Secure Enclaves for Offensive Operations (Part II)

outflank.nl/blog/2025/06/16/secure-enclaves-for-offensive-operations-part-ii

Cedric Van Bockhaven June 16, 2025

This blog post is the second part in a series about using Secure Enclaves for Offensive Operations. The first part discussed the basics of how enclaves work, provided some ideas on how to develop your own enclave, as well as analyze and debug existing enclaves. We also hinted at how enclaves could potentially be used for offensive purposes. *Remember: VTL0 is where the normal kernel lives, VTL1 is where the secure kernel operates (and our enclaves).*

In this follow-up post, we will share what we discovered while digging into enclave internals. It's been a hands-on journey filled with many (failed) experiments. We'll walk you through some of the practical techniques we used to exploit a read-write primitive in a vulnerable enclave DLL, and how we managed to turn that into VTL1 code execution.



The outcome of this research has been integrated into Outflank C2 (part of our OST offering) for implant sleepmasking. As a result, when the implant is dormant, its memory remains completely hidden. Even from VTL0 ring0 (this is where an EDR typically lives), it's not

possible to inspect the implant's memory or view the call stack of the VTL1 thread. We also added a sleepmask to Beacon Booster for Cobalt Strike. It handles sleep masking by storing the key material used to encrypt beacon memory in VTL1, while the implant data itself is stored in VTL0.

To make this work, we had to jump through several hoops. In this post, we'll walk through how we got there and what we had to deal with along the way. This research was presented during Insomni'Hack 2025 in Lausanne, make sure to check out the video of our talk.



Crossing the Trust Boundary

In Part I, we discussed how to find enclave DLLs and how to debug them. Microsoft SQL server also ships with several enclave DLLs, let's take a brief look at one of them.

As part of the Always Encrypted with secure enclaves feature in SQL server, secure enclave attestation is supported. The attestation process ensures that only trusted code is running inside the enclave before sensitive data is processed by establishing trust between the database client and server. The component responsible for handling this attestation is AzureAttest.dll.

In AzureAttest.dll (and several other enclave-related DLLs), you'll find typical pointer validation checks that determine whether a buffer resides inside or outside the enclave.



Why is that important? If the enclave were to read from or write to an arbitrary user-supplied pointer without validation, it could result in leaking or overwriting sensitive enclave memory. This could expose cryptographic key material or even allow tampering with code running in VTL1.

Josh Watson (Microsoft) has recently written a blog post about the necessity of performing these kinds of checks to prevent enclave vulnerabilities: Everything Old Is New Again: Hardening the Trust Boundary of VBS Enclaves. Basically, as we're passing data between VTL0 and VTL1, we are crossing a trust boundary. VTL1 shouldn't trust the data it receives from VTL0.

> The most important thing to remember is that while the host process cannot read or write in the enclave's memory region, the converse does not hold true – an enclave *can* read and write the memory of its host VTL0 process. This can create tricky situations when the enclave operates on pointers passed from the host process to the enclave.

So these pointer validations are there for a good reason. Beyond that, the blog post also explores other types of vulnerabilities, such as time-of-check/time-of-use (TOCTOU) issues. These can occur when a function is called and the pointers it relies on are updated in between, creating a race condition. You can address this by copying the structure data into the enclave function itself, so it can't be modified externally once the check has passed.

Unfortunately, implementing checks that can prevent these vulnerabilities are left as an exercise to the enclave developer, and we've seen them being implemented in different ways. Ideally, Microsoft would offer convenience functions as part of its API that perform these validations.

Microsoft Edge Preferences Enclave

Microsoft Edge bundles an enclave DLL, prefs_enclave_x64.dll, presumably for storing preferences securely. It's present by default on newer systems at `%WINDIR%/System32/Microsoft-Edge-WebView/prefs_enclave_x64.dll`. It has three exported functions, one for initialization (`Init`), one for sealing (`SealSettings`) and one for unsealing settings (`UnsealSettings`).



Despite what the name might suggest, this DLL can be used to seal and unseal any kind of data, not just configuration settings. For example, we can use it to encrypt our implant shellcode through the enclave. The enclave DLL does not restrict what kind of information is sealed. It's important to note that while the key material resides in VTL1, the encrypted data is still stored in VTL0. The enclave is merely used as a secure processing environment in which the data is sealed/unsealed. The implant memory is still accessible as it resides in VTL0, and the sleep function is performed in VTL0 too.

This technique is implemented in our Beacon Booster tool and allows sealing a Cobalt Strike beacon via this enclave DLL during sleep. Furthermore, a PoC for sealing arbitrary shellcode that capitalizes on this idea was weaponized by Ori David (Akamai) and explained in their blog post: Abusing VBS Enclaves to Create Evasive Malware.



Enclave arbitrary read-write

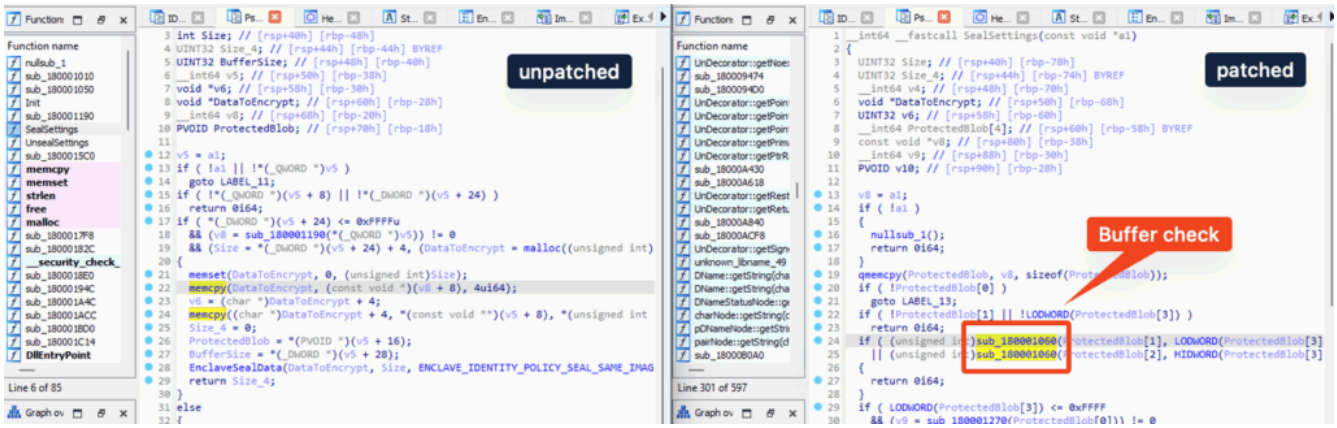
While the above sealing/unsealing of data is supported functionality, the same prefs_enclave_x64.dll DLL previously suffered from a pointer validation vulnerability as explained in the previous section.

As identified and reported by Alex Gough (quidity) from Google Project Zero, an older version of this enclave DLL was vulnerable for an abitrary read-write due to missing argument validation while passing pointers to the enclave functions `SealSettings/UnsealSettings`. This vulnerability was attributed CVE-2023-36880 and fixed by adding a pointer validation check. However, the fix for this CVE left a TOCTOU vulnerability as the user-supplied parameters weren't copied into VTL1. This was fixed separately and attributed CVE-2024-21423.

While these CVEs are classified as "Information Disclosure" that feels like a conservative label as a read-write vulnerability compromises both the confidentiality and integrity of the enclave.

The `SealSettings` and `UnsealSettings` functions take buffer arguments for reading and writing that weren't being validated (allowing leaking and overwriting of VTL1 enclave

memory). This was patched by introducing additional checks on arguments passed to the enclave function from VTL0. Here is a side by side view:



If we wanted to read from / write to VTL1 we could:

Arbitrary read: Seal VTL1 buffer → Unseal to VTL0

- `CallEnclave(SealSettings, {src: VTL1Pointer, dst: Wherever})`
- `CallEnclave(UnsealSettings, {src: Wherever, dst: VTL0Pointer})`

Arbitrary write: Seal VTL0 buffer → Unseal to VTL1

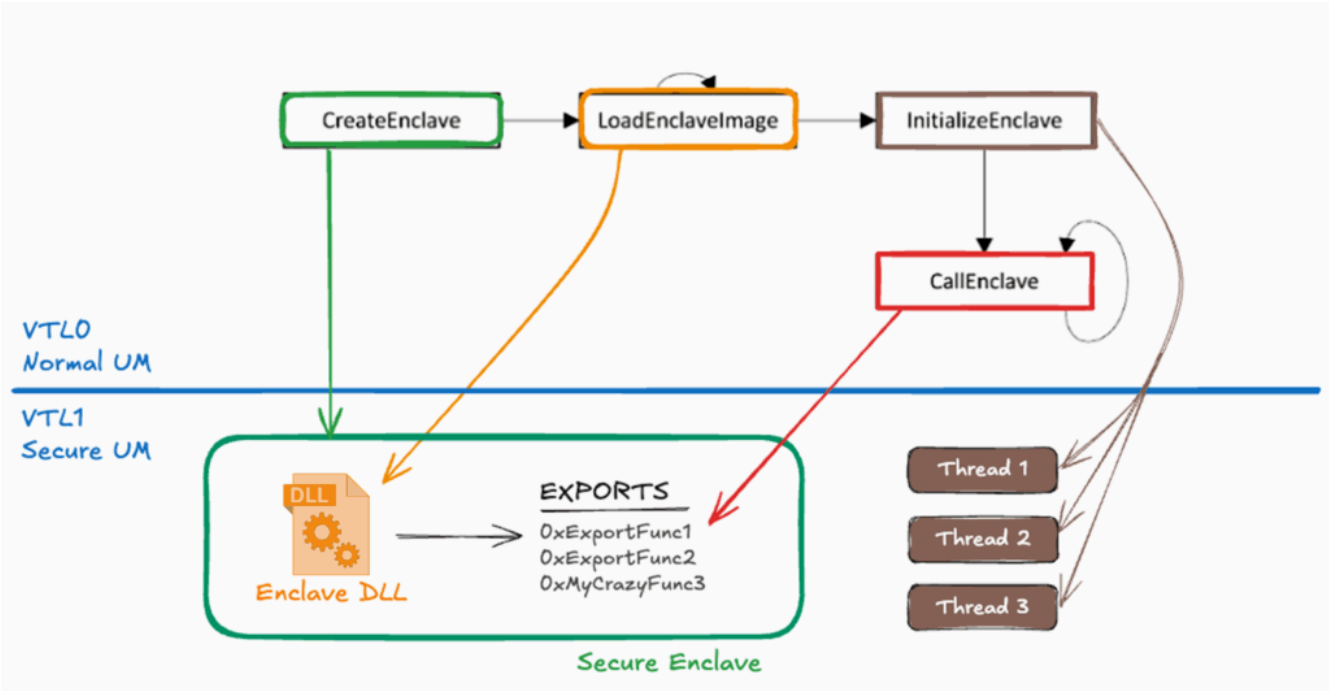
- `CallEnclave(SealSettings, {src: VTL0Pointer, dst: Wherever})`
- `CallEnclave(UnsealSettings, {src: Wherever, dst: VTL1Pointer})`



Great, this would allow us to leak some sensitive information stored by the enclave, but how can we weaponize the arbitrary write into doing something more meaningful? Let's take a step back first and review some enclave internals.

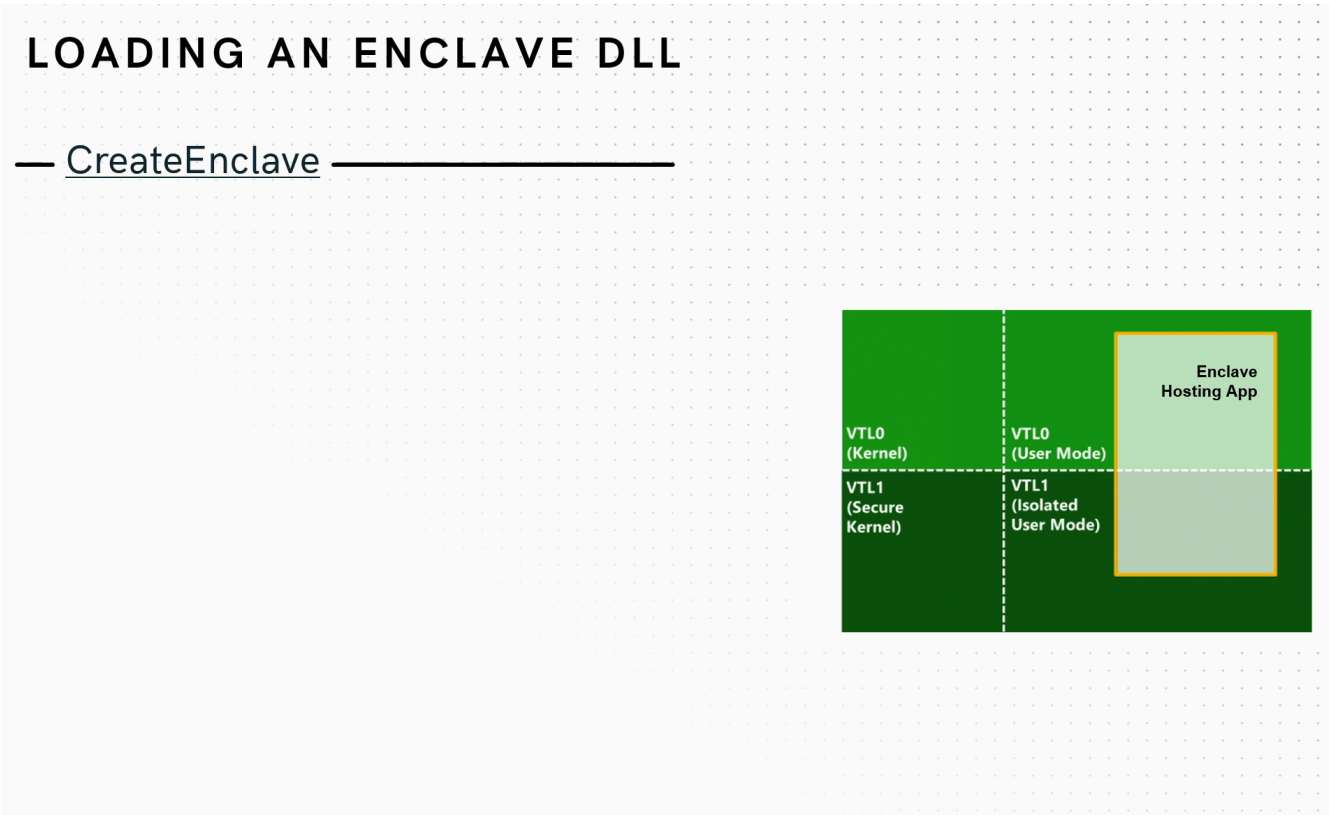
Revisiting the Enclave Life Cycle

In a naive attempt at trying to make an enclave do something for us, we tried to execute function addresses in VTL1 that weren't exported by calling `CallEnclave` with a custom address pointer. This fails, as exported functions of the primary enclave DLL are collected by both the VTL0 and VTL1 side individually, stored and verified. Let's dig in how this is enforced by exploring the enclave life cycle once more.



CreateEnclave

During creation of the enclave (`CreateEnclave`), we specify memory requirements – this memory is allocated on the VTL1 side and is in multiples of 2MB. The VTL1 memory allocation for your enclave cannot grow beyond this and comprises stack space, heap space as well as the primary and secondary enclave images. We can specify a location in memory where the enclave should be created, or have the OS assign the address. The WinAPI `CreateEnclave` eventually calls the NT API `NtCreateEnclave`.

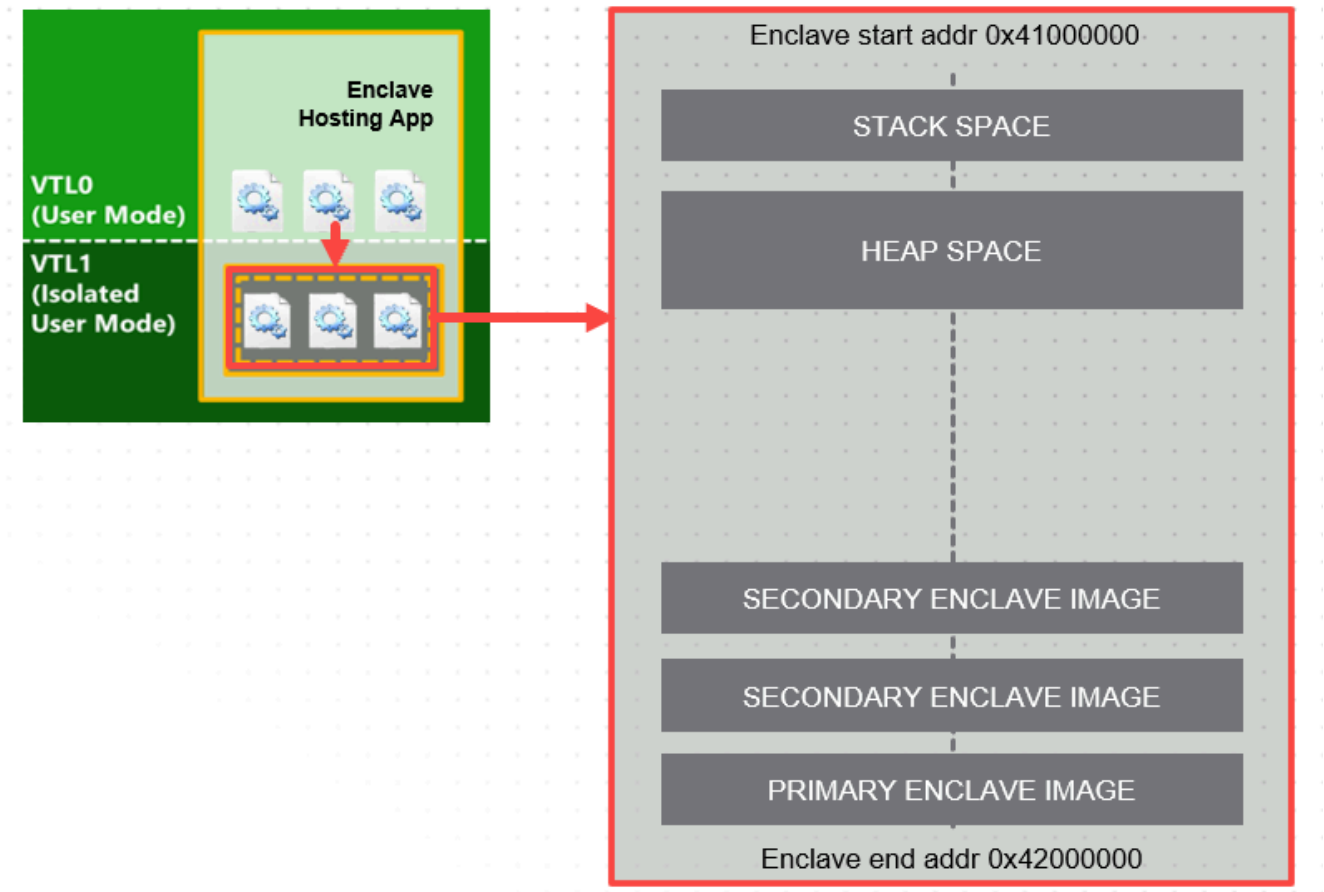


LoadEnclaveImage

We then load a primary enclave image (`LoadEnclaveImage`) of which the exports are parsed and stored. For this, the image is first loaded on the VTL0 side to determine the exports and then loaded into the enclave by passing the image section handle via a special invocation of the NT API `NtLoadEnclaveData`. Interestingly, there is no `NtLoadEnclaveImage`, there is however a higher level WinAPI named `LoadEnclaveData` (without the `Nt` prefix), which was intended for loading additional data into an SGX enclave and cannot be used by VBS enclaves (we tried).

Images are loaded top-down in the enclave address space. Here's how that looks for an enclave of size `0x1000000` (16MB) allocated at `0x41000000` (with end address `0x42000000` non-inclusive):

- Primary image address: `0x42000000–SizeOfImage`
- Secondary image address: `PreviousImageAddress–SizeOfImage`



Since the images are mapped adjacently in a deterministic layout within VTL1 memory, VTL0 can still infer the location of function pointers in VTL1. Consequently, the ASLR mitigations that we are familiar with for VTL0 user mode are not really a thing for VTL1 enclaves. First of all, there is not a whole lot of address space to randomize to begin with, but the enclave function pointers need to be in a predictable location too.

Secure kernel validates that the images are correctly signed and appropriate for use in the enclave. The export table containing function pointers is not passed from VTL0 to VTL1. Instead, VTL1 independently determines which exports are available within the enclave.

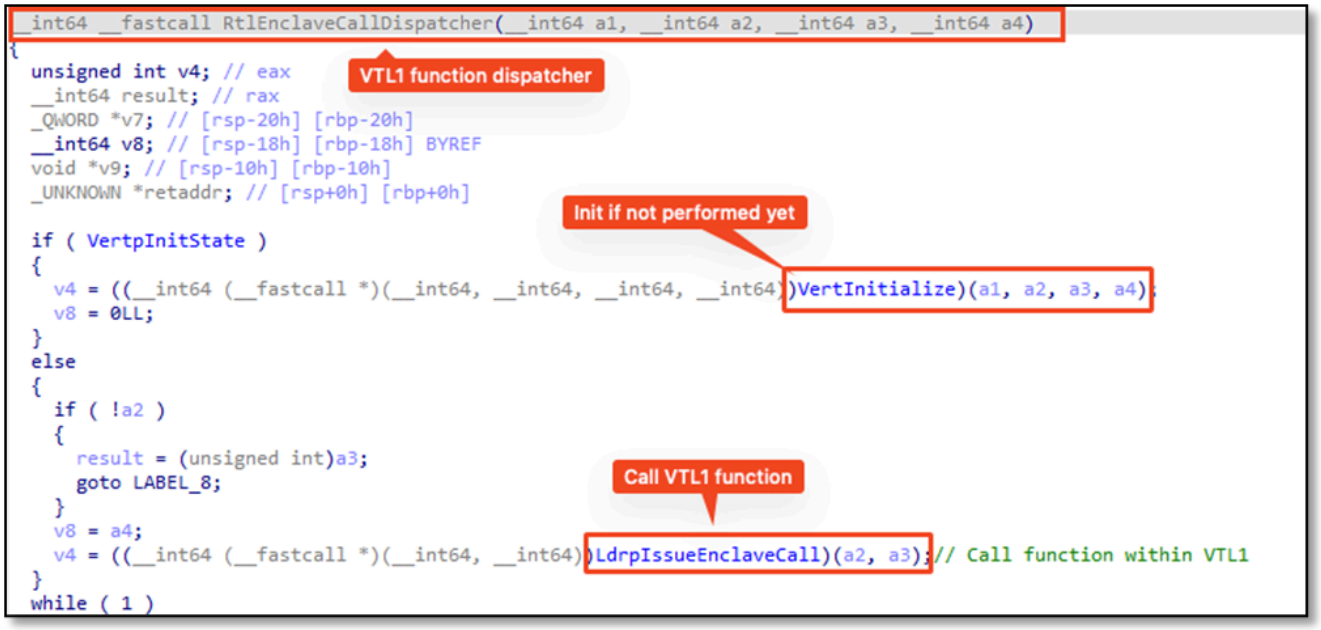
While the WinAPI implementation loads DLLs first on the VTL0 side before passing their section handle to load them in VTL1, it is possible to load DLLs from memory manually too as long as you can get a valid section handle. The WinAPI implementation will load the dependent secondary enclave images automatically, which a custom loader needs to perform manually. The WinAPI implementation keeps handles to the enclave images throughout the enclave lifetime, although they can be closed as soon as we no longer need them.

InitializeEnclave

We can then initialize the enclave (`InitializeEnclave` → `NtInitializeEnclave`) with a number of threads. Another validation is performed that the TEE is sane via a cryptographic check. Secure kernel then readies a number of suspended VTL1 threads by setting up the dispatcher address that can be called into by `CallEnclave`.

CallEnclave

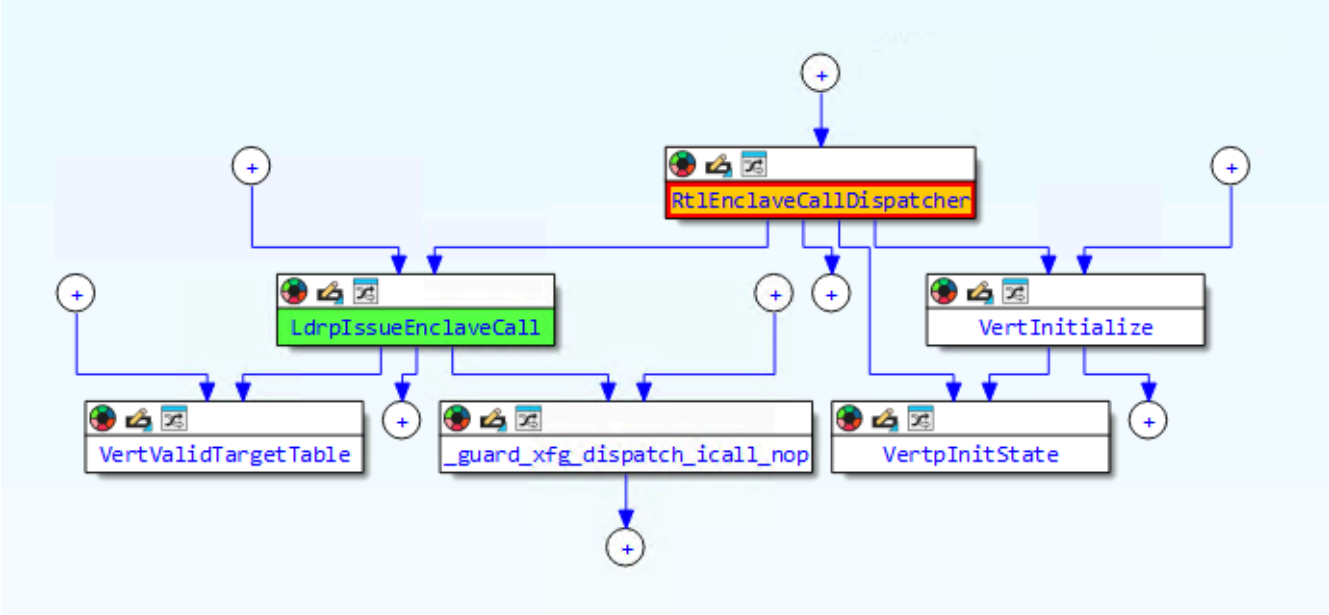
Now we can call an exported function in our primary enclave image (`CallEnclave` → `NtCallEnclave`). This function is available both in VTL0 and VTL1 to allow to transition to either side. When calling `CallEnclave` from VTL0, we will cross the trust boundary towards VTL1. On the VTL1 side, vertdll (basically the ntdll of our enclave) handles the call in `RtlEnclaveCallDispatcher`.




```
__int64 __fastcall RtlEnclaveCallDispatcher(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  unsigned int v4; // eax
  __int64 result; // rax
  _QWORD *v7; // [rsp-20h] [rbp-20h]
  __int64 v8; // [rsp-18h] [rbp-18h] BYREF
  void *v9; // [rsp-10h] [rbp-10h]
  _UNKNOWN *retaddr; // [rsp+0h] [rbp+0h]

  if ( VertpInitState )
  {
    v4 = ((__int64 (__fastcall *)(__int64, __int64, __int64, __int64))VertInitialize)(a1, a2, a3, a4);
    v8 = 0LL;
  }
  else
  {
    if ( !a2 )
    {
      result = (unsigned int)a3;
      goto LABEL_8;
    }
    v8 = a4;
    v4 = ((__int64 (__fastcall *)(__int64, __int64))LdrpIssueEnclaveCall)(a2, a3);// Call function within VTL1
  }
  while ( 1 )
```

First, the `VertpInitState` variable is checked to initialize and perform validations for the VTL1 side (e.g. verifying imports), as well as performing thread (and TLS) init routines. The VTL1 user-mode initialization is triggered by the first call into this function from a single enclave thread, which then performs the initialization routines for all other threads. The `VertpInitState` variable is locked in the meantime and then updated to signal that the enclave is ready for business.



During calls into the enclave, `RtlEnclaveCallDispatcher` forwards calls to `LdrpIssueEnclaveCall`, which handles execution of the requested enclave function. The return value of the enclave is then returned to the VTL0 side via the `NtCallEnclave` syscall (implemented by the VTL1 kernel).

In `LdrpIssueEnclaveCall` the enclave function pointer that we want to execute is verified by checking whether it is located in an inverted table of allowed targets: `VertValidTargetTable`. Using `CallEnclave` to call a function in an enclave will always reach that enclave (when the pointer is within the enclave's address space), but will be revoked if it is not in the `VertValidTargetTable`.

```
__int64 __fastcall LdrpIssueEnclaveCall(__int64 (__fastcall *a1)(_QWORD), __int64 a2, _QWORD *a3)
{
  __int64 result; // rax
  __int64 (__fastcall *Key)(_QWORD); // [rsp+70h] [rbp+8h] BYREF
  _QWORD *v6; // [rsp+80h] [rbp+18h]
  int v7; // [rsp+88h] [rbp+20h]

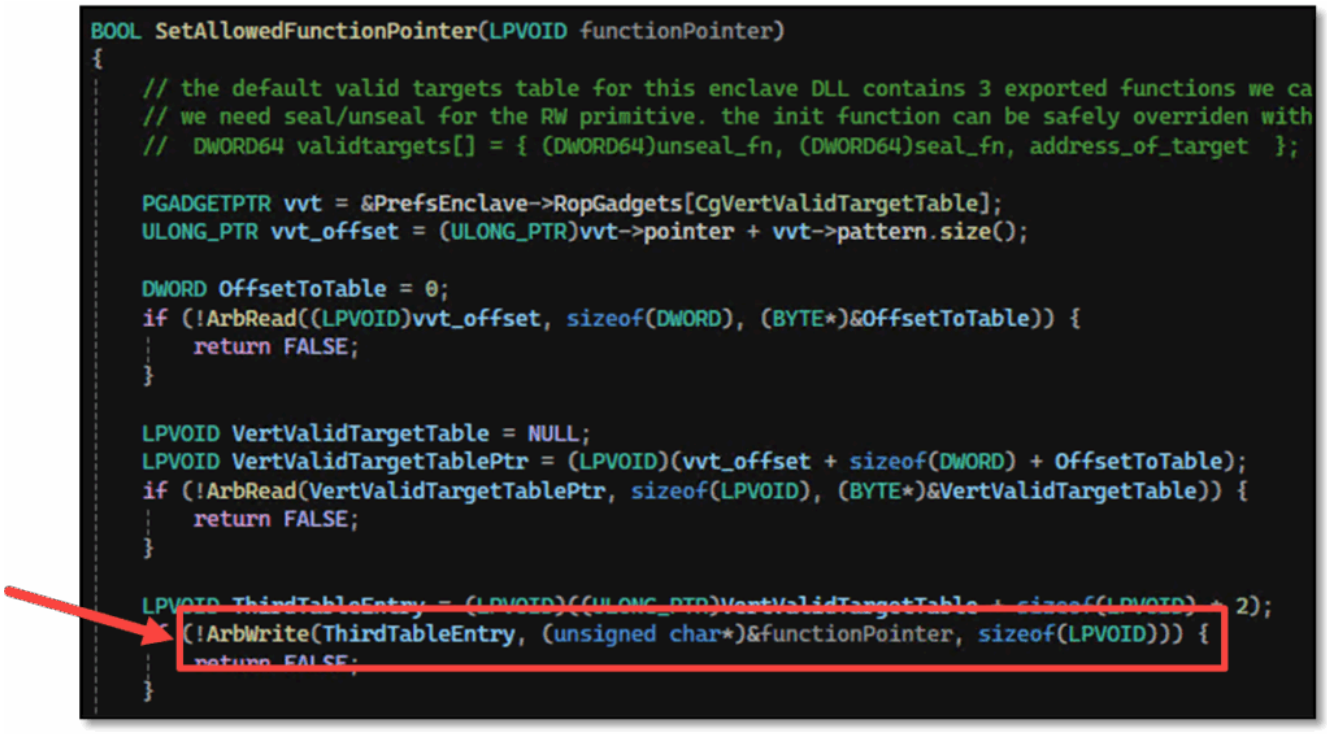
  v6 = a3;
  Key = a1;
  if ( (NtCurrentTeb()->SameTebFlags & 0x20) != 0 || (result = VertpThreadInit(), v7 = result, (int)result >= 0) )
  {
    if ( (_DWORD)VertValidTargetCount
      && bsearch(&Key, VertValidTargetTable, (unsigned int)VertValidTargetCount, 8uLL, VertpCompareTargets) )
    {
      *a3 = Key(*a3);
      return 0LL;
    }
    else
    {
      return 3221225594LL;
```

Valid target functions

Call the function if found

To allow other functions to be called we could use the arbitrary write primitive to patch the `VertValidTargetTable` to include functions of our choice. This also works for functions located in secondary enclave images. In case of the preferences enclave, we could patch the enclave's `Init` function as it is no longer needed after initialization; only the `SealSettings`

and `UnsealSettings` functions. The target table is located in the `.data` section of vertdll and remains writable.



```
BOOL SetAllowedFunctionPointer(LPVOID functionPointer)
{
    // the default valid targets table for this enclave DLL contains 3 exported functions we ca
    // we need seal/unseal for the RW primitive. the init function can be safely overriden with
    //  DWORD64 validtargets[] = { (DWORD64)unseal_fn, (DWORD64)seal_fn, address_of_target  };

    PGADGETPTR vvt = &PrefsEnclave->RopGadgets[CgVertValidTargetTable];
    ULONG_PTR vvt_offset = (ULONG_PTR)vvt->pointer + vvt->pattern.size();

    DWORD OffsetToTable = 0;
    if (!ArbRead((LPVOID)vvt_offset, sizeof(DWORD), (BYTE*)&OffsetToTable)) {
        return FALSE;
    }

    LPVOID VertValidTargetTable = NULL;
    LPVOID VertValidTargetTablePtr = (LPVOID)(vvt_offset + sizeof(DWORD) + OffsetToTable);
    if (!ArbRead(VertValidTargetTablePtr, sizeof(LPVOID), (BYTE*)&VertValidTargetTable)) {
        return FALSE;
    }

    LPVOID ThirdTableEntry = (LPVOID)((ULONG_PTR)VertValidTargetTable + sizeof(LPVOID) * 2);
    if (!ArbWrite(ThirdTableEntry, (unsigned char*)&functionPointer, sizeof(LPVOID))) {
        return FALSE;
    }
}
```

Alternatively, to gain RIP control, we could theoretically also attempt to patch the return address on the stack. This would require first leaking the stack pointer, then overwriting the return address during an `UnsealSettings` call. The `VertValidTargetTable` approach on the other hand has the benefit that we can call arbitrary functions but also return cleanly to VTL0 without corrupting the stack.

Enclave Arbitrary Function Call

Now that we know how to patch the `VertValidTargetTable` using our arbitrary write primitive to gain an arbitrary function call capability, we can use `CallEnclave` to invoke any function pointer within the enclave. There's an important restriction though; enclave functions can only have a single parameter (which could be a pointer to a structure with more parameters though). This restriction also applies to our arbitrary function call primitive.

```
C++

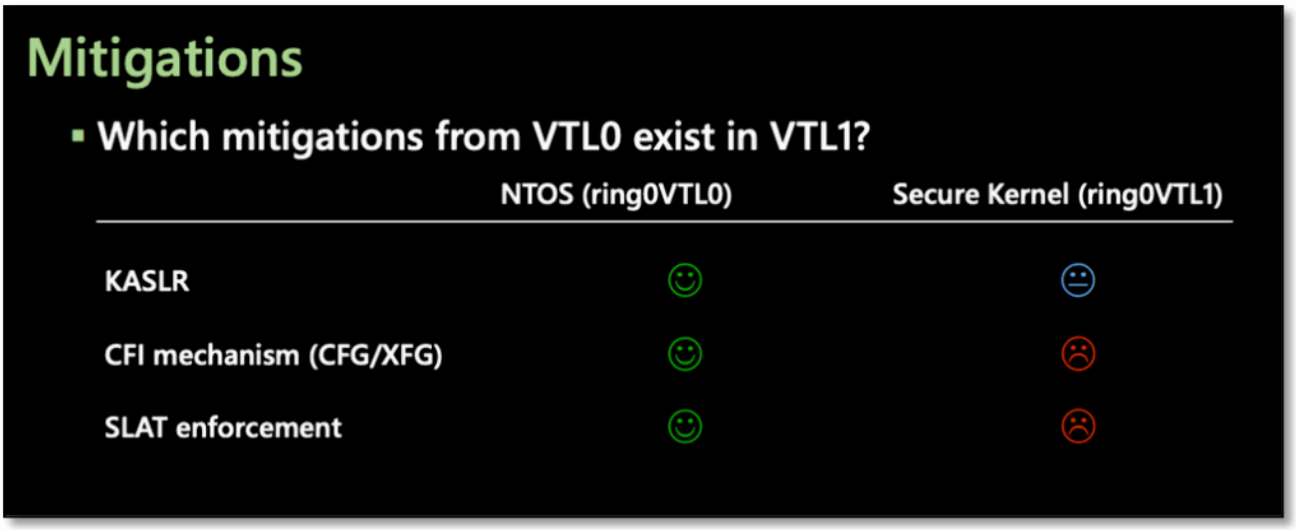
BOOL CallEnclave(
  [in]  LPENCLAVE_ROUTINE lpRoutine,
  [in]  LPVOID            lpParameter,
  [in]  BOOL              fWaitForThread,
  [out] LPVOID            *lpReturnValue
);
```



Luckily, `NtContinue` is also implemented on the VTL1 side, which means we can benefit from context hijacking. The x64 ABI uses a four-register, fast-call calling convention by default, which means that the first 4 arguments to a function call are stored in `rcx`, `rdx`, `r8`, `r9`. We can restore the state of these registers (and `rip`) by passing a custom `CONTEXT` structure to `NtContinue`, which allows an arbitrary function to be called with up to 4 arguments. For more arguments, we'd have to put them on the stack. To do this, we first need to call `RtlCaptureContext` to capture the VTL1 thread context, which conveniently only requires a single argument. Then modify the returned `CONTEXT` structure, and pass it to `NtContinue`.

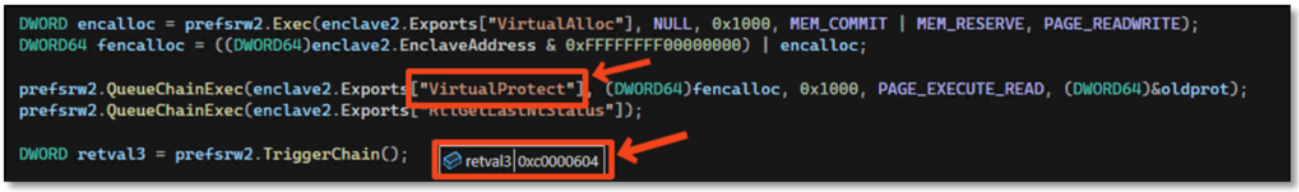
Note that enclaves can be initialized with multiple threads. Using a `CONTEXT` structure from one thread with `NtContinue` on another will crash the enclave. In practice, this isn't a major obstacle. Based on our experience, VTL1 threads tend to be scheduled in a round-robin fashion when enclave calls are made consecutively from VTL0 in a non-threaded context. That said, you really only need a single thread to get the job done.

Interestingly, even for processes with CET RIP validation enabled, we were unencumbered in calling arbitrary functions via `NtContinue` in VTL1. While CFG/XFG stubs were present in vertdll, we could also call arbitrary ROP gadgets in any primary and secondary enclave DLL. In a way, once you've got code execution in VTL1, there seem to be fewer security mechanisms in place.



BlackHat USA 2020 – “Breaking VSM by Attacking Secure Kernel” by Saar Amar and Daniel King

Now that we can call arbitrary VTL1 functions with more arguments, let’s try using `VirtualAlloc` to allocate some RWX memory. This is a function that’s also exported by vertdll. However, when we attempt to allocate RWX memory inside the enclave, the call fails. The reason is Arbitrary Code Guard (ACG) kicking in, which prevents dynamic allocation of executable memory. Ori’s blog post has some more information on how this is enforced in secure kernel. Note that this approach might work in case a debuggable enclave was shipped, which disables the ACG checks.



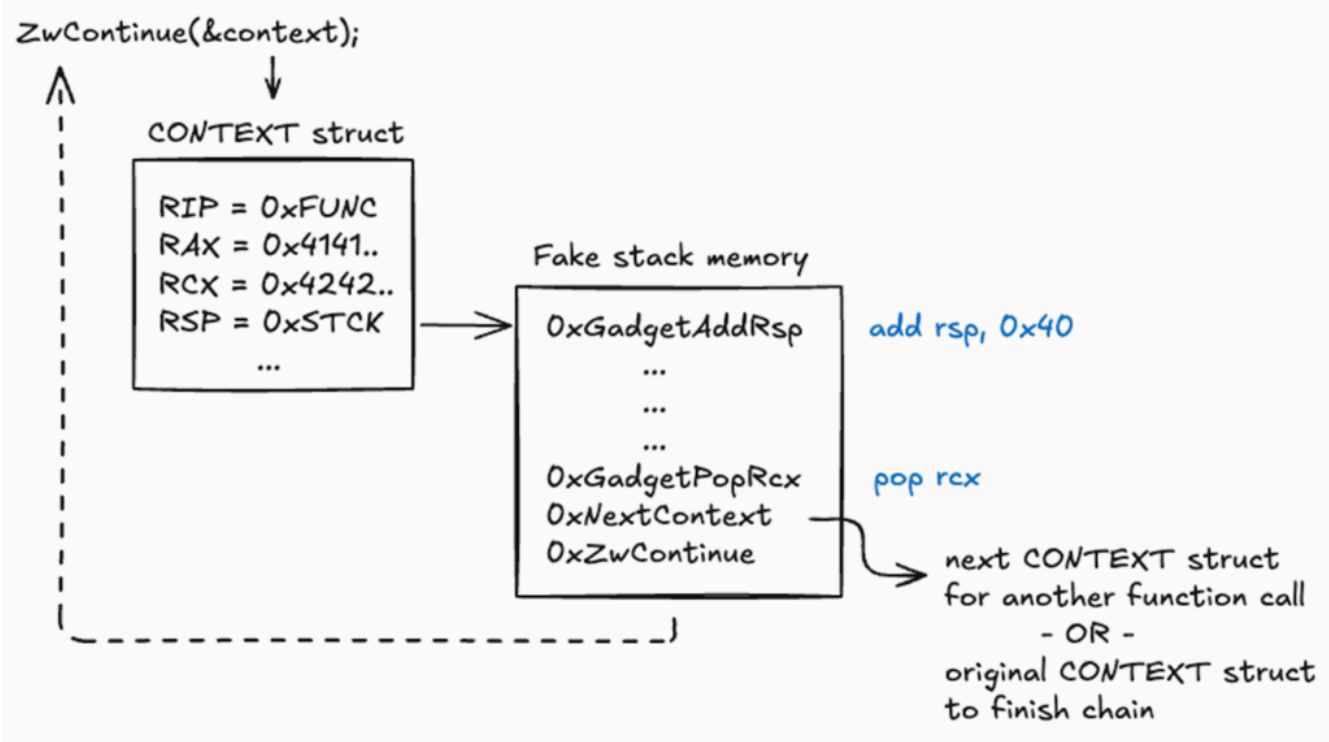
`0xC0000604`: The operation was blocked as the process prohibits dynamic code generation.

ROP chaining

If we want to execute more code in VTL1 without returning control to VTL0, we can chain ROP gadgets by repeatedly invoking `NtContinue`. Since the return address for a function call is stored on the stack, we can point it to a fake stack that contains a sequence of additional ROP gadgets. Of course, the exact gadgets depend on what’s available in the enclave DLLs. On this crafted stack, we’ll need:

- A gadget to advance the stack to ‘jump over’ shadow space (at least 32 bytes).
- A gadget to pop the next `CONTEXT` structure pointer into `rcx`.
- The `CONTEXT` structure pointer to be popped into `rcx`.
- A pointer to `NtContinue`, which will receive its first argument (the `CONTEXT` struct pointer) via the `rcx` register.

We can repeat this process as many times as we need to call the functions we want for a full ROP chain. If we want to finish up the chain (without crashing the enclave), we can point the final `CONTEXT` structure to the original `CONTEXT` with the original unharmed stack.



While this allows us to chain VTL1 function calls, the APIs available to us in an enclave are quite restricted (e.g. no network communication, no file handling). If we want to do something more meaningful (e.g. allocating or freeing VTL0 implant memory) from VTL1, we could chain a call to `CallEnclave` in VTL1 to transition to VTL0.

Transitioning back to VTL0 presents us with the same restriction as we had before: we can only pass a single argument. We can repeat the same trick: use context hijacking with `NtContinue` in VTL0 to expand the number of arguments. This means we first need to capture a valid `CONTEXT` for the VTL0 thread too. When transitioning from VTL1→VTL0 we ran into stack validation/corruption issues with the fake stack that was used by the ROP chain, and ended up duplicating the original VTL1 stack (4KB) for each call.

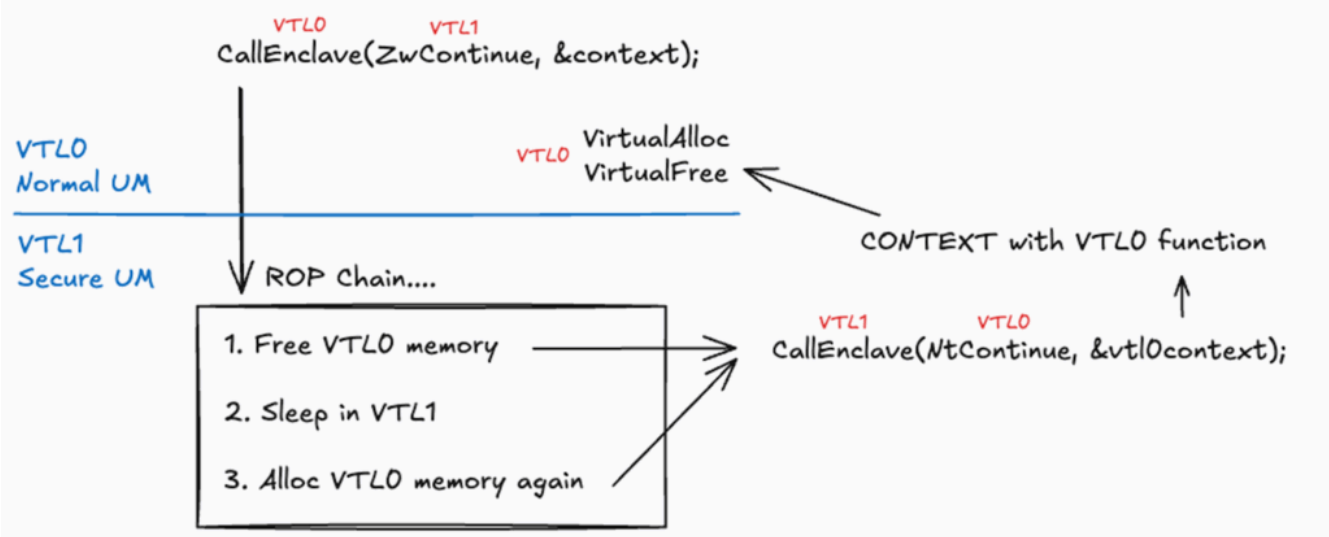
For a VTL1 sleepmask, we first ensure that we load the entire implant shellcode inside the VTL1 enclave upon initialization. Then we make a VTL1 ROP chain that:

- Frees the VTL0 implant memory

  $\texttt{CallEnclave}^{\text{VTL1}} \rightarrow \texttt{NtContinue}^{\text{VTL0}} \rightarrow \texttt{VirtualFree}^{\text{VTL0}}$

- Performs a sleep in VTL1
  - `WaitForAddress`^{VTL1}
  - Using a timeout, as there is no `Sleep` function, and signalling between VTLs does not work.
- Allocates the VTL0 memory again (let's assume RWX)

  `CallEnclave`^{VTL1} → `NtContinue`^{VTL0} → `VirtualAlloc`^{VTL0}
- Copies the memory over to VTL0 via VTL1 `memcpy`
  - `memcpy`^{VTL1}
  - The containing process memory is available to the enclave.



While the VTL1 thread is sleeping, the execution of the associated VTL0 thread blocks. At this point, its VTL0 user-mode thread stack is "inexistent". The VTL0 implant allocation is gone, and the VTL1 allocation cannot be inspected.

Stitching all of that together, we have a working VTL1 ROP chain that performs sleepmasking of our implant shellcode.

Outflank C2 implant sleepmask

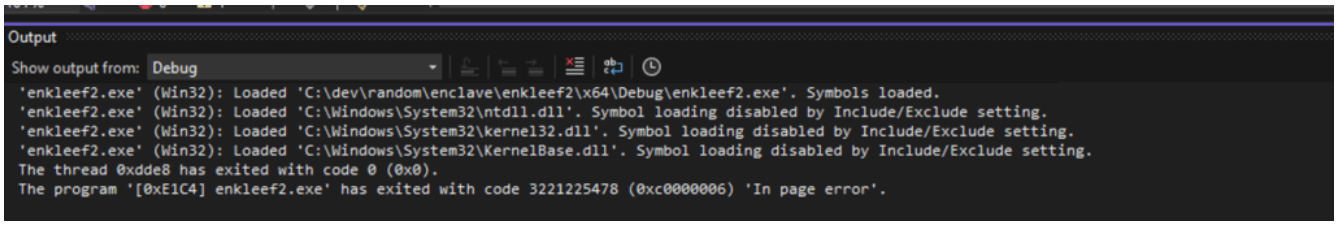
For the Outflank C2 sleepmask implementation, we implemented a custom enclave DLL loader that allows to load the primary image from memory and closes image mappings as soon as we no longer need them. The loader also performs exports parsing and gadget collection so these can be used in the ROP chain later on. This sleepmask is a full VTL1 ROP chain that decommits the implant code, sleeps in VTL1 and resumes the implant execution after returning from VTL1 sleep.

More Enclave Experiments

As we had the setup ready anyway, we played around a bit more with enclaves, resulting in a bunch of failed experiments. Nonetheless, we wanted to share some of our findings. Enclaves are a feature that are in constant flux. Since this research was performed, several new enclave functionalities were released. For example, the EnclaveRestrictContainingProcessAccess function was added, which can restrict an enclave from accessing the address space of the containing process.

Loading multiple enclaves in the same process

We tried loading multiple enclaves in the same process and tried to access enclave memory from the other enclave. This throws an “In page error” and we didn’t investigate further. Note that an enclave is allowed to access containing process memory by default, but apparently not the memory of other enclaves (unless restricted by the new `EnclaveRestrictContainingProcessAccess` API).



Loading a secondary image as if it were a primary image

It is not possible to load a secondary enclave image such as vertdll as if it were a primary enclave image, which is verified based on its `IMAGE_ENCLAVE_CONFIG`. You can also not load additional secondary DLLs (even if they are valid) into the enclave when they are not expected by the primary image.

Even though secondary images have exports, you can also not call those through CallEnclave, even if you have the correct offset as allowed pointers are verified by the `VertValidTargetTable`.

Loading extra data into the enclave

Supposedly, SGX enclaves used to allow loading enclaving data via `LoadEnclaveData`. We weren’t able to use this API to load data into a VBS enclave (except for enclave DLLs via the underlying `NtLoadEnclaveData`).

Keep the enclave thread doing something in the background

We tried several ways of keeping the vulnerable enclave doing something in the background (unrelated to the `CallEnclave` call).

`CallEnclave` has a parameter `fWaitForThread`. This parameter doesn't allow you to have something execute in VTL1 that is disconnected from the execution flow in VTL0. If set to `TRUE`, it simply waits for an enclave thread to become available. If set to `FALSE` the function fails if no thread is available. Similarly, `TerminateEnclave` (for ending execution of enclave threads) has a parameter `fWait` to wait for the threads to finishing executing, but we weren't able to keep something running in the enclave after signalling thread termination.

While we've seen `DLL_THREAD_ATTACH` and `DLL_PROCESS_ATTACH` events, the corresponding `DETACH` events weren't raised upon termination, so they didn't serve as a place to hook into.

When multiple threads are launched in VTL0, and a VTL1 enclave thread is executing a continuous loop, forcibly terminating the parent VTL0 thread will also result in the termination of the linked VTL1 thread.

Using CallEnclave in VTL1 to call VTL1 functions

Some enclaves (e.g. SFAPE.dll) also allow to set VTL0 callback functions to be called from VTL1. These VTL0 functions will be invoked via the `CallEnclave` API implemented in VTL1. It is not possible to make the VTL1 `CallEnclave` execute arbirary VTL1 functions. However, you can use them for some quirky indirect VTL0 function calling.



Side note: when passing a function pointer to VTL0's `CallEnclave` that is located within VTL0 instead of VTL1, the VTL0 function is executed indirectly – a technique implemented in the [FlavorTown CallEnclave] example to call a shellcode pointer. Conversely, VTL1's `CallEnclave` will not allow you to call an arbitrary VTL1 pointer.

Using system calls in VTL1

From a vulnerable enclave we can invoke VTL1 system calls by using a ROP gadget that points to a `syscall` instruction. Using a small IDAPython snippet we can enumerate the available traditional syscalls as well as IUM syscalls. The latter are services that the secure kernel exposes to IUM applications.

```
import idaapi
import idc

syscall_disp_table_ea = idc.get_name_ea_simple('IumSyscallDispEntries')

print(f"[+] Dumping SK syscall entries")

i = 0
while True:
    syscall_entry = idaapi.get_qword(
```

```
        syscall_disp_table_ea + i*8)

    syscall_number = idaapi.get_qword(
        syscall_disp_table_ea + i*8 + 8)

    if syscall_entry == 0:
        break

    entry_name = idc.get_name(syscall_entry)
    print(f"[*] Syscall 0x{syscall_number:X}: {entry_name}")

    i+=2

print(f"[+] Dumping SK secure syscall entries")

secureservicetable_ea = idc.get_name_ea_simple(
    'SkiSecureServiceTable')

secureservicetable_len_ea = idc.get_name_ea_simple(
    'SkiSecureServiceLimit')

sst_len = idaapi.get_dword(secureservicetable_len_ea)
SSK_start = 0x8000000

for i in range(sst_len):
    ium_func = idaapi.get_name(idaapi.get_qword(
        secureservicetable_ea + i*8))

    print(f"[*] Seccall 0x{SSK_start+i:X} {ium_func}")
```

```
[+] Dumping SK syscall entries
[*] Syscall 0x1: NtWorkerFactoryWorkerReady
[*] Syscall 0x4: NtWaitForSingleObject
[*] Syscall 0xA: NtReleaseSemaphore
[*] Syscall 0xD: NtSetInformationThread
[*] Syscall 0xE: NtSetEvent
[*] Syscall 0xF: NtClose
[*] Syscall 0x12: NtOpenKeyEx
[*] Syscall 0x18: NtAllocateVirtualMemory
[*] Syscall 0x19: NtQueryInformationProcess
[*] Syscall 0x1C: NtSetInformationProcess
[*] Syscall 0x1E: NtFreeVirtualMemory
[*] Syscall 0x21: NtQueryInformationToken
[*] Syscall 0x23: NtQueryVirtualMemory
[*] Syscall 0x24: NtOpenThreadToken
[*] Syscall 0x25: NtQueryInformationThread
[*] Syscall 0x28: NtMapViewOfSection
[*] Syscall 0x2A: NtUnmapViewOfSection
[*] Syscall 0x2C: NtTerminateProcess
[*] Syscall 0x31: NtQueryPerformanceCounter
[*] Syscall 0x36: NtQuerySystemInformation
[*] Syscall 0x3C: NtDuplicateObject
[*] Syscall 0x3E: NtClearEvent
[*] Syscall 0x43: NtContinue
[*] Syscall 0x48: NtCreateEvent
[*] Syscall 0x49: NtQueryVolumeInformationFile
[*] Syscall 0x4A: NtCreateSection
[*] Syscall 0x50: NtProtectVirtualMemory
[*] Syscall 0x53: NtTerminateThread
[*] Syscall 0x70: NtAlertThreadByThreadId
[*] Syscall 0x76: NtAllocateVirtualMemoryEx
[*] Syscall 0x77: NtAlpcAcceptConnectPort
[*] Syscall 0x79: NtAlpcConnectPort
[*] Syscall 0x89: NtAlpcQueryInformation
[*] Syscall 0x8C: NtAlpcSendWaitReceivePort
[*] Syscall 0x8D: NtAlpcSetInformation
[*] Syscall 0x90: NtAssociateWaitCompletionPacket
[*] Syscall 0x91: NtCallEnclave
[*] Syscall 0xA3: NtContinueEx
[*] Syscall 0xC5: NtCreateSemaphore
[*] Syscall 0xC7: NtCreateThreadEx
[*] Syscall 0xD3: NtCreateWorkerFactory
[*] Syscall 0xFA: NtGetCurrentProcessorNumber
[*] Syscall 0xFB: NtGetCurrentProcessorNumberEx
[*] Syscall 0x129: NtOpenKeyEx
[*] Syscall 0x14A: NtQueryDebugFilterState
[*] Syscall 0x16C: NtQuerySystemInformationEx
[*] Syscall 0x172: NtRaiseException
[*] Syscall 0x17C: NtReleaseWorkerFactoryWorker
[*] Syscall 0x1AE: NtSetInformationWorkerFactory
[*] Syscall 0x1CD: NtSystemDebugControl
[*] Syscall 0x1CE: NtTerminateEnclave
[*] Syscall 0x1D3: NtTraceControl
[*] Syscall 0x1DC: NtUnmapViewOfSectionEx
[*] Syscall 0x1E0: NtWaitForAlertByThreadId
[*] Syscall 0x1E3: NtWaitForWorkViaWorkerFactory

[+] Dumping SK secure syscall entries
[*] Seccall 0x8000000 IumEmitSmc
[*] Seccall 0x8000001 IumEmitSmc
```

```
[*] Seccall 0x8000002 IumCreateSecureDevice
[*] Seccall 0x8000003 IumCreateSecureSection
[*] Seccall 0x8000004 IumCreateSecureSectionSpecifyPages
[*] Seccall 0x8000005 IumCrypto
[*] Seccall 0x8000006 IumDmaMapMemory
```

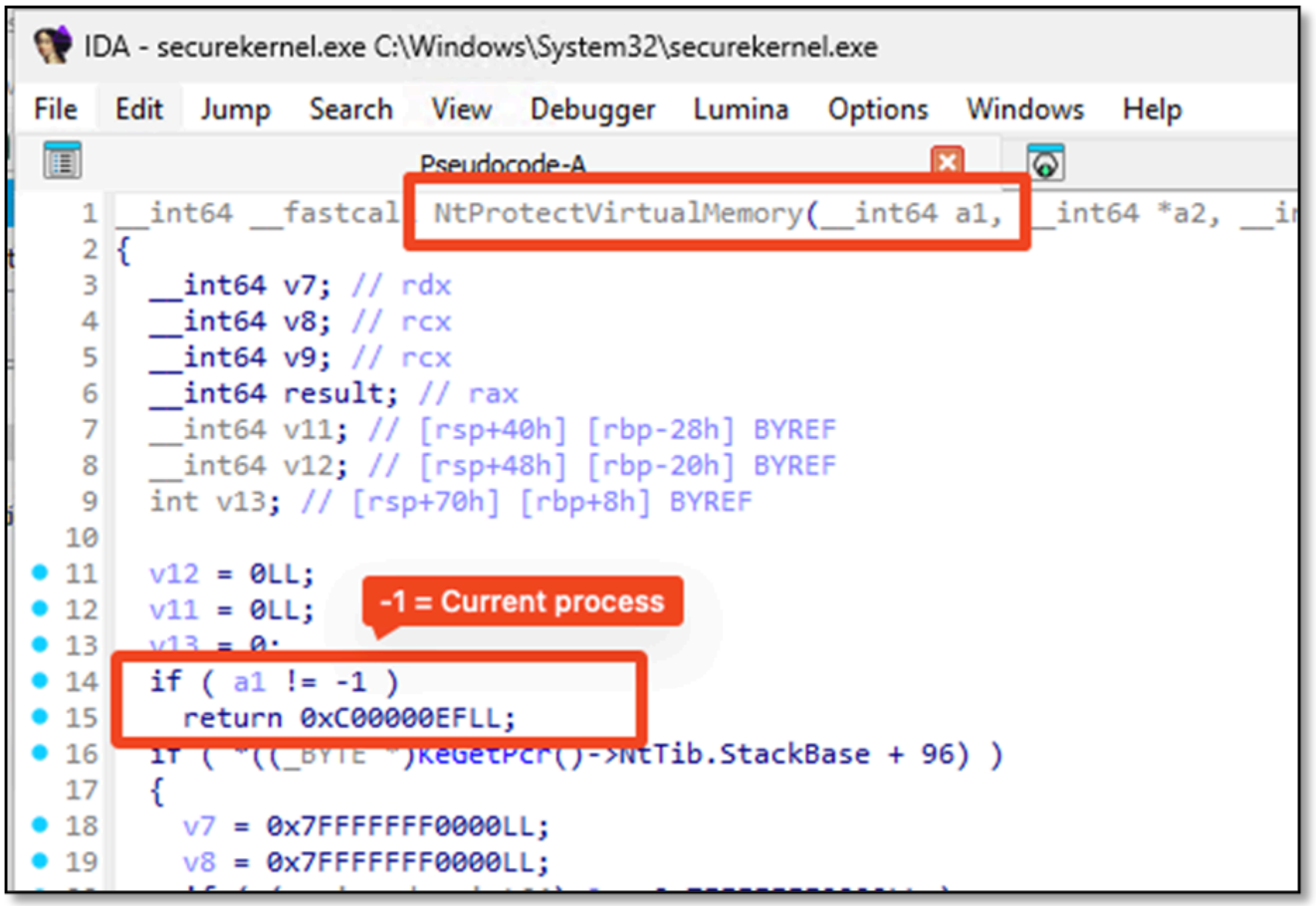
However, only a limited number of these syscalls can really be called from an enclave. IUM syscalls are additionally mediated via `IumSyscallDescriptor` in secure kernel. Below is an overview of syscalls that didn't immediately return an error.

The remaining syscalls that we're allowed to invoke are also more restricted versions of their normal kernel counterpart. For example, we wanted to check if we could use SK system calls to perform actions on other processes. However, SK syscalls check whether the first parameter refers to the current process (`-1`) and don't even implement interaction with other processes.

Allowed from enclave

```
[*] Syscall 0xD: NtSetInformationThread
[*] Syscall 0x18: NtAllocateVirtualMemory
[*] Syscall 0x19: NtQueryInformationProcess
[*] Syscall 0x1E: NtFreeVirtualMemory
[*] Syscall 0x23: NtQueryVirtualMemory
[*] Syscall 0x25: NtQueryInformationThread
[*] Syscall 0x2C: NtTerminateProcess
[*] Syscall 0x36: NtQuerySystemInformation
[*] Syscall 0x43: NtContinue
[*] Syscall 0x50: NtProtectVirtualMemory
[*] Syscall 0x70: NtAlertThreadByThreadId
[*] Syscall 0x76: NtAllocateVirtualMemoryEx
[*] Syscall 0x91: NtCallEnclave
[*] Syscall 0xA3: NtContinueEx
[*] Syscall 0xFA: NtGetCurrentProcessorNumber
[*] Syscall 0xFB: NtGetCurrentProcessorNumberEx
[*] Syscall 0x172: NtRaiseException
[*] Syscall 0x1CE: NtTerminateEnclave
[*] Syscall 0x1E0: NtWaitForAlertByThreadId

[*] Seccall 0x8000005 IumCrypto
```



Outro

That's all folks! In this blog post, we've outlined an exploitation strategy for vulnerable enclaves, demonstrating how to transform an arbitrary write primitive into ROP-based code execution. If you're a bit intimidated by the amount of text in this post, then check out the video recording of our talk on this topic.

This research was integrated in our OST tools (Beacon Booster and Outflank C2) as a sleepmask to keep implant code hidden during sleep. If you're not an OST customer but are interested in learning more about how our toolset can help boost your offensive operations, we recommend scheduling an expertled demo to learn more.