Intel SGX Enclave Support in Windows 10 Fall Update (Threshold 2)

Technical Whitepaper

Revision 1.0 – November 8th, 2015

Authored by Alex Ionescu (@aionescu)
http://www.alex-ionescu.com
Winsider Seminars & Solutions Inc.

Table of Contents

Introduction

One of the most exciting features being introduced by Intel in the latest Skylake processors is a new security feature called Software Guard Extensions (SGX), often referred to as Secure Enclaves. This is similar to the technology ARM baked into their processors under the TrustZone (ARM, 2015) brand, with some improvements.

Due to space constraints, this whitepaper will not cover hardware-specific SGX details – it is definitely recommended that you read the official Intel SGX Reference Documentation Manual (Intel, 2014), which will be referred to quite frequently. In some cases, this paper may embed certain tables from the Intel SGX documentation in order to facilitate cross-referencing, but this is not a replacement for reading the official documentation.

At this point, it should also be stressed, that if you have already purchased a Skylake processor, you likely did *not* get an SGX-compatible part. In an incredible, pretty non-talked about snafu, only Skylake parts after October 26th, at the *earliest* will have SGX enabled, and you're likely to be able to buy them only after November 30th.

These parts have a different "S-part" number, but otherwise have the same CPUID, Stepping, Model Numbers, etc. Manufacturers are free to interchangeably sell you whichever part they have in stock, without any recourse to you, the buyer. You can read Intel's Product Change Notification (Intel, 2015) to see which parts you should attempt to buy, if your vendor will let you verify this information before buying.

Worse, even if you end up getting a compatible S-part processor, your BIOS must still be updated to set the appropriate enable bit in the `IA32_FEATURE_CONTROL` (`0000`003Ah) MSR, in this case the **`SGX Global Enable Bit`** (18) and your BIOS must additionally reserve a region of RAM for SGX use. At this moment, the author is aware of only the following machines which appear to have the required support:

- Inspiron 11 i3153
- Inspiron 11 i3158
- Inspiron 13 i7353
- Inspiron 13 i7359
- Inspiron 15 i7568

Note that when walking into a local Microsoft Store and examining one of the released Surface Pro 4 machines, no SGX support was indicated in CPUID. It's possible that newer models, such as the upcoming i7 releases, may support this.

Boot-Time Enclave Support Initialization

Once a compatible BIOS has initialized SGX support by enabling the feature in the IA32_FEATURE_CONTROL MSR, and configured the PRMRR MSRs to store the Processor Reserved Memory (PRM) descriptors which still store SGX-related memory, system software must next initialize its own support for SGX. In Windows, this is done both by the boot loader, as well as the kernel's initialization functions.

Boot Loader Enclave Support

During boot, Winload (the Windows Boot Loader) will call *OslEnumerateEnclavePageRegions*, which first checks if CPUID 0000`0012h is supported. Next, it uses CPUID 0000`0007h to obtain the list of CPU feature flags, and checks if bit 2 is set (SGX). If this is the case, it then actually issues CPUID 0000`0012h with sub-function 2. This corresponds to the Intel SGX Resource Enumeration protocol described in the Intel SGX Manual, and returns whether or not an Enclave Page Cache (EPC) is present. If so, the physical address and size of the EPC section is described.

By calling *BlMmAddEnclavePageRange*, the boot loader adds a descriptor to the MemoryDescriptorListHead structure that is stored in the well-known LOADER_PARAMETER_BLOCK that is used to transfer information between the boot loader and the kernel. This descriptor is built with a type value of **LoaderEnclaveMemory**. The loader then attempts issuing sub-function 3, and so on, until the CPU returns an indicator specifying that no EPC section exists for this index.

Early Boot Memory Manager Enclave Support

In order for the MiEnclaveRegions AVL tree to be populated, the memory manager calls *MiCreateEnclaveRegions* during Phase 1 initialization. This receives the LOADER_PARAMETER_BLOCK structure that is well-known to those studying boot loader internals, and parses its MemoryDescriptorListHead, looking for descriptors of type **LoaderEnclaveMemory**. As the routine does, it builds the AVL tree (allowing quick lookups to see if a region of memory is actually part of an enclave – which we'll see will be needed in some cases), and it also constructs the descriptors for each physical page that is owned by the hardware enclave.

The Windows memory manager stores physical page information in an array called the PFN Database, which is made up of structures called PFN Entry (MMPFN), and enclave pages are no different. Avid Windows Internals readers may remember that these PFN entries are stored in a variety of lists, such as the Free Page List, or the Modified Page List. What about enclave pages?

In Windows 10 Update 2, the kernel adds a new Enclave Page List, and a special flag passed to *MiInsertPageInFreeOrZeroedList* enables functionality to utilize this new list. Internally, however, since the memory manager has actually run out of list identifiers, the kernel actually identifies these pages as being "bad" pages currently suffering an in-page error. This actually makes sense – the memory manager knows never to use bad pages, so calling enclave pages "bad" is another way to keep them at bay.

At this point, the *MiEnclaveRegions* AVL tree describes the physical page regions, while the Enclave Page List describes each page in the PFN database.

Enclave Construction

The new Windows 10 update provides an easy-to-use set of APIs that roughly map to the Intel SGX manual's described mechanisms for enclave construction, and which will ultimately issue the supervisor-mode (Ring 0) instructions which are required to perform the operations. These new APIs are present in the Enclave API Set documented in `enclaveapi.h` and implemented in `Kernelbase.dll` on supported systems. Let's begin with the APIs for creating an enclave.

Checking for Enclave API Hardware Support

The first step to using the Enclave API is to first figure out if the hardware supports enclaves. The *IsEnclaveTypeSupported* API provides this functionality, and accepts as input the enclave type you wish to verify hardware support for:

```
BOOL
WINAPI
IsEnclaveTypeSupported (
    _In_ DWORD flEnclaveType
    );
```

This ultimately results in the API reading the `EnclaveFeatureMask` in the KUSER_SHARED_DATA structure that Windows Internals folks will know is located at the hardcoded address 0x7FFE0000. As of now, the only enclave type supported is the SGX enclave (**ENCLAVE_TYPE_SGX)**, so requesting any other type of enclave will directly fail.

In kernel-land, this determination is made by *KiSetFeatureBits*, which first checks the `IA32_FEATURE_ENABLE` MSR (`0000_03Ah`) for the SGX Global Enable Bit (18), which indicates support SGX has been enabled the BIOS. It then issues CPUID `0000_0012h` with sub-level 0 (SGX Capability Query) which, as per the Intel CPU documentation, will return 0 for SGX1 support, or 3 for SGX2 support.

If SGX1 or higher is supported, the kernel writes 2 for the `EnclaveFeatureMask` (since this is a bitmask, this means that Enclave Type 1, i.e.: **ENCLAVE_TYPE_SGX** is supported). Finally, the **KF_SGX** (`0x0000010000000000`) is set in the KPRCB `FeatureBits` field, which is later copied into *KeFeatureBits*.

Creating an Enclave

Creating an enclave is much like allocating virtual memory, as you can see from the *CreateEnclave* prototype below:

```
PVOID
WINAPI
CreateEnclave (
    _In_      HANDLE  hProcess,
    _In_opt_  LPVOID  lpAddress,
    _In_      SIZE_T  dwSize,
    _In_      SIZE_T  dwInitialCommittment,
    _In_      DWORD   flEnclaveType,
```

```
    _In_      LPCVOID lpEnclaveInformation,
    _In_      DWORD   dwInfoLength,
    _Out_opt_ LPDWORD lpEnclaveError
    );
```

Just like any other Win32 API, this ultimately results in the following system call:

```
_Must_inspect_result_
NTSYSAPI
NTSTATUS
NTAPI
NtCreateEnclave (
    _In_      HANDLE      ProcessHandle,
    _Inout_   PVOID      *BaseAddress,
    _In_      ULONG_PTR   ZeroBits,
    _In_      SIZE_T      Size,
    _In_      SIZE_T      InitialCommitment,
    _In_      ULONG       EnclaveType,
    _In_reads_bytes_(EnclaveInformationLength)
              CONST VOID *EnclaveInformation,
    _In_      ULONG       EnclaveInformationLength,
    _Out_opt_ PULONG      EnclaveError
    );
```

This pretty much matches the Win32 counterpart, with the addition of the `ZeroBits` parameter, which should be familiar to those that have looked at the *Nt* side of the virtual memory allocation routines

Because enclaves work on **`MEM_RESERVE/MEM_COMMIT`** semantics, just like standard virtual memory allocations, the third and fourth parameters basically determine how much the memory manager should reserve, and which part of that it should commit.

And again, similar to other memory APIs, the second parameter can either allow you to pre-determine the address of the enclave, or allow the kernel to select it for you. The fifth parameter, on the other hand, simply maps to the same Enclave Type definitions we saw above – i.e.: **`ENCLAVE_TYPE_SGX`** must be used at this point.

The next two parameters are where things get harder. Each of the supported enclave types has its own information structure. For the current SGX implementation, you must use `ENCLAVE_CREATE_INFO_SGX` here, which Microsoft documents as follows:

```
typedef struct _ENCLAVE_CREATE_INFO_SGX
{
    UCHAR Secs[4096];
} ENCLAVE_CREATE_INFO_SGX, *PENCLAVE_CREATE_INFO_SGX;
```

This probably looks meaningless to you unless you've read the Intel SGX documentation referenced earlier. Indeed, Microsoft expects developers to have their own header/definition for this Secs field. In fact, this corresponds to the SGX Enclave Control Structure, or SECS, and you would cast it to this buffer as needed. Here's what Skylake expects you to send:

Table 2-2. Layout of SGX Enclave Control Structure (SECS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
SIZE	0	8	Size of enclave in bytes; must be power of 2
BASEADDR	8	8	Enclave Base Linear Address must be naturally aligned to size
SSAFRAMESIZE	16	4	Size of one SSA frame in pages (including XSAVE, pad, GPR, and condition-ally MISC).
MISCSELECT	20	4	Bit vector specifying which extended features are saved to the MISC region of the SSA frame when an AEX occurs
RESERVED	24	24	
ATTRIBUTES	48	16	Attributes of the Enclave, see Table 2-3
MRENCLAVE	64	32	Measurement Register of enclave build process. See SIGSTRUCT for proper format.
RESERVED	96	32	
MRSIGNER	128	32	Measurement Register extended with the public key that verified the enclave. See SIGSTRUCT for proper format.
RESERVED	160	96	
ISVPRODID	256	2	Product ID of enclave
ISVSVN	258	2	Security version number (SVN) of the enclave
EID	Implementation dependent	8	Enclave Identifier
PADDING	Implementation dependent	352	Padding pattern from the Signature (used for key derivation strings)
RESERVED	260	3836	Includes EID, other non-zero reserved field and must-be-zero fields

Since the structure is pre-defined as occupying exactly a page (4KB), the 7th parameter must be exactly `sizeof(ENCLAVE_CREATE_INFO_SGX).` Finally, you'll get the error code as the 8th parameter, which differentiates SGX-specific error codes from OS error codes.

One important thing that the kernel does before performing any additional steps however, is to check if there are any enclave regions that hardware reported to begin with. You see, since we're asking the CPU to store data in a protected region of memory, such a protected region must exist to begin with!

The kernel tracks this by using the *MiEnclaveRegions* AVL tree, and at this point simplify verifies that there is a root (meaning at least one region was detected). In the earlier section, we saw how the kernel populates this AVL tree.

The *NtCreateEnclave* system call, after validating parameters, allocates a kernel copy of the SECS that was passed in, in order to avoid race conditions during creation, and then takes a reference to the process that was passed in, attaching to its virtual address space with *KeStackAttachProcess* if needed. Interestingly, as per the API parameters, this allows cross-process enclave creation.

Once the EPROCESS object is obtained, and the kernel SECS copy is made, *MiCreateEnclave* will take over the rest of the enclave creation process, signaling that enclaves continue to be the purview of the memory manager.

The first thing *MiCreateEnclave* does is allocate the AWE information structure that Address Widowing Extension APIs also utilize. This is because similar to the AWE functionality, an enclave allows a user-mode application to directly have access over physical pages (that is to say, the physical pages that are EPC pages based on the detection described earlier). Anytime a user-mode application has such direct control over physical pages, AWE data structures and locks must be used. This data structure is stored in the `AweInfo` field of `EPROCESS`.

Next, *MiCreateEnclave* will call *MiAllocateEnclaveVad*. Again, this is not surprising for those familiar with Windows memory management – all user-mode virtual memory allocations require the usage of a Virtual Address Descriptor, or VAD, to describe them.

This type of VAD is a little different than the usual Short or Long VADs described in Windows Internals – it additional contains a system PTE (whose use we'll describe later) as well as the array of PFN entries that correspond to the underlying EPC pages associated with the enclave (which, at this point, is empty). We call this VAD an MMVAD_ENCLAVE and surmise its data structure to be defined as follows:

```
typedef struct _MMVAD_ENCLAVE
{
    MMVAD_SHORT Core;
    PMMPTE SecsSystemPte;
    PMMPFN EnclavePfnArray;
    ULONGLONG EnclavePfnCount;
    struct
    {
        ULONG Initialized:1;
        ULONG Debuggable:1;
    };
} MMVAD_ENCLAVE, *PMMVAD_ENCLAVE;
```

Additionally, Enclave VADs are marked as VadAwe due to the reasons explained above. However, a related bit in the MMVAD structure, `u.VadFlags.Enclave`, is also set, allowing differentiation from "true" AWE memory. Finally, as part of VAD allocation, this is where the user-mode address for the enclave memory will be chosen, which utilizes the same anonymous-memory ASLR improvements offered by Windows 8.

Next, *MiCreateEnclave* will check if an initial commitment was specified, as per the API documentation. For now, let's assume this isn't the case – we'll shortly see how to commit enclave memory and what the effects are.

The last step of the memory manager's role is to acquire an enclave page regardless of the enclave size or initial commitment. This is because, as per Intel's SGX documentation, all enclaves require at least a one-page "control structure" to be associated with them. *MiGetEnclavePage* is used to obtain the required allocation. This function simply scans the Enclave Page List that was described earlier, and extracts one page as needed.

With the page returned, remember the system PTE we saw *MiAllocateEnclaveVad* allocate? That PTE will correspond to the system's kernel virtual address that will map the enclave page that was just obtained.

MiInitializeEnclavePfn is used to setup the related MMPFN structure, which is now marked as Modified and ActiveAndValid.

There are no actual bits at this point that would help you differentiate this enclave PFN from any other active region of memory (such as non-paged pool). As you can probably imagine, this is where the *MiEnclaveRegions* AVL tree comes into play, and *MI_PFN_IS_ENCLAVE* is a function that the kernel will use whenever it needs to check if a PFN is indeed describing an EPC region.

With the PFN initialized, the system PTE is now converted to a final global kernel PTE, and its resulting virtual address is computed (you can always convert a virtual address into a PTE, and vice-versa, on Windows, since all PTEs are mapped in a flat array).

The memory manager has now completed its part of the work! What happens next is a call to *KeCreateEnclave*, which will now do the low-level kernel enclave creation steps, including communication with the actual SGX hardware implementation.

The memory manager passes the virtual address of the control area (from the PTE above), the size and virtual address of the enclave memory in user-mode, and a flag specifying whether or not this process is a WoW64 process, which will be needed shortly.

KeCreateEnclave now creates the SGX-required PAGEINFO structure. Again, you should refer to the Intel SGX Manual for more information, but suffice it to say that this structure is the implicit input structure to most of Intel's SGX instructions. Before doing so however, it parses the SECS structure and performs a few validation checks.

First, if *KeFeatureBits* does not have the **KF_SGX** bit set (meaning the kernel did not detect SGX support at boot), the function fails. You'll see this pattern repeated many times, and it has to do with the fact that *Ke*'s job is to potentially support multiple types of hardware enclave features, so in the future it may use the feature bits to decide which hardware implementation to use (or to determine SGX2 vs SGX1 support.

Next, you'll remember that the documented API provides the ability to input the size of the enclave. Well, so does the SECS structure, and *KeCreateEnclave* will validate that these are the same value. Next, the Attributes field of the SECS will be verified – specifically, if this is a WoW64 process, then the MODE64BIT attribute will not be allowed.

Conversely, if this is a 64-bit process, the MODE64BIT attribute is enforced. If you were thinking of using 64-bit enclaves in a WoW64 process for stealth cross-architecture code execution, think again.

Next, since the caller may have requested the typical behavior of having the kernel select the base address for the allocation, they would not have been able to fill out the pre-requisite BASEADDR field of the SECS, so *KeCreateEnclave* takes care of filling out this data.

Finally, if the DEBUG attribute has been set in the SECS, the function sets a special flag on output, which *MiCreateEnclave* will consume. At this point, the validation and completion of the SECS has been completed, and the PAGEINFO structure will be created.

You can see its structure, for reference, below.

Table 2-14. Layout of PAGEINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
LINADDR	0	8	Enclave linear address
SRCPGE	8	8	Effective address of the page where page contents are located
SECINFO/PCMD	16	8	Effective address of the SECINFO or PCMD (for ELDU, ELDB, EWB) structure for the page
SECS	24	8	Effective address of EPC slot that currently contains a copy of the SECS

The job of the PAGEINFO structure is to describe the SECS page – which will be copied into the EPC, and take up the kernel address we saw earlier (the one backed by a system PTE).

Note that the Intel documentation specifies that the SECS and LINADDR fields should be NULL, as these will be the destination addresses that the SGX hardware will fill out on its own. SRCPGE, on the other hand, is initialized to the address of the SECS, while SECINFO is initialized to another structure that *KeCreateEnclave* will fill out. As per the Intel manual, SECINFO represents a series of flags that are used to configure the security permissions of an EPC page – similar to the PTE for a regular page. Here they are below:

Table 2-16. Layout of SECINFO.FLAGS Field

Field	Bit Position	Description
R	0	If 1 indicates that the page can be read from inside the enclave; otherwise the page cannot be read from inside the enclave
W	1	If 1 indicates that the page can be written from inside the enclave; otherwise the page cannot be written from inside the enclave
X	2	If 1 indicates that the page can be executed from inside the enclave; otherwise the page cannot be executed from inside the enclave
PENDING	3	If 1 indicates that the page is in the PENDING state; otherwise the page is not in the PENDING state.
MODIFIED	4	If 1 indicates that the page is in the MODIFIED state; otherwise the page is not in the MODIFIED state.
RESERVED	7:5	Must be zero
PAGE_TYPE	15:8	The type of page that the SECINFO is associated with
RESERVED	63:16	Must be zero

Since the copy of the SECS in the enclave needs to be protected from code running in the enclave, both the R, W, and X fields are set to 0. The pending and modified bits are as well, since that state is not currently valid. Finally, the PAGE_TYPE is set to PT_SECS (0), which specifies that this is a SECS page. In other words, all of the fields are zero!

At this point, *KeCreateEnclave* has created all the pre-requisite structures to describe the SECS and its underlying EPC page, which can then be consumed by the SGX hardware to create the enclave. This is done by using the new **ENCLS** instruction offered by SGX. **ENCLS** in an interesting mnemonic – it is a single instruction with a verity of leaf functions that are specified in the **EAX** register, somewhat similarly to CPUID.

Each leaf function has its own name, and uses the **RBX**, **RCX**, and **RDX** registers in specific ways to store implicit data structures. The kernel now exposes a *KiEncls* function, which issues the **ENCLS** instruction, and moves the input arguments into the appropriate registers:

```
VOID
KiEncls (
    _In_ ULONG LeafFunction,
```

```
_In_ PVOID Parameter1,
_In_ PVOID Parameter2,
_In_ PVOID Parameter3
);
```

Here are the currently defined lead functions supported by SGX1. We will see many of these throughout this whitepaper, and describe their use by the Windows kernel.

Table 1-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an Enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EEXIT]	Exit an Enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDBGRD]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Re-enter an Enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		
ENCLS[EREMOVE]	Remove a page from EPC		
ENCLS[ETRACK]	Activate EBLOCK checks		
ENCLS[EWB]	Write back/invalidate an EPC page		

In order to create the enclave, *KeCreateEnclave* needs to use the **ECREATE** (0) leaf function, which has the following inputs:

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

As you can imagine, **RBX** will contain the PAGEINFO structure that was create above. **RCX**, on the other hand, is the destination of the SECS page – again, that's going to be the kernel address backed by the system PTE from earlier. Once this instruction returns, *KeCreateEnclave* finally returns back to *MiCreateEnclave*.

The last thing this function does is to read any output flags – such as the one describing that this is a debuggable enclave, based on the DEBUG attribute we saw *KeCreateEnclave* verifying earlier. If the DEBUG attribute is set, a Debug flag will be set in the Enclave VAD, whose use we'll understand later. Additionally, the NumberOfDebugEnclaves field in the MI_USER_VA_INFO sub-structure part of the current process' working set (MMWSL) structure is incremented (interestingly, no such field exists for regular enclaves – only debuggable enclaves are accounted).

MiCreateEnclave now returns back to *NtCreateEnclave*, which returns back to the user, with the base address of the enclave that was allocated, or any errors if anything in the above failed in any way. The hard part is done – now it's time to load data into the enclave!

Loading Data into an Enclave

Now that a secured region of memory has been associated with your process, the next step is to actually load code or data into the enclave. This can be done in one of two ways. First, if the enclave created with *CreateEnclave* doesn't yet have any committed enclave pages, they must first be obtained, which will result in zeroed out memory being added to the enclave, which can then be filled with non-zero memory from outside the enclave. Otherwise, if an initial pre-committed initialization size was passed in, then the enclave's pages can directly be filled in with non-zero memory from outside of the enclave. We will see how both of these operations are done.

Committing Enclave Memory

Because enclave memory is described by a VAD, many of the traditional memory management APIs will function, at least partly, on this memory as well. For example, calling *VirtualAlloc* (i.e.: *NtAllocateVirtualMemory*) on such an address with the **MEM_COMMIT** flag will result in *MiCommitEnclavePages* being called, which will validate that the protection mask for the new pages is compatible (i.e.: a combination of Read, Write, and/or Execute, without any special caching or write combining flags), and then call *MiAddPagesToEnclave*, passing a pointer to the Enclave VAD associated with the address range, the protection mask that was specified to *VirtualAlloc*, and the PTE addresses that correspond to the virtual address range being committed.

MiAddPagesToEnclave will first check if the Enclave VAD has any existing EPC pages associated with it, and if there's enough of them to satisfy the commit. If not, *MiReserveEnclavePages* will be called to obtain a sufficient amount.

MiReserveEnclavePages will look at the current Enclave Page List and count the total – if there aren't enough physical EPC pages provided by the processor (based on the information we saw is obtained at boot), the function will fail. Otherwise, it will call *MiGetEnclavePage* in a loop for the required amount of pages, which is a function we encountered earlier.

For each PFN entry that is retrieved, it will be linked into the PFN array in the Enclave VAD by using the `u1.Next` pointer in `MMPFN`. Essentially, this means that once an enclave PFN is removed from the Enclave Page List and put into an active state, the Enclave VAD acts as the list of active enclave PFNs.

Back in *MiAddPagesToEnclave*, the memory manager reserves another system PTE, and maps it to one of the pool of zeroed out pages that it maintains. Next, it computes the appropriate `SECINFO` flags for the page (recall that `SECINFO` is analogous to a PTE for an EPC, and were shown on Table 2-16).

The default flags that are set correspond to `PT_REG | R`, meaning a regular read-only page. If **PAGE_READWRITE** (or another mask containing write access) was specified, then `PT_REG | R | W` is used, and if **PAGE_EXECUTE** (or another mask containing execute access) was specified, then `PT_REG | R | X` is used. Finally, if the Enclave VAD flags specify that the enclave has already been initialized (through *InitializeEnclave*, which we have not yet seen), then a special flag is set, which will be validated later.

Once the right attributes have been computed, *MiAddPagesToEnclave* must reserve the appropriate number of system PTEs to hold the paging information for each of the EPC pages that will be required, and it does so by creating a PTE Copy List with *MiCreatePteCopyList*, looping around *MiReservePtes*. If enough system PTEs exist, the operation can continue.

Continuing on, *MiAddPagesToEnclave* will sit in a loop, calling *MiGetPageForEnclave,* which at this point will get one of the EPC pages that the Enclave VAD now has in its PFN array thanks to the call to *MiReserveEnclavePages* we saw earlier. Then it will call *MiGetPteFromCopyList* to get one of the system PTEs it has pre-reserved a moment ago. With this information in hand, it can now call into *KeAddEnclavePage*, which will perform the low-level SGX hardware work to add this page.

KeAddEnclavePage once again checks *KeFeatureBits* to make sure SGX is supported on this platform. It then takes as input the source address of the data to add into the enclave (which corresponds to the zeroed-out system PTE the function first requested), the destination address of where this data should be copied into (which is one of the system PTEs it just requested above), and finally the enclave virtual address that maps to this data, which is based on the input PTE ranges that *MiAddPagesToEnclave* received based on the virtual addresses that *VirtualAlloc* is currently being called on with **MEM_COMMIT**. Finally, it receives the computed page attributes that will be used in the SECINFO structure.

Remember that special flag *MiAddPagesToEnclave* sets if the enclave is already initialized? Well, *KeAddEnclavePage* checks for it, and immediately fails with **STATUS_CONFLICTING_ADDRESSES** if it's set. In other words, the SGX implementation does not allow adding EPC pages to an initialized enclave, and the *Ke* layer enforces this. You might ask why the *Mm* layer even bothered with the flag to start with. As per the usual rules of the NT kernel's portability and architectural layers, *Ke* refers to the low-level hardware-specific implementation of the kernel. It's here that SGX-specific rules are obeyed.

Mm, on the other hand, provides enclave functionality on top of any underlying CPU feature that supports it – and perhaps other enclave technologies in the future will not have this requirement. In fact, SGX2, which Windows does not yet support, actually includes an **EAUG** (0Dh) leaf function that allows adding pages to initialized enclaves.

KeAddEnclavePage now builds a PAGEINFO structure, which you familiarized yourself with already. In this case, since the enclave already exists, the LINADDR and SECS addresses must be filled in. But the kernel has this data – LINADDR corresponds to the user-mode enclave address that is being committed, and SECS is known because the Enclave VAD stores the system PTE we saw was reserved earlier in *CreateEnclave*'s logic.

SRCPGE in this case is set to the zeroed-out system PTE we mentioned above, and we finally have the SECINFO structure, which contains the EPC page protection attributes, based on the page protection flags. We already saw how **PAGE_READWRITE** and **PAGE_EXECUTE** were handled, but what about **PAGE_ENCLAVE_THREAD_CONTROL**? Simply put, this will be converted into the **PT_TCS** value in the PAGE_TYPE flag, instead of the usual **PT_REG** we saw earlier.

Now it's time to call into SGX with the **ENCLS** instruction offered by *KiEncls* again. However, leaf function **EADD** (1) will be used this time instead. Here are its required parameters:

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

The destination address, in this case, is one of the system PTEs that was reserved for this purpose, part of the PTE Copy List we described above, which maps to one of the EPC page PFN entries. We are not quite done yet, however. One of the features of SGX is its ability to host an internal TPM, regardless of the

external TPM chip your machine may have come with. This avoids potential tampering on the bus or with the external chip. As such, the SGX enclave can be measured, and any data new being added into it must be extended into the measurement.

The leaf function **EEXTEND** (6) provides this functionality, with the parameters below:

Instruction Operand Encoding

Op/En	EAX	RCX
IR	EEXTEND (In)	Effective address of a 256-byte chunk in the EPC (In)

KeAddEnclavePage will thus call *KiEncls* again, with each 256 byte chunk of data – it's important to remember that in this case, the data is all zeroes, as all we're doing is simply committing new pages into the enclave. However, you may recall **PAGE_ENCLAVE_UNVALIDATED** from earlier.

As one can guess, this causes *KeAddEnclavePage* to skip this process, as the whole point is to avoid measuring this data. This flag is probably worth using as part of the **MEM_COMMIT** call, as measuring zeroed data doesn't amount to much – unless you never plan to write into this data later on, and don't expect the enclave to either.

If *KeAddEnclavePage* returns successfully back into *MiAddPagesToEnclave*, the latter will now finalize the initialization of the EPC page's PFN entry, storing the final protection mask and destination address, and initialize the kernel PTE that maps to the user-mode address where the EPC page is mapped.

This completes the **MEM_COMMIT** for this particular page, and the loop continues, performing all these operations again, for every remaining PTE part of the user-mode address range that was specified.

Copying Existing Data into an Enclave

Once zeroed data is committed into the enclave, or if enclave pages had been pre-committed as part of creation, custom non-zeroed data can now be loaded into the enclave. This data should come from cleartext memory that is currently located outside of the enclave, and the Win32 API function *LoadEnclaveData*, shown below, is able to perform this operation.

```
BOOL
WINAPI
LoadEnclaveData (
    _In_      HANDLE  hProcess,
    _In_      LPVOID  lpAddress,
    _In_      LPCVOID lpBuffer,
    _In_      SIZE_T  nSize,
    _In_      DWORD   flProtect,
    _In_      LPCVOID lpPageInformation,
    _In_      DWORD   dwInfoLength,
    _Out_     PSIZE_T lpNumberOfBytesWritten,
    _Out_opt_ LPDWORD lpEnclaveError
    );
```

The user-mode API does nothing but convert this to the following system call:

```
_Must_inspect_result_
NTSYSAPI
NTSTATUS
NTAPI
NtLoadEnclaveData  (
    _In_      HANDLE      ProcessHandle,
    _In_      PVOID       BaseAddress,
    _In_reads_bytes_(BufferSize)
              CONST VOID *Buffer,
    _In_      SIZE_T      BufferSize,
    _In_      ULONG       Protect,
    _In_reads_bytes_(PageInformationLength)
              CONST VOID *PageInformation,
    _In_      ULONG       PageInformationLength,
    _Out_opt_ PSIZE_T     NumberOfBytesWritten,
    _Out_opt_ PULONG      EnclaveError
    );
```

Most of the parameters here should be self-documenting. In many ways, this is a combination of a *memcpy* call with some *VirtualAlloc* attributes. The second parameter here refers to an address inside the enclave, while the third is a regular buffer that's inside normal address space. Note that the size, in the fourth parameter, must be a page-multiple.

Next, the firth parameter refers to page protection attributes, which accepts the usual **PAGE_XXX** attributes from *VirtualAlloc*. It also, however, accepts **PAGE_ENCLAVE_THREAD_CONTROL**, which indicates that the memory is the TCS structure documented in the Intel SGX manual, as well as the **PAGE_ENCLAVE_UNVALIDATED** attribute, which indicates that the data inside this memory block should not be measured by the **EEXTEND** leaf function that SGX offers (SGX offers its own embedded TPM). The next two parameters regarding page information are actually not used for SGX purposes, and the final two should be self-descriptive.

NtLoadEnclaveData will validate the parameters based on MSDN documentation, such as checking the page alignment of the buffers and their sizes, as well as reject usage of the `PageInformation` parameters, which are not currently used. It will then obtain a pointer to the `EPROCESS` object for the handle, and finally call *MiCopyPagesIntoEnclave*, which will actually perform all of the work.

Before we go into those details, however, it's important to note that the target enclave address must have actually been committed first. Recall from the previous discussion that unless the `dwInitialCommitment` passed in to *CreateEnclave* was non-zero, the VAD currently merely describes a reserved, but not yet committed range. In order to be able to store enclave data into the range, it must first have been committed, by one of the two mechanisms we've already seen.

We return our analysis to *MiCopyPagesIntoEnclave*, which provides the bulk implementation of the *NtLoadEnclaveData* system call that provides the *LoadEnclaveData* Win32 API backend. The initial work

here, much like in the **MEM_COMMIT** situation we saw above, is to compute the appropriate internal enclave attributes that match to the various documented page protection flags, and to reject incorrect attributes with **STATUS_INVALID_PAGE_PROTECTION** (such as write-combining).

The function is also careful to respect one of the new Windows 8.1 process mitigation options – if "DisableDynamicCode" was enabled, then **PAGE_EXECUTE** enclaves will not be allowed. It's impressive that the kernel team thought about this – as it would've otherwise allowed for an interesting bypass through SGX.

Next, *MiCopyPagesIntoEnclave* calculates the required number of pages for the data, and interestingly caps this to 80 KB (in fact, the **MEM_COMMIT** path we saw earlier applies the same cap). This limitation is actually not documented on MSDN, and may be a limitation of how much data copying and PTE reservations the kernel may be willing to do at one time. It does not look like calling the function multiple times won't work – so keep that in mind if you wish to load larger amounts of enclave data. Just like in the **MEM_COMMIT** scenario, *MiCreatePteCopyList* is used to obtain the required System PTEs, and the Enclave VAD is then retrieved with *MiObtainReferencedVad*.

This VAD is then double-checked to ensure it's indeed an Enclave VAD, by checking that VadType is equal to VadAwe, and that the Enclave bit it set – which matches that we saw in *MiAllocateEnclaveVad* much earlier.

Next, if the Enclave VAD flags indicate that this enclave has already been initialized through *InitializeEnclave*, just like in the **MEM_COMMIT** scenario, a special flag is set. Then, if the source buffer address is not aligned on a page boundary, a kernel-mode paged pool buffer of 64KB is allocated, which will be used as a temporary copy buffer. If the data is aligned, on the other hand, this copy buffer is skipped.

Moving on, a 16 page copy loop is started (matching the 64KB copy buffer). As part of this loop, the user-mode data is copied into the kernel buffer if it was not aligned correctly. Otherwise, if the data was correctly aligned, but the buffer is located in user-mode, an MDL is built to probe and lock the user data in memory. Since we're dealing with a large buffer, this is more efficient than double-buffering (but as MDLs have page-alignment requirements, we had no choice in the other situation).

Once the MDL or copy buffer is filled in, *MiCopyPagesIntoEnclave* will attach to the target process (again, allowing for cross-process enclave construction), and another inner loop will start, calling *MiGetPageForEnclave* followed by *MiGetPteFromCopyList* for each required page, to finally pass in this data to *KeAddEnclavePage*.

This loop, and the operations within it, have already all been described in the **MEM_COMMIT** scenario above, with the only difference that instead of mapping in a zeroed-out system PTE as the source page, this loop will actually use one of the PTEs that corresponds to either the copy buffer or the MDL, so that actual initialized data is being copied into the enclave.

Once the inner and outer copy loops return, *MiCopyPagesIntoEnclave* returns back to *NtLoadEnclaveData*, which will return back to user-mode the number of bytes that were actually copied into the enclave. If the number is less than expected, the caller can then retry with the remainder of the data they wish to copy. At this point, the caller can initialize their enclave.

Enclave Lifetime Management

There are two final operations that can be performed on an enclave, now that it has been created. The first, and most critical, is to actually initialize the enclave, which will begin code execution in SGX mode, using all the accumulated data that has been specified so far. Related to initialization, once SGX execution has completed, the non-enclave caller must take care to destroy the enclave, in order to release all SGX resources.

Initializing an Enclave

The last step that a developer must take to utilize enclave technology is to initialize it, and the *InitializeEnclave* API, shown below, does exactly that:

```
BOOL
WINAPI
InitializeEnclave (
    _In_ HANDLE  hProcess,
    _In_ LPVOID  lpAddress,
    _In_ LPVOID  lpEnclaveInformation,
    _In_ DWORD   dwInfoLength,
    _In_ LPDWORD lpEnclaveError
    );
```

With this data, *InitializeEnclave* once again does nothing more than performing the following system call:

```
_Must_inspect_result_
NTSYSAPI
NTSTATUS
NTAPI
NtInitializeEnclave  (
    _In_      HANDLE      ProcessHandle,
    _In_      PVOID       BaseAddress,
    _In_reads_bytes_(EnclaveInformationLength)
              CONST VOID *EnclaveInformation,
    _In_      ULONG       EnclaveInformationLength,
    _Out_opt_ PULONG      EnclaveError
    );
```

The first two, and last parameters are hopefully nothing new by now. But there's a new enclave information structure, and associated size, that must be passed in. For SGX enclaves, this corresponds to the ENCLAVE_INIT_INFO_SGX structure, shown below:

```
typedef struct _ENCLAVE_INIT_INFO_SGX
{
    UCHAR SigStruct[1808];
    UCHAR Reserved1[240];
```

```
    UCHAR EInitToken[304];
    UCHAR Reserved2[744];
} ENCLAVE_INIT_INFO_SGX, *PENCLAVE_INIT_INFO_SGX;
```

The two key fields here correspond to `SIGSTRUCT` and `EINITTOKEN`, two more Intel SGX-specific structures that are described in the Intel SGX documentation, and which the kernel will consume as we'll shortly see.

As for *NtInitializeEnclave,* it acts almost identically to *NtCreateEnclave*. It validates that the parameters make sense, creates a kernel copy of the passed-in SGX enclave initialization information structure, obtains a pointer to the `EPROCESS` object for the passed-in process handle, and attaches to the process as needed. Once all that's done, it calls *MiInitializeEnclave*, which will actually perform all of the work.

MiInitializeEnclave is a fairly simple implementation. It calls into *MiObtainReferencedVad* on the base address pointer to make sure this is a valid allocation, and then just as we saw before, a check is made to ensure this is a `VadAwe`-type VAD, and that the Enclave bit is set in the flags. It also checks to make sure that the Initialized flag isn't yet set in the Enclave VAD, as double-initialization of an enclave is not allowed. Following these checks, *KeInitializeEnclave* is called, which will perform the SGX-specific initialization (there's nothing for the memory manager to do at this point, really).

KeInitializeEnclave receives the `SIGSTRUCT` and `EINITTOKEN` structures, plus the address of the `SECS` structure for this enclave (you should now know that this comes from the Enclave VAD's system PTE that was first created in *CreateEnclave*). With this information, is first makes sure that *KeFeatureBits* indicates SGX support is present, and then begins a 16-iteration loop of *KiEncls* calls, this time with leaf function **`EINIT`** (2). The following parameters are needed:

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

Note that the Intel SGX documentation states that certain asynchronous events may have occurred while the enclave was attempting initialization, such as an Interrupt or NMI. In this case, the operation will fail with **`SGX_UNMASKED_EVENT`**, a special error code that indicates the operation should be attempted again, which explains why *KeInitializeEnclave* will attempt this 16 times. If it still fails, **`STATUS_RETRY`** will be returned back up to the caller.

At this point, the kernel is satisfied, and *MiInitializeEnclave* will set the Initialized flag in the Enclave VAD. That's because the actual validation of the enclave, and all the cryptographic security around it is all handled in hardware. For those interested, the Intel SGX manual has detailed steps of what the **`EINIT`** instruction will perform on the CPU.

Destroying an Enclave

You may have noted that there is no matching *DestroyEnclave* or similarly named function in the Win32 API (nor in the system call API). Since Enclaves are seen as VADs by the memory manager (which is what allowed calling *VirtualAlloc* to **`MEM_COMMIT`** EPC pages into memory), then you can probably guess how destruction is done: by calling *VirtualFree*, i.e.: *NtFreeVirtualMemory*. A special check now exists inside of *MiDeleteVad* as the VAD is being destroyed, and if the `Enclave` bit is set on an AWE VAD, *MiDeleteEnclavePages* will be called instead.

MiDeleteEnclavePages will first clear out the PTE attributes, flush the TLB as needed, and then call *MiDeleteEnclavePage* for each of the pages described by the VAD. *MiDeleteEnclavePage* will then validate if the PTE is committed, and if so, call *KeRemoveEnclavePage* with the enclave address. Inside of *KeRemoveEnclavePage*, once again the *KeFeatureBits* are checked to see if SGX is implemented, and if so, *KiEncls* is now used with leaf function **EREMOVE** (3).

Instruction Operand Encoding

Op/En	EAX	RCX
IR	EREMOVE (In)	Effective address of the EPC page (In)

Some additional error codes from SGX itself must now be analyzed. For example, if **SGX_CHILD_PRESENT** is returned, this means that the SECS is being removed, but there are still valid EPC pages loaded. In this case, **STATUS_CONFLICTING_ADDRESSES** will be returned. Otherwise, if **SGX_ENCLAVE_ACT** is returned by the SGX hardware, this means that active enclave code is still running, which will result in **STATUS_UNABLE_TO_FREE_VM**.

Interestingly, *MiDeleteEnclavePage* completely ignores any of these error codes, and continues with its logic, which destroys the PTE, and places the PFN back into Enclave Page List by calling *MiInsertPageInFreeOrZeroedList* (recall that a special flag instructs this routine to use the Enclave Page List instead).

Note that this apparent bug actually makes a lot of sense – because entering and exiting from an enclave are controlled by the user-mode process, if a user wants to terminate a process that contains a thread which is currently in the middle of executing inside of an enclave, there is no way to "force" that thread to exit the enclave. In this case, an SGX error would be returned during the destruction of the process' address space, which would force the VAD to remain active, and thus force the process to remain present. Ultimately, this would allow unprivileged user-mode applications from making themselves unkillable by entering into an enclave and never exiting.

On the other hand, the enclave code *will* actually continue to execute, and the memory it resides in will have been returned back to the memory manager as free enclave pages. SGX is smart enough to prevent those not-actually-free pages from being associated with another enclave in the future, so they'll essentially be leaked and never again be made available (even if the enclave eventually exists, nobody will be calling **EREMOVE** anymore on those pages).

This is an interesting corner case attack, which at worse seems like it would result in a DoS of enclave facilities. The kernel team may not have had enough time to come up with a reliable mechanism to avoid this situation, or one may not exist (the author can imagine some sort of forced trampoline code that the EIP of a terminating thread is sent to, which contains the **EEXIT** leaf function).

Continuing back to *MiDeleteEnclavePages*, once all the EPC pages have been freed, the system PTE that was used to map the SECS to a kernel address (when needed) is released, and *MiDeleteEnclavePage* is called on it as well. Finally, *MiReturnReservedEnclavePages* is used to loop over any EPC pages that were in the PFN array part of the Enclave VAD. These may have come from when an initial commitment was requested inside of *CreateEnclave* but not all the pages were demanded (meaning that they would not appear as valid PTEs).

Miscellaneous Enclave Operations

Although everything described so far would satisfy the requirements that SGX needs from system software, a few additional services and features are offered to complement SGX support in Windows 10.

Changing Protection of Enclave Pages

As one last side-effect of the fact that enclave memory is described by a VAD, what happens if you were to call *VirtualProtect*, i.e.: *NtProtectVirtualMemory,* on enclave memory? Interestingly, this is actually supported. If the Enclave Bit is set on the corresponding VAD, *MiProtectEnclavePages* will be called. First, it validates that the protection attributes are valid, as well as that **`PAGE_ENCLAVE_UNVALIDATED`** to be set – once pages have been measured, protection changes are not allowed.

If all the validations work out, *MiUpdateEnclavePfnProtection* is called, which updates the PFN entry associated with each of the EPC pages associated with the enclave's user-mode memory. Finally, the TLB is flushed for the changes to take effect. We talked about how *MiCopyPagesIntoEnclave* had a special check to prevent executable enclave pages from being created in a process that is currently mitigating against dynamic code generation. *MiProtectEnclavePages* does not have a similar check, but that's OK – *MiProtectVirtualMemory* calls *MiAllowProtectionChange* before we even get here.

It is important to realize that the protection changes apply to the non-enclave, user mapping of the pages, and will affect if the user-mode application is allowed to read, write, or execute the enclave pages while in normal mode. Changing the protection (`SECINFO` Flags) of the actual EPC pages in SGX mode is not allowed, as no SGX instruction exists to do so. Note, however, that SGX2 does implement a leaf function, **`EMODPR`** (`0Eh`) which allows this change. Unfortunately, Windows 10 currently only supports SGX1, and so this functionality is not exposed.

Debugging Enclaves

You may have seen that one of the attributes that the SECS could specify when creating the enclave, and which was saved as one of the flags in the Enclave VAD, is the DEBUG attribute that determines if an enclave has debug capabilities. These capabilities allow usage of some additional SGX leaf functions, specifically **`EDBGRD`** (4) and **`EDBGWR`** (5).

To reach this functionality, standard Win32 API debug APIs can be used, which makes enclaves visible to standard user-mode debuggers such as WinDBG or Visual Studio. These APIs are the usual *ReadProcessMemory* and *WriteProcessMemory*, which correspond to the *NtReadVirtualMemory* and *NtWriteVirtualMemory*. Again, because enclave memory is described by a VAD, the memory manager can realize that these are actually special regions of memory.

Both of the system calls will eventually call into *MiReadWriteVirtualMemory*, which ultimately calls into *MmCopyVirtualMemory*. Until now, all these call chains behave as usual. It's inside *MmCopyVirtualMemory* that the first change is implemented: a call to *MiFindNextEnclaveBoundary* is performed early on, to scan if there are any enclaves inside of the virtual address range that is either being read from or written to. This is done by enumerating the VADs for each region, and again checking for that `Enclave` bit for any `VadAwe`-type VADs. Once an enclave VAD is located, the `Debug` flag is checked, and if not set, the VAD is skipped – behaving as if no valid memory is actually located there.

Once *MmCopyVirtualMemory* begins its actual copying logic, it checks for any regions on the list that it sees as Enclave memory (which implies it's debuggable, otherwise it would not have been returned by

MiFindNextEnclaveBoundary). For any such memory, it calls *MiDbgReadWriteEnclave* with the source and destination addresses.

Note that two possible operation modes exist: we may be reading FROM an enclave TO regular memory, or reading FROM regular memory INTO enclave memory. Similarly, one may be writing INTO regular memory FROM enclave memory, or INTO enclave memory FROM regular memory. *MiDbgReadWriteEnclave* correctly deals with both scenarios, and uses either *MiDbgReadWriteEnclaveUnaligned* if the data is not 8-byte aligned, or its own copying loop for aligned data.

Obviously, as an optimization, it uses the unaligned function for the header and footer of the buffer, keeping the bulk aligned, instead of performing a full unaligned copy. To actually read or write the data, however, it must use *KeDebugReadEnclaveMemory* and/or *KeDebugWriteEnclaveMemory*, which should get us into SGX hardware land for the operation.

Indeed, both of these functions will check *KeFeatureBits* to check for SGX availability, and issue the **ENCLS** instruction with the correct leaf function. Interestingly, however, *KeDebugWriteEnclaveMemory* uses the usual *KiEncls* function we have seen multiple times (with **EDBGWR** as a leaf), but *KeDebugReadEnclaveMemory* uses a completely new function called *KiEnclsDebugRead*. It behaves identically to *KiEncls* with one subtle difference – it returns the value of the **RBX** register, and not **RAX** as you'd expect. Let's take a look at both leaf functions in the Intel manual:

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGWR (In)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

vs.:

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGRD (In)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

Note how, for the first time, we have an SGX leaf function that returns an output parameter! Due to this, *KiEncls* would not correctly return the data required, but rather the error code. *KiEnclsDebugRead* was created as it is the only way to read the **RBX** register without resorting to writing assembly code. Even in SGX2, **EDBGRD** is the only function that uses an output parameter, so there's no overloading of *KiEnclsDebugRead* – this function may be renamed if future SGX3 instructions have outputs also in **RBX**.

Conclusion

In this whitepaper, we have seen how the latest update to Windows 10 now includes support for Intel's SGX functionality, and potential extensibility for future similar enclave technologies as they are developed.

Additionally, we've seen that SGX2 is not yet implemented by Windows 10, somewhat limiting some of the capabilities that are provided, as pages cannot be added to an initialized enclave, and existing page protections cannot be modified (a final detail is that new threads cannot be added into an enclave dynamically).

Another SGX2-related feature is the ability to evict (page-out) and load (page-in) pages more effectively, such as by marking a page as being trimmed. Although SGX1 already supports evict/load capability, it appears that Microsoft has chosen not to support it, perhaps waiting for SGX2 support in order to gain its additional flexibility.

From a security standpoint, we've seen that the kernel attempts to mitigates anise of the risks that enclaves pose – such as continuing to allow debugging of enclaves (if the enclave allows it), preventing unkillable processes due to enclave use (but still allowing DoS of enclave services, potentially), preventing the execution of enclave-associated memory if dynamic code generation is disabled, and finally preventing the cross-architecture execution of enclave instructions.

However, this doesn't change the fact that enclaves now provide an invisible way to execute instructions which has little to no visibility into endpoint security products. Hopefully, the signing and cryptographic requirements behind enclaves will make them less easily abused, but the risk remains, compounded by the fact that one can cause the kernel to see a page as being unused, even while an enclave is still executing (due to the issue described earlier around process exit).

It's important to realize that for now, only Intel has the required key to allow an enclave to be launched without knowing the required CPU-specific enclave key, and no other (even signed) enclaves can be launched without it. Once Intel releases a permissive loader, or if Intel ME vulnerabilities are found to extract the key, then the real abuse will begin.

Indeed, one area of further research is the Intel SGX Driver that was released for recent Intel SGX-enabled Dell Laptops, which contains a `le.signed.dll` file that is the Intel Launch Enclave. Additionally, it contains Intel's EINITTOKEN that can be used to launch such enclaves, as well as a service and set of APIs which appear to make it possible to launch additional enclaves. Windows 10, on its own, does not seem to ship or support its own Intel-signed Launch Enclave.

Index

References

ARM. (2015). Retrieved from http://www.arm.com/products/processors/technologies/trustzone/

Intel. (2014, October). Retrieved from https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

Intel. (2015, October). *Product Change Notification*. Retrieved from https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf