Quarkslab's blog

Android Android, ReverseEngineering Challenge Cryptography Development Exploitation Fuzzing Archives

Quarkslab's website

SOCIAL

atom feed

twitter

github

CATEGORIES

Android

Android, ReverseEngineering

Challenge

Cryptography

Development

Exploitation

Fuzzing

Hardware

Hardware, ReverseEngineering

Kernel Debugging

Life at Quarkslab

Maths

Obfuscation

PenTest

Program Analysis

Programming

ReverseEngineering

Software

Vulnerability

TAGS

Overview of Intel SGX - Part 1, SGX Internals

Date Thu 05 July 2018 **By** Alexandre Adamski **Category** ReverseEngineering. **Tags** Trusted Execution Environment Intel SGX

This blog-post provides the reader with an overview of the Intel SGX technology. In this first part, we explore the additions made to Intel platforms to support SGX, focusing on the processor and memory. We then explain the management and life cycle of an enclave. Finally, we detail two features of enclaves: secret sealing and attestation.

Introduction

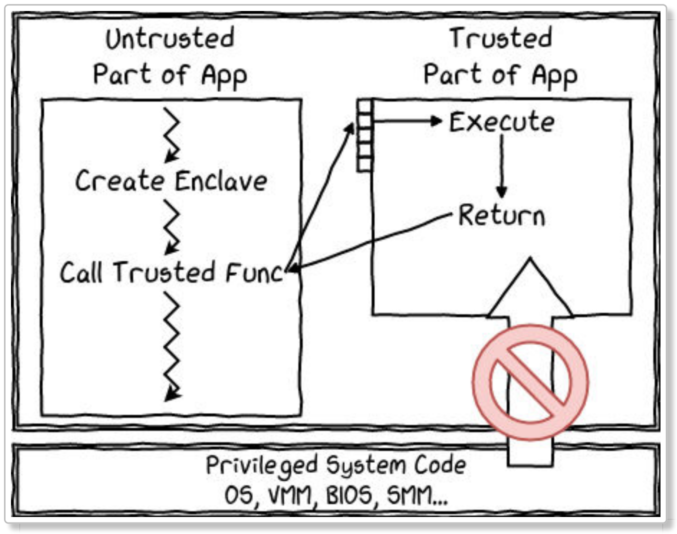
Intel SGX is a technology that was developed to meet the needs of the *Trusted Computing* industry, in a similar fashion to the *ARM TrustZone*, but this time for desktop and server platforms. It allows user-land code to create private memory regions, called enclaves, that are isolated from other processes running at the same or higher privilege levels. The code running inside an enclave is effectively isolated from other applications, the operating system, the hyper-visor, et cetera.

It was introduced in 2015 with the sixth generation Intel Core processors, which are based on the Skylake micro-architecture. SGX support can be checked by executing the `CPUID` instruction with the *Structured Extended Feature Leaf* flag set, and checking if the second bit of the EBX register is set. To be able to use SGX, it must have been enabled by the BIOS, and only a few BIOSes actually support this technology. That is one of the reasons it is not widely used.

Overview

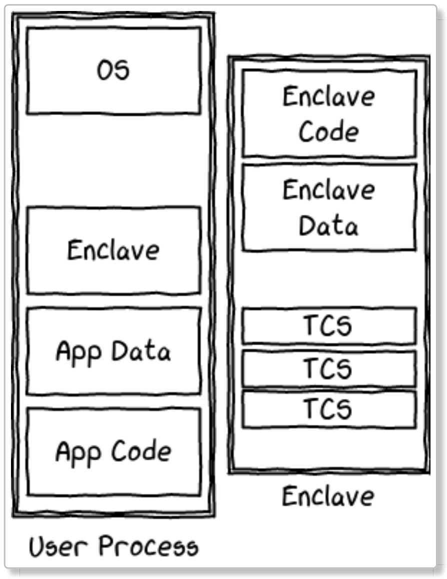
The implementation of Intel SGX can be summarized in a few points:

- an application is split into two parts: a secure one and a non-secure one;
- the application launches the enclave, which is placed in protected memory;
- when an enclave function is called, only the code within the enclave can see its data, external accesses are always denied; when it returns, enclave data stays in the protected memory.



The secure execution environment is part of the host process, which means that:

- the application contains its own code, data, and the enclave;
- the enclave contains its own code and its own data too;
- SGX protects the confidentiality and integrity of the enclave code and data;
- enclave entry points are pre-defined during compilation;
- multi-threading is supported (but not trivial to implement properly);
- an enclave can access its application's memory, but not the other way around.



Instructions

Intel SGX defines 18 new instructions: 13 to be used by the supervisor and 5 by the user. All these instructions are implemented in micro-code (so that their behavior can be modified). See below for the complete instructions list.

Super.	Description	User	Description
`EADD`	Add a page	`EENTER`	Enter an enclave
`EBLOCK`	Block an EPC page	`EEXIT`	Exit an enclave
`ECREATE`	Create an enclave	`EGETKEY`	Create a cryptographic key
`EDBGRD`	Read data by debugger	`EREPORT`	Create a cryptographic report
`EBDGWR`	Write data by debugger	`ERESUME`	Re-enter an enclave
`EINIT`	Initialize en enclave		
`ELDB`	Load an EPC page as blocked		
`ELDU`	Load an EPC page as unblocked		
`EPA`	Add a version array		
`EREMOVE`	Remove a page from EPC		
`ETRACE`	Activate EBLOCK checks		
`EWB`	Write back/invalidate an EPC page		

Structures

Intel SGX also defines 13 new data structures: 8 are used for enclave management, 3 for memory page management, and 2 for resources management. See below for the complete structures list.

- *SGX Enclave Control Structure* (`SECS`)
- *Thread Control Structure* (`TCS`)
- *State State Area* (`SSA`)
- *Page Information* (`PAGEINFO`)
- *Security Information* (`SECINFO`)
- *Paging Crypto MetaData* (`PCMD`)
- *Version Array* (`VA`)
- *Enclave Page Cache Map* (`EPCM`)
- *Enclave Signature Structure* (`SIGSTRUCT`)
- *EINIT Token Structure* (`EINITTOKEN`)
- *Report* (`REPORT`)
- *Report Target Info* (`TARGETINFO`)
- *Key Request* (`KEYREQUEST`)

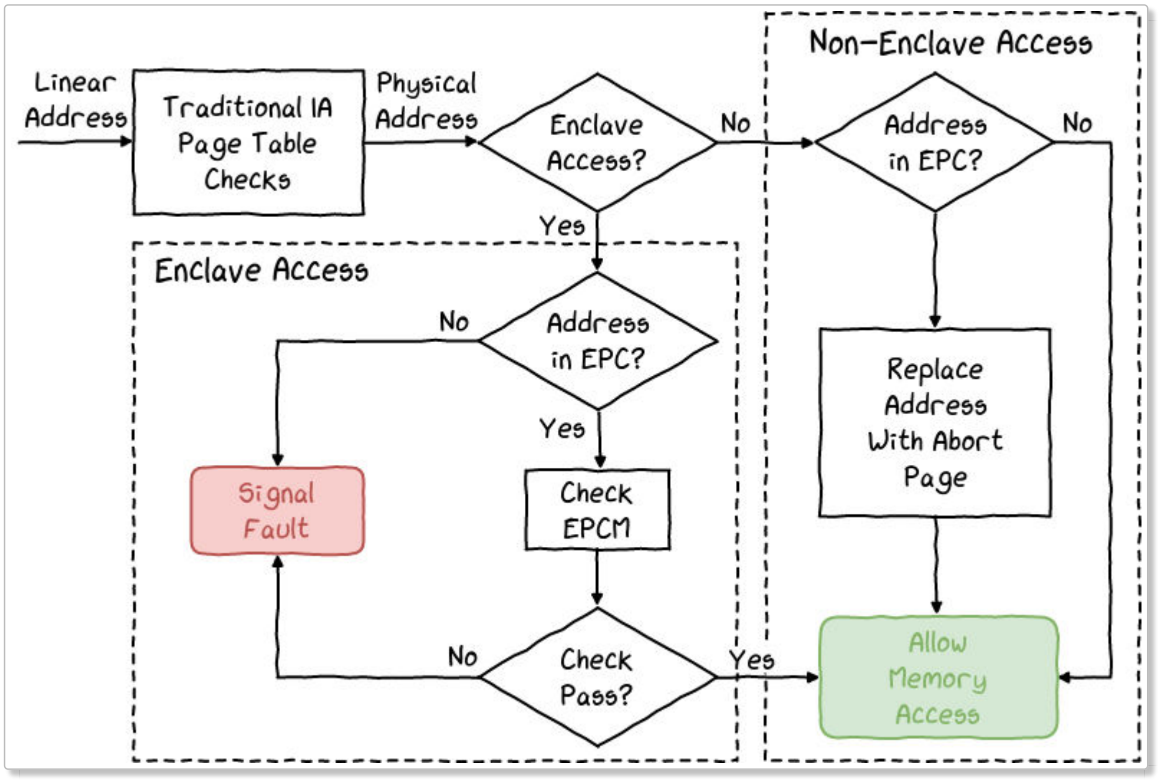
In the following sections, we detail the relevant instructions and structures, before giving a high-level explanation of how they are used to achieve some functionality.

Memory

Enclave Page Cache (EPC)

Enclave code and data are placed in a special memory area called the *Enclave Page Cache* (EPC). This memory area is encrypted using the *Memory Encryption Engine* (MEE), a new and dedicated chip. External reads on the memory bus can only observe encrypted data. Pages are only decrypted when inside the physical processor core. Keys are generated at boot-time and are stored within the CPU.

The traditional page check is extended to prevent external accesses to EPC pages.



Enclave Page Cache Map (EPCM)

The *Enclave Page Cache Map* (EPCM) structure is used to store the pages state. It is located inside the protected memory and its size limits the size of the EPC (set by the BIOS, 128MB maximum). It contains the configuration, permissions and type of each page.

Memory Management

Structures

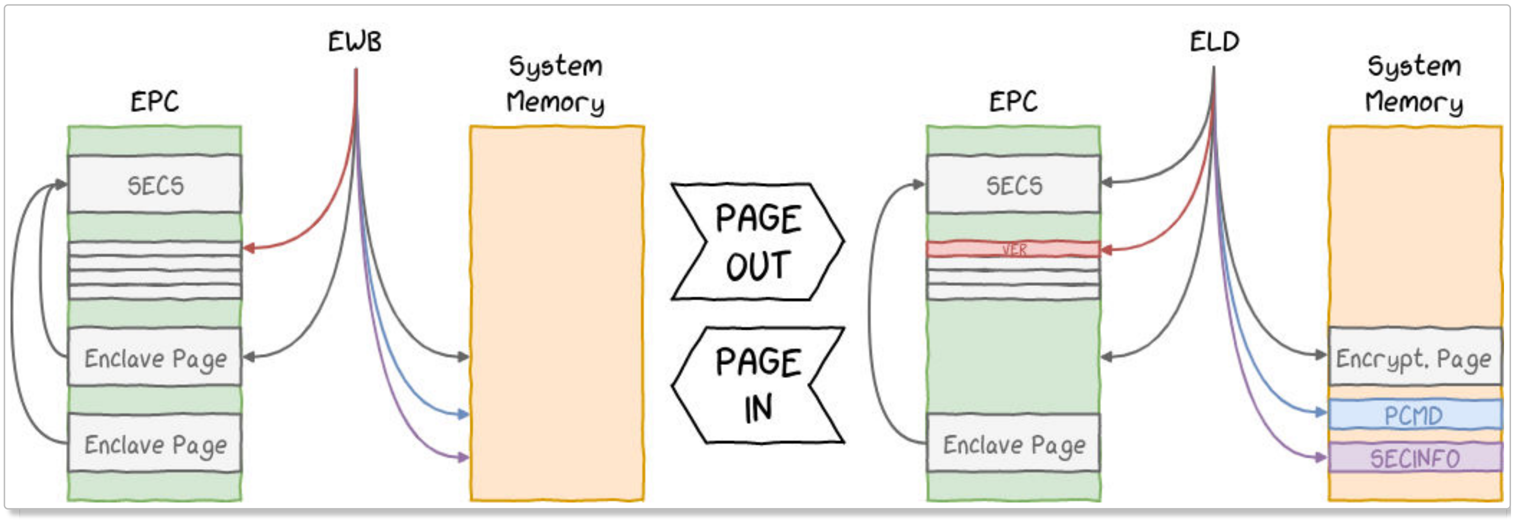
Page Information (PAGEINFO)
The `PAGEINFO` structure is used as a parameter to EPC management instructions to reference a page. It contains its linear and virtual addresses, and pointers to `SECINFO` and `SECS` structures.
Security Information (SECINFO)
The `SECINFO` structure is used to store page meta-data: access rights (read/write/execute) and type (`SECS`, `TCS`, `REG`, or `VA`).
Paging Crypto MetaData (PCMD)
The `PCMD` structure is used to track the meta-data associated to an evicted page. It contains the identity of the enclave the page belongs to, a pointer to a `SECINFO` structure and a MAC.
Version Array (VA)
The `VA` structure is used to store the version numbers of pages evicted from the EPC. It is a special page type that contains 512 slots of 8 bytes to store the version numbers.

Instructions

EPA - This instruction allocates a 4KB memory page that will contain the pages version number array (VA) to protect against replay. Each element is 64 bits long.
EBLOCK - This instruction blocks all accesses to the page being prepared for eviction. All future accesses to this page will result in a page fault ("page blocked").
ETRACK - This instruction evicts a page from the EPC. The page must have been prepared properly: it must be blocked and must not be referenced by the TLB. Before writing it into the external memory, the page is encrypted, and a version number and meta-data are generated, and a final MAC is performed.
ELDB/ELDU - This instruction loads into memory a previously evicted page, in a blocked state or not. It checks the MAC of the meta-data, version number (from the corresponding VA entry), and the page encrypted content. If the verification succeeds, the page content is decrypted and placed inside the chosen EPC page, and the corresponding VA entry deleted.

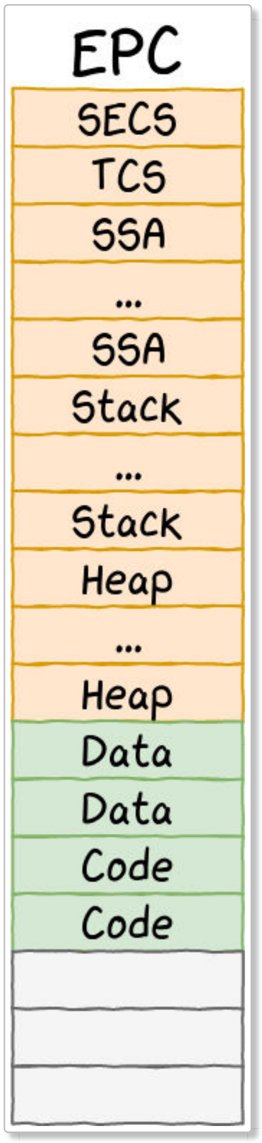
Explanations

The EPC memory is defined by the BIOS and limited in size. SGX has a way for removing a page from the EPC, placing it in unprotected memory, and restoring it later. Pages maintain the same security properties thanks to the EPC pages management instructions, that allow to encrypt an page and produce additional meta-data. A page cannot be removed until all cache entries referencing this page have removed from all processor logical cores. Content is exported or imported with a granularity of a page (which is 4KB).



Memory Content

SGX Enclave Control Structure (SECS)
Each enclave is associated with a SECS structure, which will contain its meta-data (e.g. its hash and size). It is not accessible to any secure nor non-secure code, only by the processor itself. It is also immutable once instantiated.
Thread Control Structure (TCS)
Each enclave is associated with at least a TCS structure, which indicates an execution point into the enclave. As SGX supports multi-threading, an enclave can as many active threads as it has TCS. Like the SECS structure, it is only accessible by the processor, and is also immutable.
Save State Area (SSA)
Each TCS is associated with at least a SSA structure, which is used to save the processor's state during the exceptions and interruptions handling. It is written when exiting, and read when resuming.
Stack and Heap
Each enclave can use its stack and heap. The RBP and RSP registers are saved when entering and exiting, but their value is not changed. The heap is not handled internally, enclaves need their own allocator.



Processor

Enclave Creation

Measures

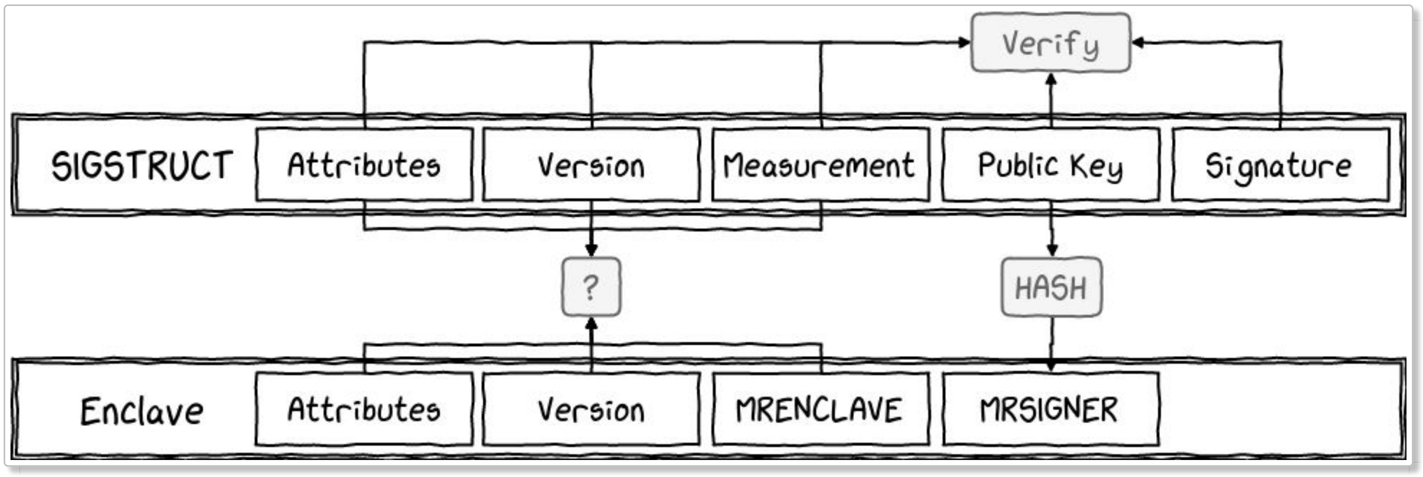
Enclave Measure
Each enclave is represented by a hash of both its attributes and the position, content and protection of its pages. Two enclaves with the same hash are identical. This measure is called `MRENCLAVE` and is used to check the integrity of the enclave.
Signer Measure
Each enclave is also signed by its author. `MRSIGNER` contains the hash of the public key of the author. `MRENCLAVE` and `MRSIGNER` are produced using the SHA-256 hash function.

Structures

EINIT Token Structure (EINITTOKEN)
The `EINITTOKEN` structure is used by the `EINIT` instruction to check if an enclave is allowed to execute. It contains the attributes, hash and signer identity of the enclave. It is authenticated using a HMAC performed with the *Launch Key*.
Enclave Signature Structure (SIGSTRUCT)
Each enclave is associated with a `SIGSTRUCT` structure, which is signed by its author and contains the enclave measure, signer public key, version number (`ISV`, reflecting the security level) and product identifier (`ISVPRODID`, to distinguish between enclaves from the same author). It allows to ensure that the enclave hasn't been modified and then re-signed with a different key.

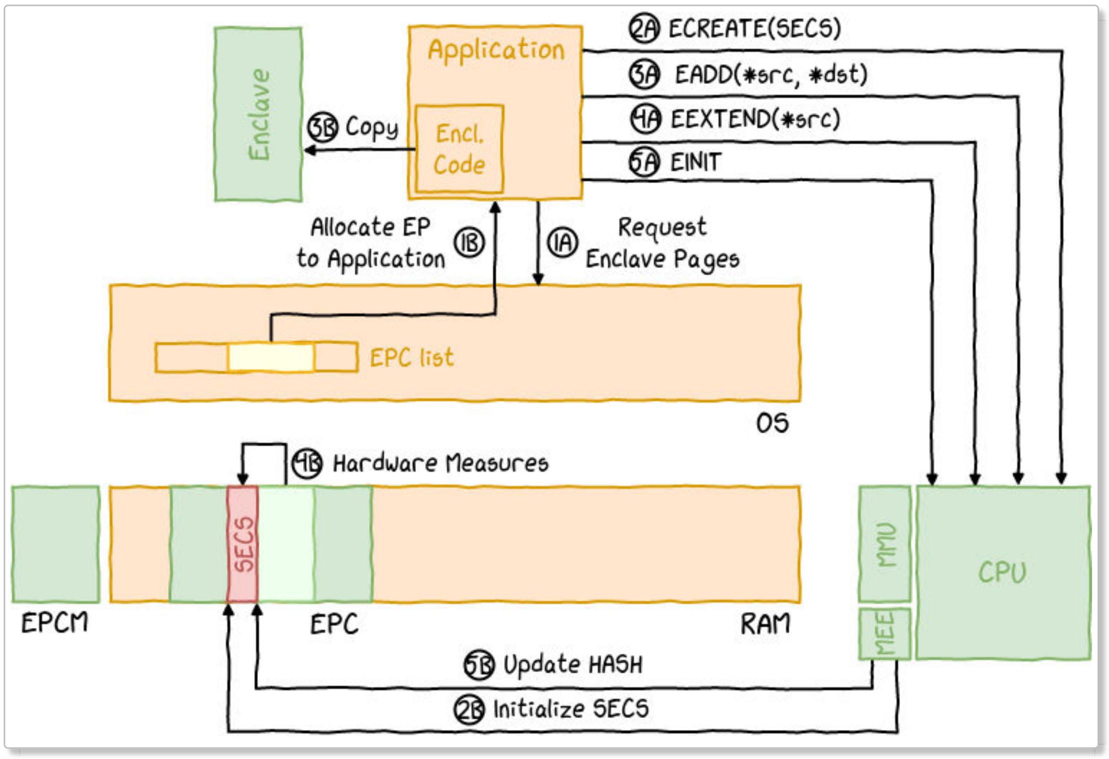
Instructions

ECREATE - This instruction instantiates a new enclave, defining its address space and root of trust. Those informations are stored in a newly allocated SECS.
EADD - This instruction allows to add a new page to the enclave. The operating system is solely responsible for choosing the page and its content. The initial entry for the EPCM denotes the page type and its protection.
EEXTEND - This instruction allows to add a page's content to the enclave measure, by block of 256 bytes. It must be called 16 times to add a complete page to the measure.
EINIT - This instruction checks that enclave corresponds to its EINITTOKEN (same measure and attributes) before initializing it. It also checks that the token is signed with the *Launch Key*.
EREMOVE - This instruction removes permanently a page from the enclave.



Explanations

1. The application requests the loading of its enclave into memory;
2. The ECREATE instruction creates and fills the SECS structure;
3. Each page is loaded into protected memory using the EADD instruction;
4. Each page is added to the measure of the enclave using the EEXTEND instruction;
5. The EINIT instruction finalizes the enclave creation.



Enclave Entry/Exit

Instructions

EENTER - This instruction transfers the control from the application to a pre-determined location within the enclave. It checks that the TCS is free and purges the TLB entries. It then puts the processor in enclave mode and saves the RSP/RBP and XCR0 registers. Finally, it disables the *Precise Event Based Sampling* (PEBS) to make the enclave execution appear as one giant instruction.
EEXIT - This instruction puts the process back in its original mode and purges the TLB entries for addresses located within the enclave. Control is transferred to the address located within the application and specified in the RBX register, and the TCS structure is freed. The enclave needs to clear its registers before exiting to prevent data leaks.

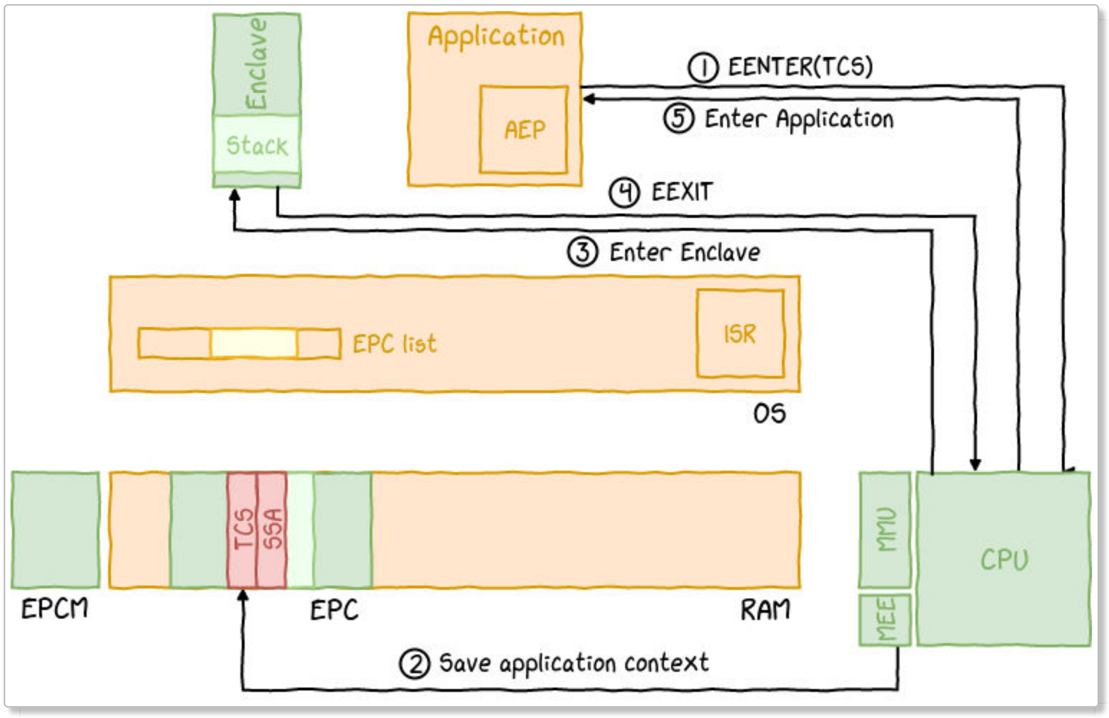
Explanations

Enclave Entry

1. EENTRY instruction is executed;
2. The application context is saved;
3. The processor is put in enclave mode.

Enclave Exit

4. EEXIT instruction is executed;
5. The processor is put in normal mode.



Interrupt Handling

Instructions

ERESUME - This instruction restores the context from the current SSA and resume the execution.

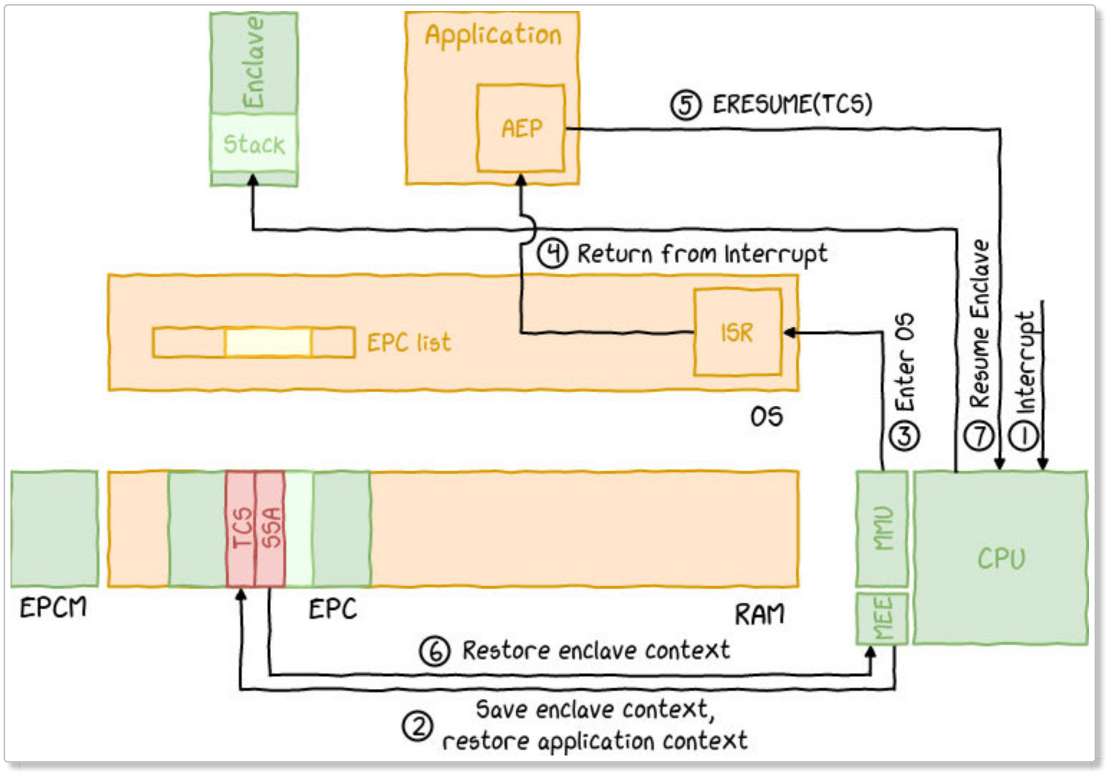
Explanations

Interruptions and exception result in *Asynchronous Enclave Exits* (AEX). The *Asynchronous Exit Pointer* (AEP) points to a handler located inside the application that will resume the execution after the exception has been handled by the *Interrupt Service Routine* (ISR). The handler can decided to resume or not the execution of the enclave, by executing the ERESUME instruction.

When an AEX happens, the context of the enclave is saved in the current SSA and the application context is restored. The enclave context is restored when the ERESUME instruction is executed. The TCS contains a counter denoting the current SSA, forming a stack of contexts.

Handling an Interruption

1. The interruption or exception arrives to the processor;
2. The enclave context is saved, the application context is restored;
3. The execution continues in the handler of the operating system;
4. The handler returns (IRET) to the AEP, a trampoline function;
5. AEP execute ERESUME if it decides to resume enclave execution;
6. The enclave context previously saved is restored;
7. The execution resuming where it stopped within the enclave.



Features

Sealing

Instructions

EGETKEY - This instruction is used by an enclave to access the different keys provided by the platform. Each key enables a different operation (sealing, attesting).

Explanations

When an enclave is instantiated, its code and data are protected from external accesses. But when it stops, all of its data is lost. Sealing is a way of securely saving the data outside of an enclave, on a hard-drive for example. The enclave must retrieve its *Seal Key* using the EGETKEY instruction. It uses this key to encrypt and ensure its data integrity. The algorithm used is chosen by the enclave author.

Using the Enclave Identity

The sealing can be done using the enclave identity. The key derivation is then based on the value of `MRENCLAVE`. Two distinct enclaves have different keys, but also two versions of the same enclave, which prevent the local migration of data.

Using the Signer Identity

The sealing can also be done using the signer identity. The key derivation is then based on the value of `MRSIGNER`. Two distinct enclaves still have different keys, but two versions of an enclave share the same key and can read the sealed data. If multiples enclaves are signed using the same keys, then they can all read each other's data.

Security Version Number (SVN)

Older versions of an enclave should not be allowed to read data sealed by a newer version of an enclave. To prevent it, the *Security Version Number* (SVN) is used. It is a counter incremented after each update impacting the security of the enclave. Keys are derived using the SVN in a way that an enclave can retrieve the keys corresponding to current, or older, security level, but not newer.

Attestation

Structures

Key Request (KEYREQUEST)
The KEYREQUEST structure is used as an input to the EGETKEY instruction. It allows to choose which key to get, and also additional parameters that might be needed for the derivation.

Report Target Info (TARGETINFO)
The TARGETINFO structure is used as an input for the EREPORT instruction. It is used to identify which enclave (hash and attributes) will be able to verify the REPORT generated by the CPU.
Report (REPORT)
The REPORT structure is the output of the EREPORT instruction. It contains the enclave's attributes, measure, signer identity and some user data to share between the source and destination enclaves. The processor performs a MAC over this structure using the *Report Key*.

Instructions

EREPORT - This instruction is used by the enclave to generate a REPORT structure containing multiple informations about it and authenticated using the *Report Key* of the destination enclave.

Explanations

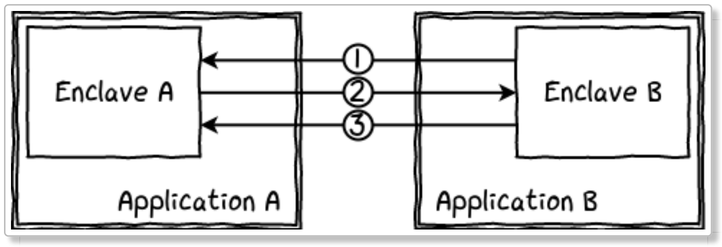
The enclave code and data are in plain-text before its initialization. While sections could technically be encrypted, the decryption key cannot be pre-installed (or it would not provide any additional security). Secrets have to come from the outside, might they be keys and sensitive data. The enclave must be able to prove to a third-party that it can be trusted (has not been tampered with) and is executing on a legitimate platform.

Two types of attestation exists:

- local attestation: an attestation process between two enclaves of the same platform;
- remote attestation: an attestation process between an enclave and third-party not on the platform.

Local Attestation

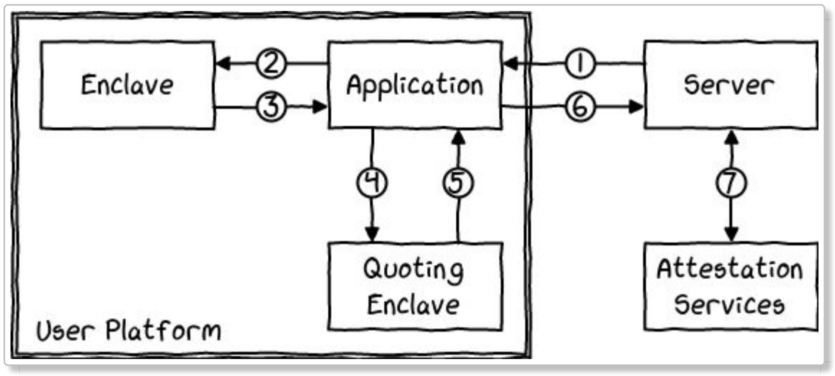
1. A channel must have already been established between enclave A and enclave B. It is used by enclave A to retrieve the MRENCLAVE of B.
2. Enclave A calls EREPORT with the MRENCLAVE of B to generate a signed report for the latter.
3. Enclave B calls EGETKEY to retrieve its *Report Key* and verify the MAC of the EREPORT structure. If valid, the enclave is the one expected and running on a legitimate platform.



Remote Attestation

Remote attestation requires an architectural enclave called the *Quoting Enclave* (QE). This enclave verifies and transforms the REPORT (locally verifiable) into a QUOTE (remotely verifiable) by signing it with another special key, the *Provisioning Key*.

1. Initially, the enclave informs the application that it needs a secret located outside of the platform. The application establishes a secure communication with a server. The server responds with a challenge to prove that the enclave executing has not been tampered with and that the platform it executes on is legitimate;
2. The application gives the *Quoting Enclave* identity and the challenge to its enclave;
3. The enclave generate a manifest including the challenge answer and an ephemeral public key that will be used later to secure the communications between the server and the enclave. It generates a hash of the manifest that it includes in the user data section of the EREPORT instruction. The instruction generates a REPORT for the *Quoting Enclave* that ties the manifest to the enclave. The enclave passes the REPORT to the application.
4. The application transfers the REPORT to the *Quoting Enclave* for verification and signing.
5. The QE retrieves its *Report Key* using the EGETKEY instruction and verifies the REPORT. It creates the QUOTE structure and signs it using its *Provisioning Key* before giving it back to the application.
6. The application sends the QUOTE and associated manifest to the server for verification.
7. The server uses the attestation service provided by Intel to validate the QUOTE signature. It then checks the manifest integrity using the hash from the QUOTE user data. Finally, it makes sure that the manifest contains the expected answer to the challenge.



Conclusion

This blog-post gives an overview of the SGX internals. We've seen how memory is managed, how to load and call an enclave, and we detailed sealing, local and remote attestation.

In the next blog-post, we will present the SGX externals (everything that is not embedded in the CPU). We will talk about the development process of an SGX enclave, the SDK and PSW (Architectural Enclaves).

Comments

6 Comments Quarkslab Disqus' Privacy Policy Login

Recommend 4 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Awais • 5 months ago

State State Area (SSA) ? i think you meant Save State Area (SSA), kindly correct if I am right

• Reply • Share ›

MnemonicX -_ • 3 years ago

Best introduction write-up I've read about SGX. I really appreciate how you summarize the usage and functions of each instruction. Thumbs up!

• Reply • Share ›

Marc H. • 3 years ago

good paper
just wondering which tool you are using for those "hand made" diagrams.

• Reply • Share ›

小书僮 → Marc H. • 3 years ago

These diagrams are awesome! I am also very curious about the tool.

• Reply • Share ›

Alexandre Adamski → Marc H. • 3 years ago

Hello there. I am using the draw.io website with the "Comic" effect to make the distorted lines and a xkcd font for the text.

• Reply • Share ›

asfd • 3 years ago

this is the worst write up -- as far as density is concerned.
There is no head no tail, just a hodge podge pickinup of intel instruction set documentation and copy pasting it.

3 • Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

Powered by Pelican, Theme is from Bootstrap from Twitter