Open Enclave Design Overview

*September 4, 2017*

Table of Contents

[1. Introduction 2](#_Toc507526014)

[2. The Enclave Application 3](#_Toc507526015)

[3. The Inter-Call Model 5](#_Toc507526016)

[OE\_Enter() 7](#_Toc507526017)

[OE\_Main() 7](#_Toc507526018)

[OE\_Exit() 7](#_Toc507526019)

[4. Thread Binding 9](#_Toc507526020)

[Operation 9](#_Toc507526021)

[Implementation 10](#_Toc507526022)

[5. Enclave Creation 12](#_Toc507526023)

[6. Enclave Page Layout 14](#_Toc507526024)

[The Signature Section (.oesig) 14](#_Toc507526025)

[Text Pages 15](#_Toc507526026)

[Data Pages 15](#_Toc507526027)

[Relocation Pages 16](#_Toc507526028)

[ECALL Pages 16](#_Toc507526029)

[Heap Pages 17](#_Toc507526030)

[Thread Context 17](#_Toc507526031)

[Guard Pages 18](#_Toc507526032)

[Per Thread Stack 18](#_Toc507526033)

[Thread Control Structure 18](#_Toc507526034)

[Set-Aside Areas 19](#_Toc507526035)

[Segment Page 19](#_Toc507526036)

[Thread-Specific Data Page 20](#_Toc507526037)

[7. Exception Handling 20](#_Toc507526038)

[Hardware exceptions 20](#_Toc507526039)

[Vectored exception handling 21](#_Toc507526040)

[8. The Signing Tool 22](#_Toc507526041)

[Example 22](#_Toc507526042)

[9. The Debugger Extension 24](#_Toc507526043)

[Ptrace Library 24](#_Toc507526044)

[Python Extension 24](#_Toc507526045)

[Stack Stitching 24](#_Toc507526046)

[10. The IDL Generator [Experimental] 26](#_Toc507526047)

[Supported Data Types 26](#_Toc507526048)

[Function and parameter constraints 27](#_Toc507526049)

[Example 1 28](#_Toc507526050)

[Example 2 28](#_Toc507526051)

# Introduction

This document provides a brief design overview of the **Open Enclave SDK** (**OE**). It describes the parts of the SDK and how they work together to create, invoke, and terminate enclaves. This document assumes the reader is familiar with enclaves.

OE aims to provide an abstract API for developing enclave applications whether they are based on the Intel® Software Guard Extensions (SGX) or other technologies such as Microsoft® Virtualization Based Security (VBS). This early version though only supports SGX, so the design discussion that follows refers exclusively to SGX.

# The Enclave Application

OE helps developers build **enclave applications**. An enclave application is partitioned into an untrusted component (called a **host**) and a trusted component (called an **enclave**). An enclave is a secure container whose memory (text and data) is protected from access by outside entities, including the host, privileged users, and even the hardware. The enclave may run in an untrusted environment with the expectation that secrets will not be compromised.

On Linux, enclaves are packaged as shared objects that have been digitally signed. For example, an enclave called **turtle** might be redistributed with the following filename.

turtle.signed.so

A host may load and instantiate this enclave as shown by the following snippet.

OE\_Enclave\* enclave;

OE\_Create(“turtle.signed.so”, 0, &enclave);

The **enclave** variable is a handle used to refer to the enclave throughout its lifetime. A host may wish to create several enclaves, either of the same type or different types.

Once an enclave is created, the host may invoke its functions, known as **enclave calls** or **ECALLs**. In the snippet below, the host calls the enclave’s **Walk** function.

OE\_CallEnclave(enclave, "Walk", args));

This enters the enclave and calls its Walk function. To service this request, the turtle enclave implements a Walk function with the following signature.

OE\_ECALL void Walk(void\* args);

Once an enclave is entered, the enclave may call functions within the host, called **outside calls** or **OCALLs**. For example, the enclave may call the host’s **WhoAreYou** function as shown in the snippet below.

OE\_CallHost(enclave, "WhoAreYou", args));

To service this request, the host implements a WhoAreYou function with the following signature.

OE\_OCALL void WhoAreYou(void\* args);

The **args** parameter for ECALLs and OCALLs is defined by the developer of the enclave application. It might be a pointer to a string or a C structure. Although **OE\_CallEnclave** and **OE\_CallHost** are not type safe, we will see later how to use the IDL generator (**oegen**) to produce type-safe, parameterized stubs for safely calling those functions.

The host and enclave may continue to exchange ECALLs and OCALLs. ECALLs and OCALLs may be arbitrarily nested as stack memory allows.

Eventually the host will terminate the enclave as shown in the snippet below.

OE\_TerminateEnclave(enclave);

# The Inter-Call Model

OE defines an **inter-call model** whereby the host and enclave call each other’s functions. These may be thought of as messages exchanged between the host and enclave. There are four kinds of messages, defined in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Message** | **Description** | **Sender** | **Receiver** |
| **ECALL** | An enclave call | Host | Enclave |
| **ERET** | An enclave return | Enclave | Host |
| **OCALL** | An outside call | Enclave | Host |
| **ORET** | An outside return | Host | Enclave |

An enclave may perform an OCALL while servicing an ECALL, and a host can perform an ECALL while servicing an OCALL. So ECALLs and OCALLs may be nested arbitrarily. The diagram below illustrates one level of nesting.

**Host**

**Enclave**

**ECALL (call f)**

**OCALL (call g)**

**ORET (return from g)**

**ERET (return from f)**

**f()**

**g()**

ECALLs and OCALLs (and their respective returns) are software constructs. There is no direct analogue with SGX hardware instructions. Rather these calls are implemented using the **EENTER** and **EEXIT** instructions as defined in the table below.

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| **EENTER** | Executed by the host (**OE\_Enter**) to enter the enclave (**OE\_Main**) |
| **EEXIT** | Execute by the enclave (**OE\_Exit**) to return to host (**OE\_Enter**) |

The interaction between the host and enclave during execution of these instructions is depicted below along with the OE functions that are involved.

**Host**

**Enclave**

**EENTER**

**Hardware**

**OE\_Enter()**

**OE\_Main()**

**OE\_Exit()**

**EEXIT**

**return**

**jump**

ECALLs and OCALLs are layered on the EENTER and EEXIT instructions. The table below shows how the four message types are mapped onto the SGX instructions.

|  |  |  |
| --- | --- | --- |
| **Message** | **SGX Instruction** | **Explanation** |
| **ECALL** | **EENTER** | ECALL executes EENTER to enter enclave |
| **ERET** | **EEXIT** | ERET executes an EEXIT to exit enclave |
| **OCALL** | **EEXIT** | OCALL executes EEXIT to exit enclave |
| **ORET** | **EENTER** | ORET executes EENTER to reenter the enclave |

When the host performs an ECALL, it executes an EENTER instruction to enter the enclave. The host blocks until the EENTER instruction returns (when the enclave executes EEXIT). The enclave exits either to perform an ERET or an OCALL. The reason for the exit is returned in a register. The host examines this register to determine what to do next: to process an ERET or to service an OCALL.

The table below identifies key OE functions that participate in ECALLs and OCALLs along with source code references for each.

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Source Location** |
| **OE\_Enter()** | Host calls to execute EENTER to enter an enclave. | host/enter.S |
| **OE\_Main()** | EENTER calls this enclave entry point. | enclave/main.S |
| **OE\_Exit()** | Enclave calls to execute EEXIT to exit the enclave. | enclave/exit.S |

The behavior of these functions is defined below.

## OE\_Enter()

OE\_Enter() executes the EEXIT instruction with the following register assignments.

* RDI: The address of a Thread Control Structure (TCS) in the enclave
* RSI: Address of an Asynchronous Exception Procedure (AEP) in the host
* RDX:
  + High-word: code indicating whether ECALL or ORET
  + Low-word: function number
* RCX: ECALL or ORET argument

EENTER obtains the enclave entry point from the TCS, which is always the OE\_Main() function. EENTER calls OE\_Main(), passing it the return address (the instruction in the host immediately following the EENTER instruction execution). The calling thread executes in the enclave until the enclave calls OE\_Exit(), which executes the EEXIT instruction, which jumps to the return address in the host.

## OE\_Main()

The EENTER instruction calls OE\_Main() to enter the enclave with the following register assignments.

* RAX: index of current SSA (a non-zero value indicates an exception)
* RBX: address of a TCS
* RCX: return address (address of instruction in host immediately following EENTER)
* RDI:
  + High-word: code indicating whether ECALL or ORET
  + Low-word: function number
* RSI: ECALL or ORET argument

OE\_Main() performs the following tasks:

* Saves the hosts registers
* Initializes the enclave stack frame
* Invokes \_\_OE\_HandleMain()

\_\_OE\_HandleMain() handles either an ECALL or an ORET, dispatching as necessary. \_\_OE\_HandleMain() never returns but instead calls OE\_Exit() which executes an EEXIT instruction.

## OE\_Exit()

OE\_Exit() is called to either perform an ERET or an OCALL. It performs the following:

* Clears the enclave registers
* Restores the host registers
* Executes the EEXIT instruction with the following register assignments:
  + RBX – return address
  + RCX – Asynchronous Exception Procedure (AEP)
  + RDI:
    - High-word: code indicating whether OCALL or ERET
    - Low-word: function number
  + RSI: OCALL or ERET argument

The host examines the high-word of RDI (the code) to determine what action to take and the low-word of RDI (the function number) to determine what function to call. The function argument is taken from RSI.

# Thread Binding

This chapter discusses the binding between **host threads** and **enclave thread contexts**. When an enclave is created, it has *N* thread contexts. Each **thread context** consists of the following pages.

* **Thread control structure** (**TCS**) **pages**—used by SGX to maintain thread state
* **Stack pages**—dedicated to that thread context
* **Set aside areas** (**SSA**) **pages**—used by SGX to handle exceptions and interrupts
* **Segment page**—holds thread information (bound to the GS register)
* **Thread-specific data page**—holds word-sized TSD slots (or TLS slots)

For more information about thread context pages, see **Chapter 6** (**Enclave Page Layout**).

## Operation

When a host thread performs an ECALL, the invocation is targeted at one of these thread contexts. The thread binds to the thread context for the duration of the ECALL. When the ECALL returns, this binding is dissolved.

This is simple enough, but what happens when an ECALL performs an OCALL? The same thread that invoked the ECALL performs the OCALL. The thread is depicted as **T1** in the diagram below and thread context is is bound to is depicted as **TC3**. The binding is indicated with the following notation: **[T1,TC3]**.

**Host**

**Enclave**

**f()**

**g()**

**ECALL[T1,TC3]**

**OCALL[T1,TC3]**

**ERET[T1,TC3]**

**ORET[T1,TC3]**

Note that the binding between T1 and TC3 remains in effect until the ECALL returns, and then the binding is dissolved.

Now, what happens when an OCALL performs a nested ECALL? Is the same binding used for the nested ECALL as well? The answer depends on whether the same thread that performed the OCALL is used to make the nested ECALL (note that the host can create a new thread to perform the call). If so, then Open Enclave uses the same binding. This is depicted in the diagram below.

**Host**

**Enclave**

**f()**

**g()**

**ECALL[T1,TC3]**

**OCALL[T1,TC3]**

**ERET[T1,TC3]**

**ECALL[T1,TC3]**

**h()**

**ORET[T1,TC3]**

**ERET[T1,TC3]**

As with the previous example, the binding between T1 and TC3 is dissolved when the original ECALL returns.

## Implementation

Enclaves are created by calling **OE\_CreateEnclave(),** defined as follows.

OE\_Result OE\_CreateEnclave(

const char\* path,

uint32\_t flags,

OE\_Enclave\*\* enclave);

The **enclave** output parameter is an opaque pointer to an internally defined structure. The **OE\_Enclave** structure maintains the following information.

* The base address of the enclave
* The size of the enclave in bytes
* The hash of the enclave
* An array ECALL name-address structures (ECALL array)
* An array of thread bindings (between **host threads** and **enclave thread contexts**)

To call into an enclave, one either calls **OE\_CallEnclave()** or the low-level function **OE\_ECall()**. For example, **OE\_CallEnclave()** is defined below.

OE\_Result OE\_CallEnclave(

OE\_Enclave\* enclave,

const char\* func,

void\* args);

This function performs the following steps.

1. Searches the ECALL array for a function named **func**.
2. If found, it initializes the fields of an **OE\_CallEnclaveArgs** structure as follows:
   1. **OE\_CallEnclaveArgs**.**vaddr**—the virtual address of the enclave ECALL function
   2. **OE\_CallEnclaveArgs**.**func**—the function number of the enclave ECALL function
   3. **OE\_CallEnclaveArgs**.**args**—the **args** parameter to **OE\_CallEnclave()**
3. Calls **OE\_ECall()** with these arguments:
   1. **enclave**—same as the enclave argument passed to **OE\_CallEnclave()**
   2. **func**—the predefined **OE\_FUNC\_CALL\_ENCLAVE** constant
   3. **args**—the **OE\_CallEnclaveArgs** structure initialized above
4. **OE\_ECall()** function performs these steps.
   1. Finds an available enclave thread context.
   2. Enters the enclave, targeting the thread context found in the previous step
   3. Waits for the ECALL to return

If **OE\_ECall()** finds an available thread context, it marks it as busy, else it returns **OE\_OUT\_OF\_THREADS**. When the ECALL returns, **OE\_ECall()** releases the thread context, clearing the busy flag.

If the enclave performs an OCALL before **OE\_ECall()** returns, the host might perform a nested ECALL. If it does, steps 1 through 4 above are repeated using the same enclave thread context used in the original ECALL.

# Enclave Creation

The host is responsible for creating enclaves (loading, building, and instantiating). This chapter describes this process in detail. The host does this by calling the following function.

OE\_Result OE\_CreateEnclave(

const char\* path,

uint32\_t flags,

OE\_Enclave\*\* enclave);

During enclave creation, the host performs the following steps.

1. Loads the enclave image file into memory (using an ELF-64 loader)
2. Asks the Intel® SGX driver to create the enclave (execute ECREATE)
3. Asks the Intel® SGX driver to add pages to the enclave (execute EADD and EEXTEND)
4. Asks Intel® AESM service’s launch enclave for a launch token
5. Asks the Intel® SGX driver to initialize the enclave (execute EINIT) passing the launch token

These last four steps are depicted in the diagram below.

**Host**

**Driver**

**Hardware**

**AESM**

ENCLAVE\_CREATE(SECS)

ECREATE

ENCLAVE\_ADD\_PAGE

EADD/EEXTEND

EADD/EEXTEND

**.**

**.**

**.**

EADD/EEXTEND

GetLaunchToken

ENCLAVE\_INIT

(SIGSTRUCT, EINITTOKEN)

EINIT

ENCLAVE\_ADD\_PAGE

ENCLAVE\_ADD\_PAGE

The Intel® SGX driver is an **ioctl** driver that defines the requests defined in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **OCTL Request** | **Description** | **SGX Instructions** | **SGX Structures** |
| **ENCLAVE\_CREATE** | Reserves enclave memory. | ECREATE | SECS |
| **ENCLAVE\_ADD\_PAGE** | Adds a page to the enclave and optionally extends it. | EADD, EEXTEND |  |
| **ENCLAVE\_INIT** | Finalizes the enclave measurement (MRENCLAVE) and initializes the enclave. | EINIT | SIGSTRUCT, EINITTOKEN |

The SGX structures in the third column below are briefly described in the following table.

|  |  |  |
| --- | --- | --- |
| **Structure** | **Description** | **SGX Instruction** |
| SECS | Specifies the base, size, and flags of the enclave | ECREATE |
| SIGSTRUCT | Specifies the public key and enclave hash (MRENCLAVE) | EINIT |
| EINITTOKEN | An initialization token obtained from the Intel® launch enclave | EINIT |

For more details about SGX instructions and structure, see the following document.

[Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 3D](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf)

# Enclave Page Layout

During enclave creation, the host asks the driver to add pages to the enclave, which results in an enclave image that resides in the enclave page cache (EPC). The diagram below depicts the layout of these pages.

**Data Pages**

**Relocation Pages**

**ECALL Pages**

**Heap Pages**

**Thread Context 1**

**Thread Context 2**

**Thread Context N**

**.**

**.**

**.**

**Guard Page**

**Stack Pages**

**Guard Page**

**TCS**

**SSA1**

**SSA2**

**Guard Page**

**Segment Page**

**Thread Specific Data**

**Text Pages**

The following sections describe each of these.

## The Signature Section (.oesig)

The following sections refer to information obtained from the image file’s signature section (.oesig), which the **oesign** utility injects into the image file during signing (see Chapter 7). This section defines the following structure.

typedef struct \_OE\_SignatureSection

{

oe\_uint64\_t magic;

OE\_EnclaveSettings settings;

SGX\_SigStruct sigstruct;

}

OE\_SignatureSection;

This structure defines **settings** (which specifies the number of heap pages, the number of stack pages, and the number of TCSs) and **sigstruct** (which contains the digital signature formed over certain pages of the image).

## Text Pages

The text pages are copied from the text segment of the ELF-64 image file. The first few bytes are the ELF-64 header structure (Elf64\_Ehdr). This structure contains the virtual address of the entry point (Elf64\_Ehdr.e\_entry), which refers to the **OE\_Main()** function. During enclave creation, the following fields are cleared:

Elf64\_Ehdr.e\_shoff

Elf64\_Ehdr.e\_shnum

Elf64\_Ehdr.e\_shstrndx

Otherwise, the text segment is copied from the image file exactly as it is.

The host asks the driver to add and extend (measure) the text pages.

## Data Pages

The **data pages** are copied from the data segment of the ELF-64 image file. OE initializes the following global variables during enclave creation.

|  |  |
| --- | --- |
| **Variable** | **Description** |
| \_\_oe\_baseRelocPage | Page number of first relocation page relative to start of image |
| \_\_oe\_numRelocPages | Total number of relocation pages (obtained from the image file) |
| \_\_oe\_baseECallPage | Page number of first ECALL page relative to start of image |
| \_\_oe\_numECallPages | Total number of ECALL pages (obtained from the image file) |
| \_\_oe\_baseHeapPage | Page number of first heap page relative to start of image |
| \_\_oe\_numHeapPages | Total number of heap pages (obtained from .oesig section) |
| \_\_oe\_numPages | Total number of pages in the image |
| \_\_oe\_virtualBaseAddr | Virtual address of the \_\_oe\_virtualBaseAddr variable itself |

The **relocation page** variablesare discussed in the next section. The **heap page** variables are used to determine the boundaries of the heap (used by the heap allocator). The **\_\_oe\_virtualBaseAddr** is the virtual address of the variable itself. The real base address of the enclave can be obtained by subtracting this value from the address of the variable itself. For example:

(unsigned char\*)&\_\_oe\_virtualBaseAddr - \_\_oe\_virtualBaseAddr;

The host asks the driver to add and extend (measure) the data pages.

## Relocation Pages

The **relocation pages** contain symbol relocations that must be applied the first time the enclave is entered. The enclave must be self-relocating since applying relocations during enclave creation would obviously break the enclave measurement (MRENCLAVE). Recall from the table above, that the following variables tell the enclave where it can find the base relocation pages and how many there are.

\_\_oe\_baseRelocPage

\_\_oe\_numRelocPages

The content of these pages consists of zero or more structures defined below.

typedef struct \_OE\_Reloc

{

oe\_uint64\_t offset;

oe\_uint64\_t info;

oe\_int64\_t addend;

}

OE\_Reloc;

This structure has the same layout as the **Elf64\_Rela** structure.

|  |  |
| --- | --- |
| **OE\_Reloc field** | **Description** |
| offset | Virtual offset to symbol to be relocated (zero marks last struct) |
| info | Ignored if not R\_X86\_64\_RELATIVE |
| addend | Add to enclave’s base address to relocate the symbol |

The host asks the driver to add and extend (measure) the relocation pages.

## ECALL Pages

The ECALL pages contain the virtual addresses of all ECALL functions in the enclave. The following structure defines the content of these pages.

typedef struct \_OE\_ECallPages

{

uint64\_t magic;

uint64\_t num\_vaddrs;

uint64\_t vaddrs[];

}

OE\_ECallPages;

When a host performs an OCALL, it passes a **function number** and the expected **virtual address** of the function. The function number is an index into the **OE\_ECallPages.vaddrs** array. The enclave checks that this index is within bounds and if so obtains the virtual address from the array. Then the enclave checks that the virtual address passed by the host matches the virtual address obtained from the array. If all checks are valid, the enclave adds the virtual address to the enclave’s base address to obtain and call the function.

## Heap Pages

The heap pages are zero-filled pages. Recall that the number of heap pages is obtained from the signature structure (read from the enclave image file). The following global variables (in the data pages) tell the enclave where the heap pages are located and how many there are.

\_\_oe\_baseHeapPage

\_\_oe\_numHeapPages

The heap allocator relies on these variables as well as the base address of the enclave.

The host asks the driver to add **but not extend** the heap pages.

## Thread Context

There is a thread context for each thread control structure (TCS). Each thread context has the following layout.

**Guard Page**

**Stack Pages**

**Guard Page**

**TCS**

**SSA1**

**SSA2**

**Guard Page**

**Segment Page**

**Thread Specific Data**

**Per thread stack**

**Set-aside areas**

**Thread data structure**

**Thread specific data slots**

**Thread control structure**

## Guard Pages

Non-readable and non-writable guard pages are injected into the thread context at various places. These will cause a fault if the stack is underflowed or overflowed.

## Per Thread Stack

Each TCS has its own stack pages. The number of stack pages per TCS is determined during enclave signing (using the **eisign** tool). The enclave creation code reads this information from the signature section (.oesig) of the enclave image file. These pages are zero-filled pages.

## Thread Control Structure

The thread control structure or TCS is initialized during enclave creation but not accessible to the enclave at runtime. The SGX instructions read and write information to the TCS page during enclave entry and exit. The TCS structure is defined below. The initial value provided during enclave creation is shown in the second column.

|  |  |  |
| --- | --- | --- |
| **TCS field** | **Initially** | **Description** |
| oe\_uint64\_t state | 0 | 0 indicates that TCS is available. EENTER sets to 1. |
| oe\_uint64\_t flags | 0 |  |
| oe\_uint64\_t ossa | address | Virtual address of the first set-aside area slot |
| oe\_uint32\_t cssa | 0 | Current set-aside area slot |
| oe\_uint32\_t nssa | 2 | The number of set-aside areas |
| oe\_uint64\_t oentry | address | Virtual address of the entry point: OE\_Main() |
| oe\_uint64\_t aep | 0 | EENTER stores asynchronous-exception handler here |
| oe\_uint64\_t fsbase | address | Virtual address of segment page |
| oe\_uint64\_t gsbase | address | Virtual address of segment page |
| oe\_uint32\_t fslimit | 0xFFFFFFFF |  |
| oe\_uint32\_t gslimit | 0xFFFFFFFF |  |

## Set-Aside Areas

The SGX hardware uses the set-aside areas (**SSAs**) when an exception (or fault) occurs. The hardware stores the current state in the next available SSA (**TCS.cssa** + 1) and then exits to the host, invoking the host’s AEP (Asynchronous Exception Procedure). The host may opt to resume enclave execution by executing the ERESUME instruction. If so, then OE\_Main() is invoked again, this time with **TCS.cssa** greater than zero (indicating an exception).

## Segment Page

The SGX EENTER instruction initializes the GS segment register (X86-64bit) to refer the segment page. The segment page is initially zero-filled. When the enclave is entered for the first time, it is initialized with the **thread data (OE\_ThreadData)** structure, which defines the following key fields.

|  |  |
| --- | --- |
| **OE\_ThreadData Fields** | **Description** |
| self\_addr | Self-pointer to thread data structure |
| last\_sp | The last stack pointer (used for enclave reentry) |
| initialize | Whether the enclave has been initialized |
| depth | The depth of the ECALL stack (for ECALL nesting) |
| host registers | Fields for saving and restoring host registers |
| callsites | A list of structures for resuming after an OCALL |
| simulate | Whether simulation mode is active |

The address of this structure serves as the current thread identifier, returned by **OE\_ThreadSelf()**.

## Thread-Specific Data Page

OE defines one thread-specific data page per thread, for a total of 512 8-bytes slots (4096 / 8). These slots are managed by the following functions.

|  |  |
| --- | --- |
| **Function** | **Description** |
| OE\_ThreadKeyCreate() | Creates a thread-specific data key |
| OE\_ThreadKeyDelete() | Delete a thread-specific data key |
| OE\_ThreadSetSpecific() | Sets thread-specific data for the given key |
| OE\_ThreadGetSpecific() | Gets thread-specific data for the given key |

Although the page itself is limited to 512, the implementation can be extended later to support more slots by using indirection to additional slot pages.

# Exception Handling

## Hardware exceptions

When a hardware exception happens inside an enclave, it will trigger an **Asynchronous Enclave Exit (AEX)**: the CPU will save the processor state inside the enclave in the thread’s current SSA frame, then set the processor state to a synthetic state to prevent any leakage of secrets and exit the enclave. To the host, it looks like the exception is thrown in the **Asynchronous Exit Pointer (AEP)**. The AEP is a pointer to a fixed handler function in the host, which is registered as part of the EENTER instruction for each call into the enclave.

To provide the enclave application an opportunity to handle the hardware exception inside the enclave, we use a two-pass exception handling mechanism as illustrated below:

**Host**

**Enclave**

OE\_ECALL uses EENTER

instruction that registers

the AEP

**CPU**

Enclave function executes

until hardware exception

triggers AEX

AEX causes exception in AEP

that is passed to the signal

handler registered in the host

Signal handler calls the first

pass exception dispatcher

via another EENTER

instruction that registers the

signal handler as the AEP

First pass exception dispatcher

saves minimal exception context

inside enclave and calls EEXIT

Signal handler considers the

exception handled and returns

to AEP which can ERESUME

Second pass exception

dispatcher is called and

calls each vectored

exception handler that is

registered in the enclave

## Vectored exception handling

Once the second pass exception handler is invoked inside the enclave, Open Enclave provides a vectored exception handling model for developers to register their own exception handling methods.

* Each exception handler is called sequentially until the exception is handled by one of them returning ENCLAVE\_EXCEPTION\_CONTINUE.
* If handled, execution resumes from the context where the original exception occurred.
* If unhandled, the enclave is aborted.

To register or unregister vectored exception handlers, the Open Enclave SDK provides the following methods:

void\* OE\_AddVectoredExceptionHandler(

uint64\_t isFirstHandler,

POE\_VECTORED\_EXCEPTION\_HANDLER vectoredHandler);

uint64\_t OE\_RemoveVectoredExceptionHandler(

void\* vectoredHandler);

# The Signing Tool

OE provides a signing tool for signing enclaves. This tool injects a signature section into the ELF-64 image file called **.oesig**. The content of this section is determined by three sources.

* Fields defined in the signing configure file
* The enclave’s measurement (MRENCLAVE)
* The enclave’s private key

The signature section is defined as follows.

typedef struct \_OE\_SignatureSection

{

oe\_uint64\_t magic;

OE\_EnclaveSettings settings;

SGX\_SigStruct sigstruct;

}

OE\_SignatureSection;

The **settings** field is defined by the following structure.

typedef struct \_OE\_EnclaveSettings

{

oe\_uint64\_t debug;

oe\_uint64\_t numHeapPages;

oe\_uint64\_t numStackPages;

oe\_uint64\_t numTCS;

}

OE\_EnclaveSettings;

All settings are derived directly from the signing configuration file.

The **sigstruct** field is an SGX-defined data structure, defined in **Section 38.13** of the following document (see SIGSTRUCT).

[Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 3D](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf)

The signing tool fills in this structure and passes it to the driver during enclave initialization.

## Example

This example shows how to sign an enclave. Three input files are needed:

* An enclave image file
* A signing configuration file
* A private key

Suppose that the enclave image file has already been created with the following name.

myenclave.so

Next, we define a signing configuration file called **myenclave.conf** with the following contents.

NumHeapPages=1024

NumStackPages=1024

NumTCS=2

Then, we generate a self-signed key with the following command.

$ openssl genrsa -out myenclave.key -3 3072

Finally, we are ready to sign the enclave file.

$ oesign myenclave.so myenclave.conf myenclave.key

Created myenclave.signed.so

To verify that myenclave.signed.so contains the signature section, use the following command.

$ readelf -S myenclave.signed.so | grep oesig

[25] .oesig PROGBITS 0000000000000000 00022150

This signature section will be used during enclave creation.

# The Debugger Extension

By default, debuggers like GDB do not understand the enclave model or context switches on ECALL/OCALL. To help the developer debug ECALL/OCALL transitions, Open Enclave includes a GDB debugger extension that consists of two parts: a ptrace library and a python extension.

## Ptrace Library

The ptrace library (oe\_ptrace.so) implements the customized **ptrace** and **waitpid** functions to get/set enclave registers and fix enclave breakpoints. This library will be preloaded into GDB when it is launched using the debugger/oe-gdb script.

## Python Extension

The GDB Python extension is loaded into GDB as a plugin. It allows GDB to load enclave symbols and debug enclaves with **stack stitching**. More details on the GDB Python extension model can be found at [https://sourceware.org/GDB/onlinedocs/GDB/Extending-GDB.html](https://sourceware.org/gdb/onlinedocs/gdb/Extending-GDB.html)

## Stack Stitching

Stack stitching is the process of constructing a single complete [call stack](https://en.wikipedia.org/wiki/Call_stack#Structure) across both enclave and host function calls. This is achieved by allowing the debugger to perform a stack walk that crosses seamlessly between the call stack maintained by the host thread context and the stack maintained by the enclave thread context.

First, consider the simple case where there is a single stack. Each unreturned function call is tracked by a stack frame in the call stack. A stack walk starts from the current stack frame and recursively finds the previous stack frame until no further stack frames are found.

To enable this stack walking, each stack frame needs to be set up with the information so that GDB can find the correct previous stack frame. A stack frame is usually identified by an address on the stack referred to as the **Canonical Frame Address (CFA)**. This is usually defined as the value of the stack pointer (rsp value) at the call site in the *previous* frame (see section 6.4 of the [DWARF spec](http://dwarfstd.org/doc/DWARF4.pdf) for more details).

In Open Enclave, the base pointer (rbp value) is saved onto the stack to facilitate stack walking:

* The return address is automatically pushed onto the stack when a new call is made.
* The previous base pointer value is pushed onto the stack after it.
* On a 64-bit platform, the CFA and return address are thus always at 16-bytes and 8-bytes higher than the base pointer, which can be walked as a chain given the saved values on the stack.

**…**

**Previous call**

**CFA (previous RSP)**

**…**

**…**

**Return address**

**Saved RBP**

**…**

**CFA - 16 (current RBP)**

**Higher addresses**

**Lower addresses**

**New call frame**

In the debugger output below, you can see an example where:

* CFA for the call frame is at 0x7fffffffdc20
* Saved rip (return address) is 0x7fffffffdc18 (CFA - 8)
* Saved rbp is 0x7fffffffdc10 (CFA - 16)
* Previous CFA is at 0x7fffffffdc70

(GDB) info frame  
Stack level 0, frame at 0x7fffffffdc20:  
rip = 0x40b0d5 in HandlePrint (/home/lei/repos/openenclave/host/ocalls.c:40); saved rip = 0x403c56  
called by frame at 0x7fffffffdc70  
source language c.  
Arglist at 0x7fffffffdc10, args: argIn=140737488346504  
Locals at 0x7fffffffdc10, Previous frame's sp is 0x7fffffffdc20  
Saved registers:  
rbp at 0x7fffffffdc10, rip at 0x7fffffffdc18

Note that it is possible to use a register other than rbp for this purpose. Open Enclave simply uses [Call Frame Information (CFI) directives](https://sourceware.org/binutils/docs-2.16/as/CFI-directives.html) in the enclave entry and exit assembly code to set the rules for computing the CFA relative to rbp as described above. By controlling the saved rbp values, it can thus direct the stack walk back and forth between the host and enclave stacks as needed.

However, this is not sufficient for displaying stitched stacks in GDB. Even when the saved RBP values correctly point back and forth between the two stacks, GDB will indicate that the stack is corrupted when doing a naïve stack walk. This is because it does a sanity check that stack addresses are continuous (that they monotonically decrease as stack grows) but the host and enclave stacks occupy discontinuous address ranges that violate this assertion.

For stack stitching to work then, Open Enclave takes advantage of support for [Split Stacks in GCC](https://gcc.gnu.org/wiki/SplitStacks). Although Open Enclave does not dynamically grow stacks, it takes advantage of the fact that GDB uses the \_\_morestack function name as a marker for disabling the sanity check for monotonic stack addresses. The assembly functions for both OE\_Enter and OE\_Exit are aliased to \_\_morestack function name for this reason.

For more details on the mechanisms of this, refer to GDB\binutils-GDB\GDB\frame.c in the [GDB source code](https://www.gnu.org/software/gdb/).

# The IDL Generator [Experimental]

OE provides an IDL generator called **oegen** for producing type-safe wrappers for performing ECALLs and OCALLs. Here are the major design decisions.

* The oegen utility is an add-on rather than an integral part of the OE intrinsics. This allows users to utilize alternative IDL technologies.
* The enclave stubs automatically copy parameters to and from host memory.
* All stubs are generated for the C language.
* The IDL language supports definition of structures and functions.
* The generated stubs perform limited buffer overrun checking.
* The utility generates general-purpose metadata that could be used for other purposes.
* The utility provides constraints modifiers on functions and parameters.

## Supported Data Types

The oegen tool supports the following data types.

|  |  |
| --- | --- |
| **IDL type** | **Generated type** |
| char | char |
| short | Short |
| int | Int |
| long | long |
| int8 | oe\_int8\_t |
| uint8 | oe\_uint8\_t |
| int16 | oe\_int16\_t |
| uint16 | oe\_uint16\_t |
| int32 | oe\_int32\_t |
| uint32 | oe\_uint32\_t |
| int64 | oe\_int64\_t |
| uint64 | oe\_uint64\_t |
| float | float |
| double | Double |
| bool | oe\_bool |
| size\_t | oe\_size\_t |
| ssize\_t | oe\_ssize\_t |
| wchar\_t | oe\_wchar\_t |
| void | Void |
| signed char | signed char |
| signed short | signed short |
| signed int | signed int |
| signed long | signed long |
| unsigned char | unsigned char |
| unsigned short | unsigned short |
| unsigned int | unsigned int |
| unsigned long | unsigned long |

## Function and parameter constraints

|  |  |  |
| --- | --- | --- |
| **Constraint** | **Scope** | **Meaning** |
| ecall | function | This function is an ECALL |
| ocall | function | This function is an OCALL |
| in | parameter | This is an input parameter |
| out | parameter | This is an output parameter |
| inout | parameter | This is an input and an output parameter |
| ref | parameter | This is a reference parameter (pointer to pointer) |
| unchecked | parameter | This parameter is unchecked (passed through as is) |
| count(arg) | parameter | The cardinality of this array is given by arg, where arg is a constant or the name of another parameter |
| one | parameter | The cardinality of this array is one |
| string | parameter | The parameter is a C string |

## Example 1

function [ocall] void WriteFile(

[inout, one] struct File\* file,

[in, count=size] const void\* data,

[in] size\_t size);

## Example 2

struct Person

{

[string] const char\* first;

[string] const char\* last;

uint16 age;

};

function [ecall] int CreatePerson(

[in, one] struct Person\* person);