJCktvWkl6ajBFQXdJRFNBQXdSUUlnUVFzLzA4cnljZFBhdUNGazhVUFFYQ01BbHNsb
0JlN053YVFHVGNkcGEwRUMKSVFDVXQ4U0d2eEttanBjTS96MFdQOUR2bzhoMms1ZHUxaVdEZEJrQW4rMGlpQT09Ci0tLS0tRU5EIE
NFUlRJRklDQVRFLS0tLS0KAA==
```

Then browse to the following text **Enclave: ***/home/azureadmin/azure-tee-attestation-samples/local_remote_attestation/common/attestation.cpp(106): Enclave Held Data in base64url"** and copy the Enclave Held Data (EHD) value, i.e. the enclave a's public key.



For example:

```
NABCzyP6aLa6G5dsuSGu6QMkiVT0K33FqFOrizfftA68YnyNMfE632B8xngSkX4Thqmk6147HbRZmSf2hSux3bCmN8opwZxFG74_C
bGJcUxCV6mTu7ZRGauAiyvCOmbDxTG38duwsoXwKk4Oo7-2Z1nOmTS5Sm5-
VGVKlMonPgh6utXRS5Jb8q3VLNWb9hQ1wxq4cwUjf71s-
sBGHjAbkxjSla8thD2OcTLEq1b3MQfnG47KNu3MTq7SF4oS9q56oYFC4Jtc0wB5lXGqGbS4jBP-
vKqkvZW1be3RdrP183ixUwwg0yY4tg4ngZdRRfcArV8n1zVik9rpMQts7qn_MQ==
```

Interacting with your attestation provider

Remain that Microsoft Azure Attestation (MAA) aims at certifying trusted execution environments (TEEs) (see section § *Using attestations with Microsoft Azure Attestation* above), here an SGX enclave with the Open Enclave SDK.

As illustrated above, your attestation provider accepts as input set of evidence, evaluates that evidence based on a policy and emits a set of claims in the form of a JWT token. The JWT token is signed so that a relying party can validate that the token is originated from a trusted MAA instance, here your own attestation provider.

Based on the previously outlined workflow (see previous section), the general steps 3 and 4 are conducted as follows:

1. The `az rest` command sends enclave evidence to MAA. Since you are using the Open Enclave URI, you are sending an Open Enclave Quote and an Enclave Held Data (EHD) blob.
2. Your attestation provider validates the Quote against applicable polices and issues a JWT token. The JWT token carry the public key of the client enclave in the field EHD (aas-ehd), see below.
3. Then, the relying party can validate the signature of the JWT token and then encrypt secrets/data with the public key of the client enclave and send it back to the enclave. (Not implemented in that sample code).

To interact with your attestation provider, perform the following steps:

1. First, open a command prompt.
2. Display your tenant ID, and make a note of if (`tenantID`):

```
$ az account show
```

3. Run the `login` command with the service principal you created earlier, see section § *Configuring an Azure AD identity for the* .

```
$ az login --service-principal -u <your_ApplicationId> -p <secret> --tenant <your_TenantId>
```

- Replace <*your_ApplicationId*> with the GUID of your application, see section § *Configuring an Azure AD identity for the* .
- Replace <*secret*> with the credential of your application, see section § *Configuring an Azure AD identity for the* .
- Replace <*your_TenantId*> with the GUID of your Azure AD tenant, you have retrieved during the previous step.

For example, in our illustration:

```
$ az login --service-principal -u d640e16d-5bd3-4d6a-9a8d-9a4c19e54bb7 -p "fec76…-…-…-…30c877" --
tenant 72f988bf-86f1-41af-91ab-2d7cd011db47
```

4. Send a request for an attestation token to your attestation provider. As such, an attestation token request is a serialized JSON object sent by client application to attestation provider with two properties:
    a. "Quote". A string containing a Base64Url[86] encoded representation of the attestation quote.
    b. "EnclaveHeldData". A string containing a Base64Url encoded representation of the Enclave Held Data.

```
$ az rest --method post --uri "<attestUri>:443/attest/Tee/OpenEnclave?api-version=2018-09-01-preview"
--body "{\"Quote\": \"<QUOTE>\", \"EnclaveHeldData\": \"<EHD>\"}" --resource https://attest.azure.net
```

- Replace ***<attestUri>*** with the base URL of your attestation provider, see section § *Creating a new attestation provider*.
- Replace ***<QUOTE>*** with the content of the Open Enclave Quote, see section *§* ***Error! Reference source not found.***.
- Replace ***<EHD>*** with the content of the Enclave Held Data, see section § ***Error! Reference source not found.***.

Your attestation provider will validate the provided "Quote" and will then ensure that the SHA256 of the provided Enclave Held Data is expressed in the first 32 bytes of the `reportData` field in the quote.

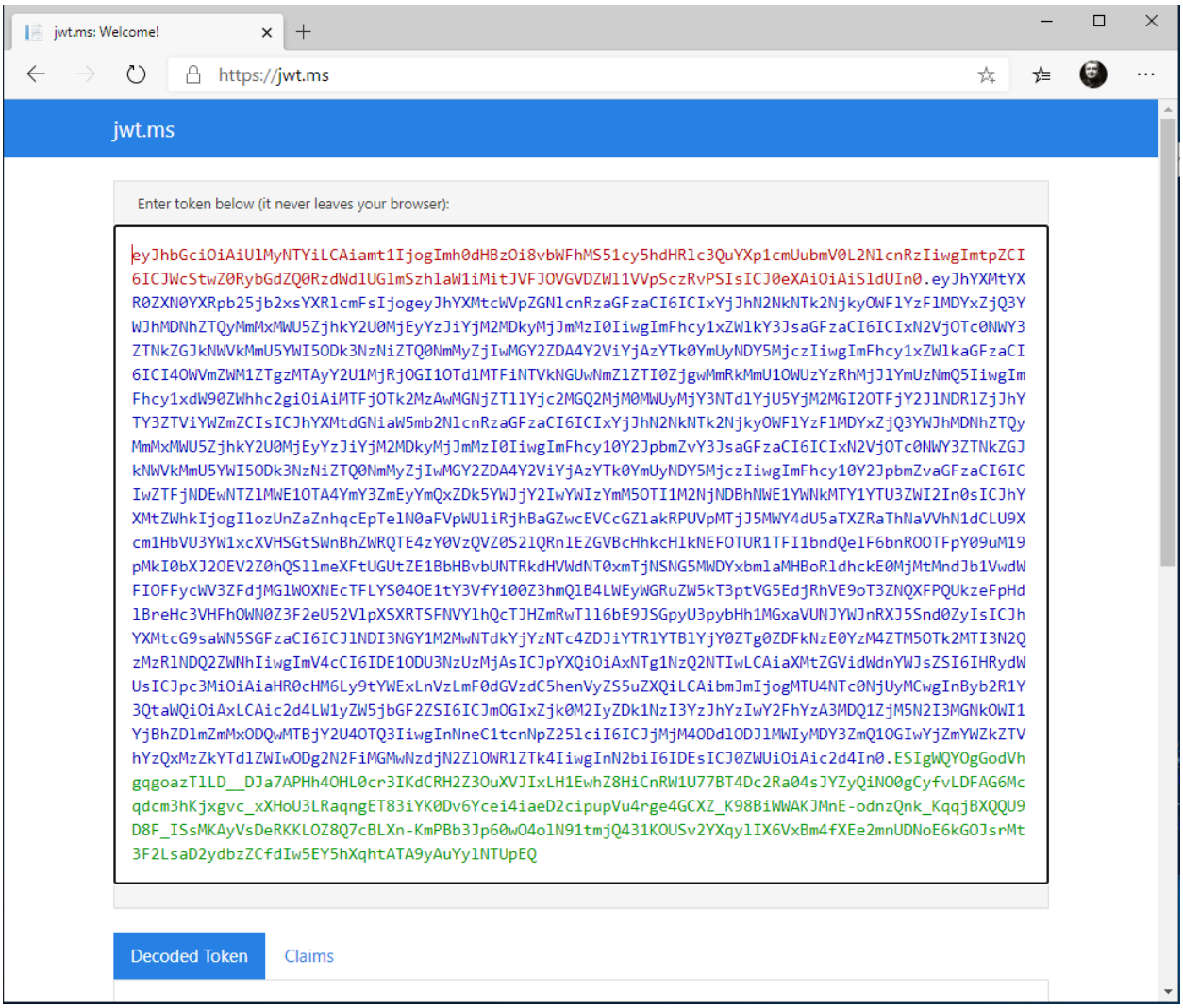
On a successful HTTP response (200), the body of the response will be a JSON string whose value contains the JWT token.

The above command should produce an output like the one below:

```
eyJhbGciOiAiUlMyNTYiLCAiamt1IjogImh0dHBzOi8vbWFhMS51cy5hdHRlc3QuYXp1cmUubmV0L2NlcnRzIiwgImtpZCI6ICJWc
StwZ0RybGdZQ0RzdWdlUGlmSzhlaW1iMitJVFJOVGVDZWl1VVpSczRvPSIsICJ0eXAiOiAiSldUIn0.eyJhYXMtYXR0ZXN0YXRpb2
5jb2xsYXRlcmFsIjogeyJhYXMtcWVpZGNlcnRzaGFzaCI6ICIxYjJhN2NkNTk2NjkyOWFlYzFlMDYxZjQ3YWJhMDNhZTQyMmMxMWU
5ZjhkY2U0MjEyYzJiYjM2MDkyMjJmMzI0IiwgImFhcy1xZWlkY3JsaGFzaCI6ICIxN2VjOTc0NWY3ZTNkZGJkNWVkMmU5YWI5ODk3
NzNiZTQ0NmMyZjIwMGY2ZDA4Y2ViYjAzYTk0YmUyNDY5MjczIiwgImFhcy1xZWlkaGFzaCI6ICI4OWVmZWM1ZTgzMTAyY2U1MjRjO
GI1OTdlMTFiNTVkNGUwNmZlZTI0ZjgwMmRkMmU1OWUzYzRhMjJlYmUzNmQ5IiwgImFhcy1xdW90ZWhhc2giOiAiMTFjOTk2MzAwMG
NjZTllYjc2MGQ2MjM0MWUyMjY3NTdlYjU5YjM2MGI2OTFjY2JlNDRlZjJhYTY3ZTViYWZmZCIsICJhYXMtdGNiaW5mb2NlcnRzaGF
zaCI6ICIxYjJhN2NkNTk2NjkyOWFlYzFlMDYxZjQ3YWJhMDNhZTQyMmMxMWU5ZjhkY2U0MjEyYzJiYjM2MDkyMjJmMzI0IiwgImFh
cy10Y2JpbmZvY3JsaGFzaCI6ICIxN2VjOTc0NWY3ZTNkZGJkNWVkMmU5YWI5ODk3NzNiZTQ0NmMyZjIwMGY2ZDA4Y2ViYjAzYTk0Y
mUyNDY5MjczIiwgImFhcy10Y2JpbmZvaGFzaCI6ICIwZTFjNDEwNTZlMWE1OTA4YmY3ZmEyYmQxZDk5YWJjY2IwYWIzYmM5OTI1M2
NjNDBhNWE1YWNkMTY1YTU3ZWI2In0sICJhYXMtZWhkIjogIlozUnZaZnhqcEpTelN0aFVpWUliRjhBaGZwcEVCcGZlakRPUVpMTjJ
5MWY4dU5aTXZRaThNaVVhN1dCLU9Xcm1HbVU3YW1xcXVHSGtSWnBhZWRQTE4zY0VzQVZ0S2lQRnlEZGVBcHhkcHlkNEFOTUR1TFI1
bndQelF6bnROOTFpY09uM19pMkI0bXJ2OEV2Z0hQSllmeXFtUGUtZE1BbHBvbUNTRkdHVWdNT0xmTjNSNG5MWDYxbmlaMHBoRldhc
kE0MjMtMndJb1VwdWFIOFFycWV3ZFdjMGlWOXNEcTFLYS04OE1tY3VfYi00Z3hmQlB4LWEyWGRuZW5kT3ptVG5EdjRhVE9oT3ZNQX
FPQUkzeFpHdlBreHc3VHFhOWN0Z3F2eU52VlpXSXRTSFNVYlhQcTJHZmRwTll6bE9JSGpyU3pybHh1MGxaVUNJYWJnRXJ5Snd0ZyI
sICJhYXMtcG9saWN5SGFzaCI6ICJlNDI3NGY1M2MwNTdkYjYzNTc4ZDJiYTRlYTBlYjY0ZTg0ZDFkNzE0YzM4ZTM5OTk2MTI3N2Qz
MzRlNDQ2ZWNhIiwgImV4cCI6IDE1ODU3NzUzMjAsICJpYXQiOiAxNTg1NzQ2NTIwLCAiaXMtZGVidWdnYWJsZSI6IHRydWUsICJpc
3MiOiAiaHR0cHM6Ly9tYWExLnVzLmF0dGVzdC5henVyZS5uZXQiLCAibmJmIjogMTU4NTc0NjUyMCwgInByb2R1Y3QtaWQiOiAxLC
Aic2d4LW1yZW5jbGF2ZSI6ICJmOGIxZjk0M2IyZDk1NzI3YzJhYzIwY2FhYzA3MDQ1ZjM5N2I3MGNkOWI1YjBhZDlmZmMxODQwMTB
jY2U4OTQ3IiwgInNneC1tcnNpZ25lciI6ICJjMjM4ODdlODJlMWIyMDY3ZmQ1OGIwYjZmYWZkZTVhYzQxMzZkYTdlZWIwODg2N2Fi
MGMwNzdjN2ZlOWRlZTk4IiwgInN2biI6IDEsICJ0ZWUiOiAic2d4In0.ESIgWQYOgGodVhgqgoazTlLD__DJa7APHh4OHL0cr3IKd
CRH2Z3OuXVJIxLH1EwhZ8HiCnRW1U77BT4Dc2Ra04sJYZyQiNO0gCyfvLDFAG6Mcqdcm3hKjxgvc_xXHoU3LRaqngET83iYK0Dv6Y
cei4iaeD2cipupVu4rge4GCXZ_K98BiWWAKJMnE-odnzQnk_KqqjBXQQU9D8F_ISsMKAyVsDeRKKLOZ8Q7cBLXn-
KmPBb3Jp60wO4olN91tmjQ431KOUSv2YXqylIX6VxBm4fXEe2mnUDNoE6kGOJsrMt3F2LsaD2ydbzZCfdIw5EY5hXqhtATA9yAuYy
lNTUpEQ
```

[86] Base64Url: https://en.wikipedia.org/wiki/Base64

5. Copy the result.
6. To decode the result, from the previous browser session, navigate to https://jwt.ms/, and paste the content.



Once decoded, the example of header for the token is:

```
{
  "alg": "RS256",
  "jku": "https://maatest.us.attest.azure.net/certs",
  "kid": "7ASd0UycYRvHJIP/+19WDf/5mhWAPr06HW50PaEa2wA=",
  "typ": "JWT"
}
```

The body of the JWT token generated by your attestation provider includes a set of claims which describe the collateral used to perform the attestation.

The `exp`, `iat`, `iss`, and `nbf` claims are defined by the JWT RFC 7800, the remaining claims were generated by the azure attestation service.

In particular, the following claims are present:

Decoded Token | Claims


```
{
  "alg": "RS256",
  "jku": "https://maatest.us.attest.azure.net/certs",
  "kid": "7ASd0UycYRvHJIP/+19WDf/5mhWAPrO6HW50PaEa2wA=",
  "typ": "JWT"
}.{
  "aas-attestationcollateral": {
    "aas-qeidcertshash": "1b2a7cd5966929aec1e061f47aba03ae422c11e9f8dce4212c2bb3609222f324",
    "aas-qeidcrlhash": "17ec9745f7e3ddbd5ed2e9ab989773be446c2f200f6d08cebb03a94be2469273",
    "aas-qeidhash": "89efec5e83102ce524c8b597e11b55d4e06fee24f802dd2e59e3c4a22ebe36d9",
    "aas-quotehash": "c91be264ea93bd06339d73bf72801ae30a2ec843c810469a0d0cca81a1e7ce50",
    "aas-tcbinfocertshash": "1b2a7cd5966929aec1e061f47aba03ae422c11e9f8dce4212c2bb3609222f324",
    "aas-tcbinfocrlhash": "17ec9745f7e3ddbd5ed2e9ab989773be446c2f200f6d08cebb03a94be2469273",
    "aas-tcbinfohash": "0e1c41056e1a5908bf7fa2bd1d99abccb0ab3bc99253cc40a5a5acd165a57eb6"
  },
  "aas-ehd": "W6HxgSwv0VWAvK6N75b2JmBf0paaCR7TlA-79WiSnBRgEO2a1PvqYrkkwvOkRoPJ7YMikDaIwDUosX5BnUDKfePS-
DNY73MUTNsyCzPXkZIvSXagGfuVHTW61GZ_GF_1bXlER0RkpVWcIil6r1JDRfMOBZp5HNVNw9tCeC4PFZLy7eeAuTXpZGAD_3SygqSDl6qgpsTkJC-
bUgvbnabkBHF56S1qiVtK5_TN1pMk4rFN-xSnYvACJ4CpPzul7mkcsoC38yh9zTuXuK-REfNiuZYZly_dAOJUC9CgIRW3nYkTiP38k7-
eMgzeOWr6GGH6nuPoYXMcsaECYaCe0c2-IQ",
  "aas-policyHash": "e4274f53c057db63578d2ba4ea0eb64e84d1d714c38e399961277d334e446eca",
  "exp": 1585244502,
  "iat": 1585215702,
  "is-debuggable": true,
  "iss": "https://maatest.us.attest.azure.net",
  "nbf": 1585215702,
  "product-id": 1,
  "sgx-mrenclave": "f8b1f943b2d95727c2ac20caac07045f397b70cd9b5b0ad9ffc184010cce8947",
  "sgx-mrsigner": "b82156c4bcc647b97edc2f6fac254be000e65f535ea84c7e19669980133ff5d6",
  "svn": 1,
  "tee": "sgx"
}.[Signature]
```


- `"aas-attestationcollateral"`. A JSON object which describes the attestation collateral used to perform the attestation (SGX only). This JSON object will contain the following properties:
    - `"aas-quotehash"`. The SHA256 hash of the incoming attestation evidence
    - `"aas-tcbinfohash"`. The SHA256 hash of the TCBInfo used to perform the attestation
    - `"aas-tcbinfocrl"`. The SHA256 of the CRL used to validate the tcbinfo
    - `"aas-qeidhash"`. The SHA256 of the Quoting Enclave ID used to perform the attestation
    - `"aas-qeidcrl"`. The SHA256 of the Quoting Enclave ID CRL
- `"aas-policyhash"`. The SHA256 hash of the policy JWS applied after the evidence was validated.
- `"aas-ehd"`. The Enclave Held Data (EHD) that has been attested.
- `"is-debuggable"`. Specifies if the enclave is in debug mode or not
- `"iat"`, `"nbf"`, `"exp"`. Timestamps corresponding to the issuance, validity start, and expiration times.
- `"iss"`. The URI of your attestation provider STS service, identifying the attestation signing key.
- `"sgx-mrenclave"`. The value of this claim is a string whose value is the hex encoded value of the `"mrenclave"` field of the quote
- `"sgx-mrsigner"`. The value of this claim is a string whose value is the hex encoded value of the `"mrsigner"` field of the quote
- `"svn"`. The value of this claim is the security version number encoded in the quote
- `"tee"`- specifies the type of Trusted execution Environment

You can extract the content of `aas-ehd`, decode it (Base64 encoded) and show that the content is a public key certificate.

> **Note** If the HTTP response is a value other than 200, other values can be returned.

The JWT token is signed so you can also validate that the token is originated from a trusted attestation provider.

Et voilà!

This concludes this illustration, as well as this starter guide.

We hope that you enjoyed this guide tour in the digital world of the attestation for your TEE- based applications.

Appendix. Frequently used acronyms

The following table lists the frequently used acronyms with TEE and Microsoft Azure Attestation

Acronym	Description
AAD	Azure Active Directory
AARP	Azure Attestation Resource Provider
AAS	Azure Attestation Service
ADE	Azure Disk Encryption
AIK	Attestation Identity Key
AKV	Azure Key Vault
AME	Azure Management Environment
ARM	Azure Resource Manager
CRP	Compute Resource Provider
CRTM	Core Root of Trust for Measurement
DAA	Direct Anonymous Attestation
EK	Endorsement Keys
GAS	Guest Attestation Service
HAS	Host Attestation Service
HGS	Host Guardian Service
HSM	Hardware Security Module
IVM	Isolated VM
KPS	Key Protector Service
MAC	Memory Access Control
PCR	Platform Configuration Registers
PUF	Physical Undone-able Functions
RA	Remote Attestation
RoT	Root Of Trust
RPs	Resource Providers
RTM	Root of Trust for Measurement
SB	Secure Boot
SGX	Software guarded Extensions
SRK	Storage Root Keys
TBS	TPM Base Services
TCB	Trusted Computing Base

TEE	Trusted Execution Environment
THC	Trusted Health Certificate
TPM	Trusted Platform Modules
UEFI	Unified Extensible Firmware integration
UMCI	User Mode Code Integrity
VBS	Virtualization Base Security
VHD	Virtual hard Disk
VSM	Virtual Secure Mode
VTL	Virtual Trusted Level
vTPM	Virtual TPM

Microsoft France
39 Quai du Président Roosevelt
92130 Issy-Les-Moulineaux