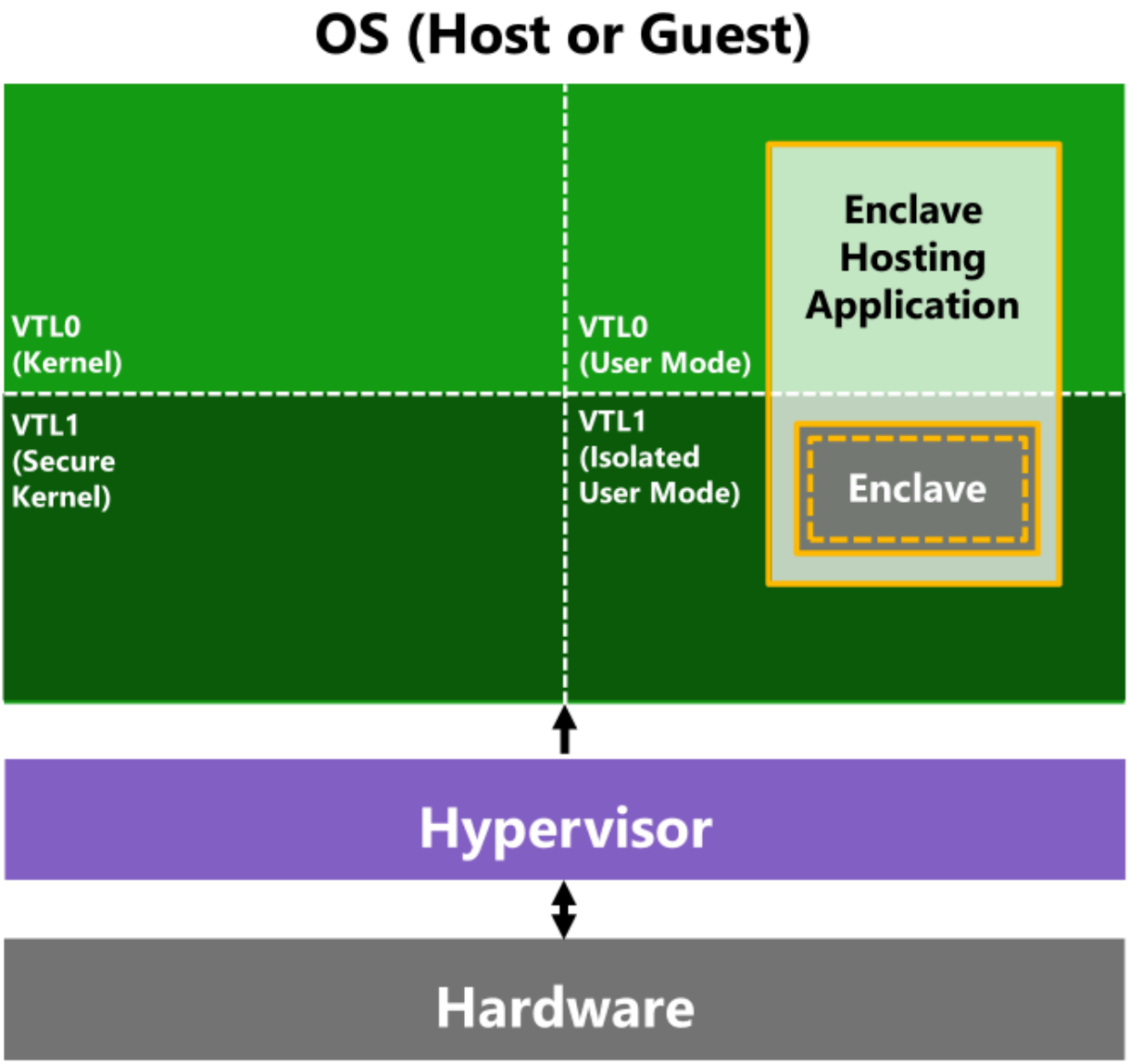
Secure Enclaves for Offensive Operations (Part I) | Outflank

outflank.nl/blog/2025/02/03/secure-enclaves-for-offensive-operations-part-i

Cedric Van Bockhaven February 3, 2025



Architecture of Enclave Hosting within VTL0 and VTL1 – source Microsoft

This blog post was co-authored by Matteo Malvica (Researcher at OffSec and *External OST developer) and Cedric Van Bockhaven (OST developer and researcher at Outflank).*

This article is the first in a two-part series in which we investigate the anatomy of Virtualization-Based Security (VBS) enclaves, their internals, and the unique ways they could be leveraged for offensive operations on Windows systems.

Enclaves provide a software-based Trusted Execution Environment (TEE) and are isolated memory regions. Only code that runs within the enclave can access data within the same enclave. TEEs protect sensitive operations in computing and are designed to keep unauthorized actors away from confidential information, whether that actor is malware with user-mode or kernel-mode access or even someone with physical access to the data center.

You can imagine that being able to hide away data and code into an enclave is a powerful capability and could also be used for offensive purposes, and that's exactly what this series will explore. Note that using enclaves for offensive operations is not new. The concept has been presented before at DEF CON 29 by Dimity 'Op Nomad' Snezhkov in their talk, "Use Of Offensive Enclaves in Adversarial Operations". Their talk was focused on the use of Intel SGX enclaves. With Intel deprecating SGX technology for workstations in 2021, we will be investigating the use of VBS enclaves instead.

This blog series will not address vulnerabilities in the architecture or implementation of the Secure Kernel that is used by VBS enclaves. However, there is a great presentation on this topic: BlackHat USA 2020 by Saar Amar and Daniel King on "Breaking VSM by Attacking Secure Kernel". Instead, we will focus on the practical use of VBS enclaves.

From an offensive perspective, the appeal of enclaves lies in their ability to securely store (and process) data, creating an opportunity for attackers to embed malicious data or logic in areas where defenders are unlikely to look – or are unable to look at all.

For red team engagements, we could see enclaves as a vector for slipping malicious code into a secure region to gain a foothold that is harder for EDR solutions to inspect. Even EDRs with kernel access cannot inspect VBS enclaves that are running in isolated user mode.

Let us preface by saying that functionality of VBS enclaves is restricted – you are able to operate on data but won't have regular Windows APIs available within the enclave to set up TCP sockets or even open files on the file system.

Virtualization-Based Security and Enclaves

In short: VBS enclaves provide a software-based TEE and were added as an extension to isolated user mode in the VBS environment.

VBS utilizes the Hyper-V hypervisor to create isolated environments. Like VM isolation, the hypervisor sets memory protections to isolate this environment from the rest of the system kernel via SLAT (second-level address translation).

VBS is the technology that enables Credential Guard to protect credentials by storing them in isolated memory. Microsoft extended the isolated user mode in the VBS environment to allow developers to protect portions of application data in a software-based TEE known as a VBS enclave. A basic premise of TEEs is that code and data residing within these isolated pockets of memory are protected from the rest of the system.

In hardware-backed TEEs like Intel SGX, the CPU enforces these privileges. In software-based TEEs like Microsoft's VBS Enclaves, the Secure Kernel (running in a separate virtualization stack) takes responsibility for ensuring that no process or even the Windows OS kernel can tamper with enclave-protected memory.

Virtualization is used to enforce isolation. This protective boundary is typically enforced using page-table manipulations and heavily restricted APIs to load and run enclave modules.

If you've worked with Intel SGX, you know that enclaves introduce some complexities in development and debugging, including limitations on standard memory inspection tools.

Microsoft's VBS enclaves, although more flexible than hardware enclaves, share a similar level of opacity and bring in additional challenges like production-signing your enclave.

The architectural design of VBS enclaves allows the protection of secrets, intellectual property, or secret operations inside the enclave.Here are two examples of how VBS enclaves are currently used by Microsoft:

- Enhanced Phishing Protection in Microsoft Defender SmartScreen makes use of secure enclaves. This mechanism records keystrokes and clipboard data to check whether you enter your password in untrusted resources. For this, your password is intercepted at login and stored in hashed format in an enclave (SFAPE.dll).
- SQL Server 2019+ and Azure SQL Database can make use of secure enclaves through its Always Encrypted feature.

As enclaves can provide a secure processing environment, they could also make a good fit for DRM processing or anti-cheat purposes.

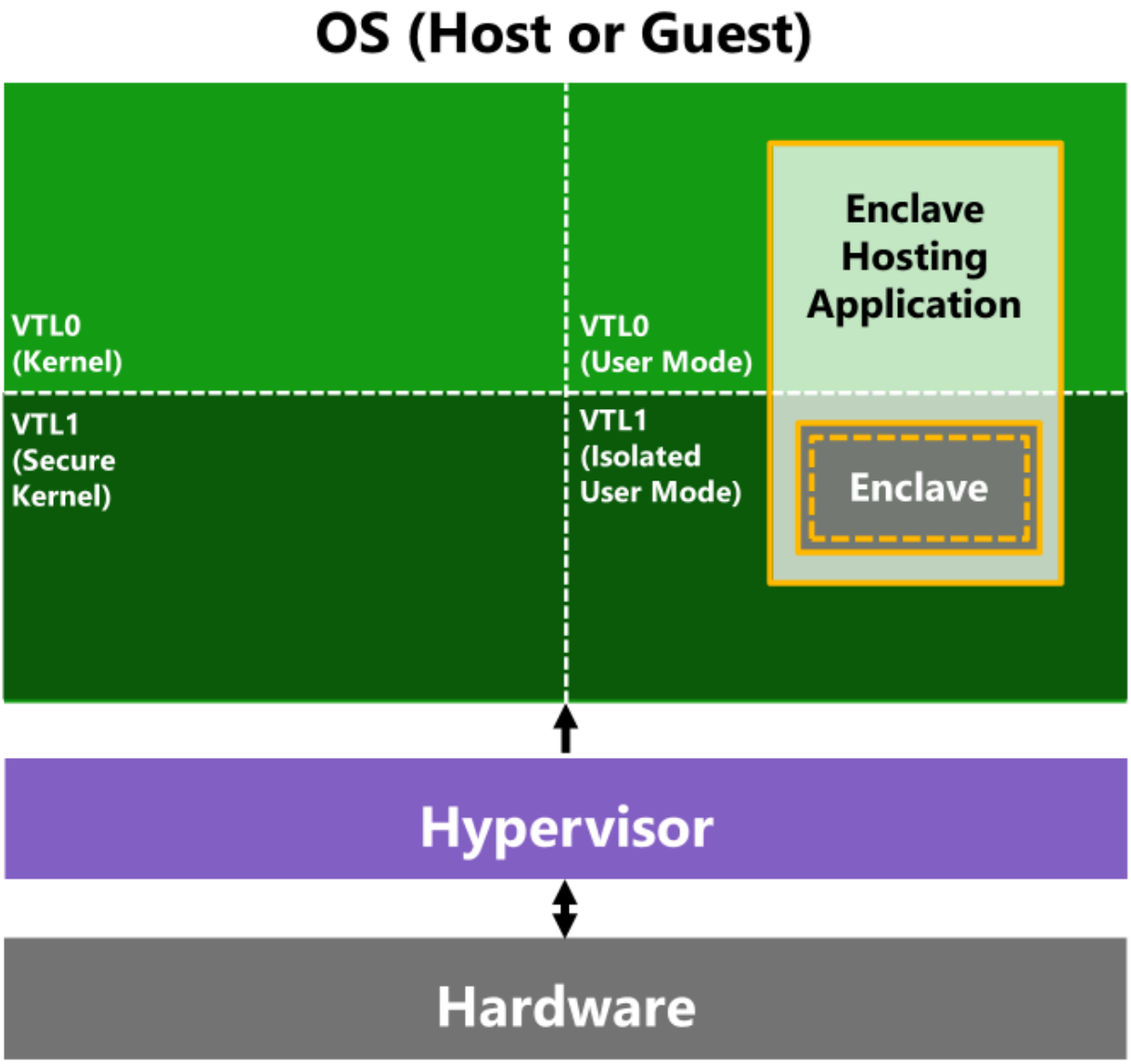
Anatomy and Operations

VBS enclaves operate within a multi-layered architecture that enforces strict isolation using Virtual Trust Levels (VTLs). These levels allow the separation of standard user-mode

processes (VTL0) and isolated secure environments (VTL1) managed by the Secure Kernel.

The diagram below illustrates how this architecture is layered. At the base, the Hyper-V hypervisor provides virtualization and enforces isolation. Above the hypervisor lies the operating system.

- VTL0 – Normal World
  - Runs the NTOS kernel
  - Regular user-mode apps (e.g. Word, Excel, …)
- VTL1 – Secure World
  - Runs the Secure Kernel
  - Isolated User Mode (IUM) Apps and Enclaves.



Architecture of Enclave Hosting within VTL0 and VTL1 – source Microsoft

By building on this architecture, VBS enclaves implement two components: the Enclave Host Application in VTL0 and the Enclave DLL in VTL1. Let's examine how each of these components operates and interacts within the VBS framework.

- **Enclave Host App (VTL0 – Standard User Mode)**
  The host application creates and initializes the enclave, allocating a virtual address range and loading the enclave module (a signed DLL). It then communicates with the enclave via the `CallEnclave` API.

  It's worth emphasizing that the host application cannot access the enclave's protected memory. We can only interface with the Enclave through the available APIs.
- **Enclave DLL (VTL1 – Isolated User Mode)**
  The enclave is represented by a digitally signed DLL containing configuration metadata (e.g., family IDs, policies). The Secure Kernel verifies the DLL's integrity before execution and enforces isolation.

  Once loaded, the enclave's memory is isolated. Code running within the enclave can interact with its internal memory or the parent process's address space in VTL0 but cannot access or be accessed by other system components. This isolation is further enhanced by Secure Kernel policies that include *Sealing* (encryption of sensitive data tied to the enclave's identity) and *Attestation* (providing proof of integrity and identity to third-party systems).

To better understand how enclaves operate, let's break down the enclave lifecycle into its key stages.

VBS Enclave Lifecycle – source Microsoft

- ***Creation***: The Secure Kernel sets up the enclave's memory, verifies signatures, and ensures the virtual memory for the enclave is isolated.
- ***Loading***: Modules are loaded into the enclave via `LoadEnclaveImage`, and the Secure Kernel maps these modules to VTL1 memory using a top-down strategy.
- ***Initialization***: The enclave is initialized using `InitializeEnclave`, which finalizes the setup by hashing the loaded modules and configuring execution threads for VTL1.
- ***Runtime Operations***: Once initialized, the host app invokes callable functions in the enclave using `CallEnclave`. Each call is routed to the Secure Kernel, which maintains a thread mapping mechanism to handle nested or concurrent enclave interactions.

Isolated User Mode Apps vs Enclaves

A popular example of a mitigation that is based on VBS is Credential Guard. Credential Guard hides secrets in user-mode VTL1, which is unreadable to VTL0.

Credential Guard makes use of an Isolated User Mode app, also known as a trustlet (LsaIso.exe). The IUM app runs entirely in VTL1. This differs from the VBS Enclave model we have described above as the Enclave Host App runs in VTL0.

Isolated User Mode apps (trustlet) – source Microsoft

There are other trustlets such as NgcIso.exe (Windows Hello Security Process) and BioIso.exe (Secure Biometrics). These trustlets are created by Microsoft – there is currently no way of creating and loading your own trustlet as a third-party developer.

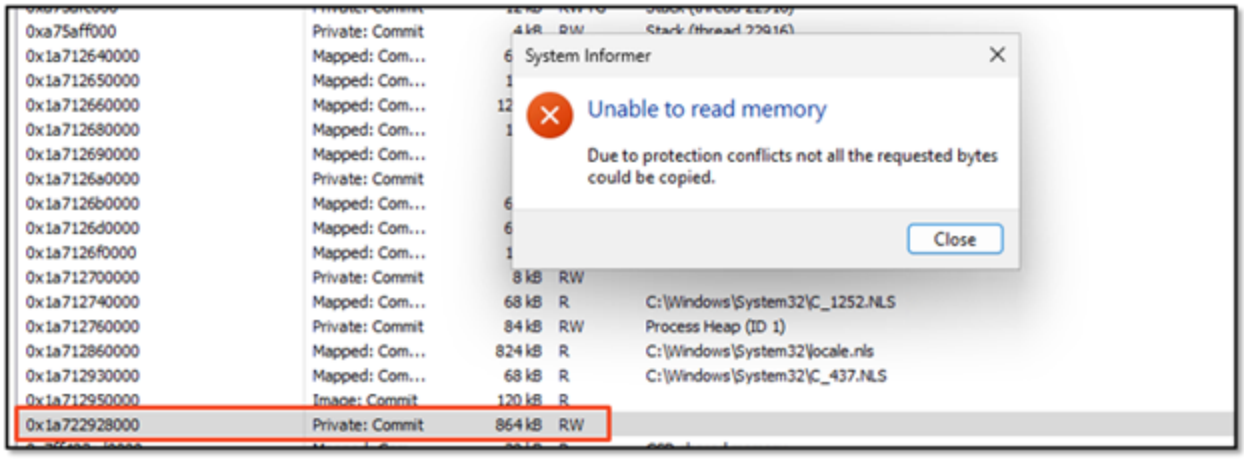
Developing and Debugging Enclaves

VBS Enclave development was made available to third-party application developers in May 2024. VBS enclaves have been around longer than May 2024 though, and functionality to support them has been around since the first versions of Windows 10.

The application creating the enclave is responsible for providing a signed DLL that gets loaded by VTL1. For the Secure Kernel to consider it trustworthy, the DLL must be production-signed using Azure Trusted Signing.

The Enclave DLL should export the functions that can be called via `CallEnclave` from the host application. Non-exported functions cannot be called (we will see why in the next blog post in this series). After creating the two key components – our host application and the Enclave DLL – we are ready to test our newly created VBS enclave application.

As we begin inspecting the memory footprint, we'll encounter a key characteristic of VBS enclaves: their memory is marked as read-write (RW) but remains inaccessible due to the enclave running in Virtual Trust Level 1 (VTL1).



Unable to access VBS enclave memory

Even though debugging VBS enclaves is challenging due to their isolated memory model and strict runtime requirements, we have multiple ways of inspecting them.

When developing our own enclave DLL we can enable debugging by:

- First, configuring the enclave's `IMAGE_ENCLAVE_CONFIG` to include the `IMAGE_ENCLAVE_POLICY_DEBUGGABLE` flag during compilation.
- Second, setting the `ENCLAVE_VBS_FLAG_DEBUG` flag in the `ENCLAVE_CREATE_VBS_INFO` structure passed to the `CreateEnclave` call.

In addition to setting these flags, the DLL must be signed with a test certificate to allow loading in a development environment with test-signing enabled. You can then use a debugger like WinDbg to debug the enclave indirectly by attaching to the hosting process and inspect memory. In our experience, other debuggers don't correctly support Enclave DLLs at the time of writing.

Below is an example showcasing the critical parts needed to enable debugging in a VBS enclave DLL:

```
1   extern void* Shellcode(void* Context);
2
3   const IMAGE_ENCLAVE_CONFIG __enclave_config = {
4       sizeof(IMAGE_ENCLAVE_CONFIG),
5       IMAGE_ENCLAVE_MINIMUM_CONFIG_SIZE,
6       IMAGE_ENCLAVE_POLICY_DEBUGGABLE,
7       // Additional configurations...
8   };
9
10  BOOL DllMain(_In_ HINSTANCE hinstDLL, _In_ DWORD dwReason, _In_ LPVOID lpvReserved) {
11      // Initialization logic...
12      return TRUE;
13  }
14
15  void* CALLBACK CallEnclaveTest(_In_ void* Context)
16  {
17      void* newContext;
18
19      // Mock two NOP instructions
20      _mm_pause();
21      _mm_pause();
22
23      newContext = Shellcode(Context);
24      return newContext;
25  }
```

The `CallEnclaveTest` callback is invoked by the user-mode host application. Within this function, we simulate two NOP instructions with `_mm_pause` and then call our custom assembly shellcode function.

```
1  Shellcode PROC
2      mov rax, 0DADAF00Dh      ; Load static key into RAX
3      xor rax, rcx             ; XOR key with the first argument
4      nop                      ; No-op for alignment
5      ret                      ; Return result in RAX
6  Shellcode ENDP
```

The Shellcode function takes the first argument (passed in RCX), XORs it with a fixed 4-byte static key (`0xDADAF00D`), and returns the result in the RAX register.

Once the enclave DLL is compiled and loaded by the host process, you can attach a debugger to the host and inspect the enclave's operations. The video clip below demonstrates this process, showing how debugging works seamlessly within these constraints.

Crafting our own enclave DLL is an excellent exercise for understanding enclave development, debugging, and code execution. **However, for offensive purposes, deploying custom Enclave DLLs in target environments is not really feasible unless test-signing is available.** Additionally, it is likely that usage of enclave DLLs like this would likely fall under Azure Trusted Signing misuse/abuse. Furthermore, usage of Azure Trusted Signing requires organizations to be founded at least 3 years ago. Not a route you want to take.

Instead, we could shift our focus to identifying and analyzing Enclave APIs as well as existing native enclave DLLs already present on our systems.

Abusing Existing Enclave APIs and DLLs

Enclave APIs and available Enclave DLLs might hold the keys to uncovering undocumented functionality, exploitable behaviors, or even stealthy ways for our operational advantage.

Regarding APIs we could potentially abuse, in the past there was this Driver-Signature Enforcement (DSE) bypass abusing the NtLoadEnclaveData syscall. There's also this CallEnclave shellcode execution "dispatcher" on the FlavorTown GitHub. We will dive deeper into these APIs in the next blog post in this series.

Using existing DLLs we avoid the need to use Azure Trusted Signing. This also ensures that we use already trusted components, helping us hopefully remain under the radar a bit better without having to introduce a new DLL on disk.

What if we want to debug an existing Enclave DLL to discover attack surface? Since existing Enclave DLLs would normally be production-signed, we would need to resort to Secure Kernel debugging, which has quite some steps involved. However, it allows us not only to debug enclave DLLs but also the Secure Kernel itself.

Side note: at the time of writing, we had issues setting breakpoints within enclave DLL code functions on newer Windows versions. In our testing, Windows 22H2 was the only version in which we could set breakpoints.

If we do not need to debug the Secure Kernel and only want to target the user mode enclave, we could also benefit from test-signing and modify the `IMAGE_ENCLAVE_CONFIG` of an existing Enclave DLL. We made a small Python tool that allows inspect this structure in an Enclave DLL and toggle the `IMAGE_ENCLAVE_POLICY_DEBUGGABLE` flag in the `PolicyFlags` field. The script also allows directly viewing the exported functions in an Enclave DLL.

Since enclave-related metadata is embedded in the DLL, we can parse the PE file to locate the `EnclaveConfigurationPointer`, found within the `IMAGE_LOAD_CONFIG_DIRECTORY` structure of the PE headers. The pointer, when present and valid, points to enclave-specific configuration data. By examining this field, we can uncover details like the enclave's size and exported functions. For our purposes, this information is gold – knowing what an enclave can do (or be made to do) is the first step in abusing it.

The `enclave_config.py` script can be found on GitHub (https://github.com/outflanknl/Scripts/blob/master/enclave_config.py).

```
C:\dev\random\enclaves\Enclaves>python enclave_config.py --help
usage: enclave_config.py [-h] [--output OUTPUT] [--debuggable] input_pe

Extract and modify IMAGE_ENCLAVE_CONFIG in PE files.

positional arguments:
  input_pe              Path to the input PE file

options:
  -h, --help            show this help message and exit
  --output OUTPUT, -o OUTPUT
                        Path to save the modified PE file (required for --debuggable)
  --debuggable          Toggle the IMAGE_ENCLAVE_POLICY_DEBUGGABLE field
```

```
C:\dev\random\enclaves\Enclaves>python enclave_config.py --debuggable SFAPE.dll --output SFAPE.dbg.dll

Detected 64-bit PE file.

IMAGE_ENCLAVE_CONFIG extracted:
 - Size: 80
 - MinimumRequiredConfigSize: 76
 - PolicyFlags: 0x0 (Not Debuggable)
 - NumberOfImports: 3
 - ImportList: 0x8fb4
 - ImportEntrySize: 80
 - ImageVersion: 0x1
 - SecurityVersion: 1
 - EnclaveSize: 0x1000000
 - NumberOfThreads: 10
 - EnclaveFlags: 0x1

Exported Functions:
 - 0x180006c90: CharactersInput
 - 0x180006d70: ClipboardInput
 - 0x180006e50: CloseCcHandle
 - 0x180006fa0: GetCcAttributes
 - 0x180007080: Initialize
 - 0x180007100: MeasurePerformance
 - 0x180007250: SealUnsealCredentials
 - 0x180007400: SetCcTimeStamps
 - 0x1800074e0: UserCredentials

Modified IMAGE_ENCLAVE_POLICY to 0x1 (Debuggable)
Modified PE saved to SFAPE.dbg.dll
```

To find existing enclave DLLs, we can attempt to find PE files which have a valid `EnclaveConfigurationPointer`. We started a search for DLLs on our own system to locate ones with a valid enclave configuration. The results of this are shown below:

```
[VBS-Enclave DLL] "C:\windows\system32\bcrypt.dll"
    -> EnclaveConfigurationPointer = 0x18001E000

[VBS-Enclave DLL] "C:\windows\system32\bcryptprimitives.dll"
    -> EnclaveConfigurationPointer = 0x180069CE0

[VBS-Enclave DLL] "C:\windows\system32\SFAPE.dll"
    -> EnclaveConfigurationPointer = 0x1800099A0

...
```

When scanning the system, we will typically come across either primary or secondary enclave images.

- Primary Enclave Image: the Enclave DLL that is loaded by the host app.
- Secondary Enclave Image: provides functions (dependencies of the primary image).

We need to load at least one primary image (more will not work). It is not possible to load a secondary image as if it were a primary image and call its exports directly. More on this in Part II.

Not many DLLs are on the system (by default) that leverage enclaves. One example of a primary image is SFAPE.dll, used by Enhanced Phishing Protection in Microsoft Defender SmartScreen. You will typically find the following secondary images:

- bcrypt.dll, bcryptprimitives.dll
  Provides cryptographic functions.
- ucrtbase_enclave.dll
  Provides Universal C Runtime functions.
- vertdll.dll
  Provides low-level APIs and acts as primary interface between user-mode and the Secure Kernel. (A bit like the ntdll.dll of VTL1).

Now it's time to investigate whether those DLLs can provide further primitives that can be used for offensive purposes.

An Eye Toward Part II

We will be presenting the second part of our research in March at Insomni'Hack 2025 in Lausanne, Zwitserland. In that talk we will discuss the architecture of VBS enclaves, demonstrate how to create your own enclave, and explore practical exploitation techniques for enclave APIs and vulnerable enclave DLLs. We will show how to securely seal a beacon in an enclave during sleep and make it untouchable.

OST customers can already view a technical deep dive on this topic as part of the Tradecraft section of OST, or engage in community discussions on the Slack channel. If you're not an OST customer but are interested in learning more about OST, including the exclusive tradecraft such as this topic offered to users, we recommend scheduling an expert led demo to learn more.

With this post, we've laid out how enclaves are meant to work and explored how to get started with debugging. As offensive researchers, we at Outflank see these specialized security mechanisms as both a blessing and a curse for defenders – at the same time they almost always create new opportunities for attackers. Virtualization-Based Security offers robust isolated environments, with enclaves serving as an effective enhancement for processing sensitive data.

In Part II of this series, we'll delve into advanced techniques for exploiting VBS enclaves, including how to manipulate signed enclave DLLs and leverage enclave APIs to achieve stealthy defense evasion and hiding an implant.

In the meantime, if you are eager to experiment on your own, consider setting up a Windows 11 lab with VBS enabled and the latest version of Visual Studio. For detailed instructions on setting up VBS enclave development in Windows 11, refer to the Development Guide for VBS Enclaves.