Virtual Secure Mode: Communication Interfaces

Aleksandar Milenkoski
amilenkoski@ernw.de✉

This work is part of the *Windows Insight* series. This series aims to assist efforts on analysing inner working principles, functionalities, and properties of the Microsoft Windows operating system. For general inquiries contact Aleksandar Milenkoski (amilenkoski@ernw.de) or Dominik Phillips (dphillips@ernw.de). For inquiries on this work contact the corresponding author (✉).

The content of this work has been created in the course of the project named 'Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10 (SiSyPHuS Win10)' (ger.) - 'Study of system design, logging, hardening, and security functions in Windows 10' (eng.). This project has been contracted by the German Federal Office for Information Security (ger., Bundesamt für Sicherheit in der Informationstechnik - BSI).

Required Reading

In addition to referenced work, related work focussing on Windows Architecture and Virtual Secure Mode (VSM), part of the *Windows Insight* series, are relevant for understanding concepts and terms mentioned in this document.

Technology Domain

The operating system in focus is Windows 10, build 1607, 64-bit, long-term servicing branch (LTSB).

1 Introduction

A VSM-enabled Windows environment implements multiple communication interfaces:

- Isolated User Mode (IUM) system calls: Interface between IUM applications and the secure kernel, where the secure kernel provides services to IUM applications (Section 2);
- normal-mode services: Interface between the secure and the normal kernel, where the normal kernel provides services to the secure kernel (Section 5);
- secure services: Interface between the secure and the normal kernel, where the secure kernel provides services to the normal kernel (Section 4); and
- hypercalls: Interface between the normal and the secure kernel, and the hypervisor, where the hypervisor provides services to the normal and the secure kernel (Section 3).

In addition to the interfaces mentioned above, there is the traditional non-VSM-specific system call interface enabling communication between user applications and the normal kernel. Section 2 focuses on the execution path of IUM system calls, comparing it with that of traditional system calls.

2 IUM System Calls

IUM system calls implement services that the secure kernel exposes to IUM applications. This includes critical system services enabling the operation of IUM applications, such as memory management services.

The execution path of IUM system calls is conceptually identical to that of traditional system calls. An IUM application executes IUM system calls by invoking functions implemented in the *IUMDLL.dll* library file. These functions have names starting with *Ium* and implement execution context switching between IUM applications and the secure kernel. Figure 1 depicts the implementation of the *IumPostMailbox* IUM system call in *IUMDLL.dll*. For comparison purposes, Figure 2 depicts the implementation of the traditional system call *NtCreateUserProcess* in the *NTDLL.dll* library file.

```
1  iumdll!IumPostMailbox:
2  mov     r10,rcx
3  mov     eax,800000Ah
4  syscall
5  ret
```

Figure 1: Implementation of IumPostMailbox in IUMDLL.dll

```
1  ntdll!NtCreateUserProcess:
2  mov     r10,rcx
3  mov     eax,0C1h
4  test    byte ptr [SharedUserData+0x308],1
5  [...]
6
7  ntdll!NtCreateUserProcess+0x12:
8  syscall
9  ret
10 [...]
```

Figure 2: Implementation of NtCreateUserProcess in NTDLL.dll

Same as traditional system calls, each IUM system call can be uniquely identified by a system service index. Indexes specifying IUM system calls have the highest bit set. An example is *0x800000A*, a system service index specifying the *IumPostMailbox* IUM system call (see line 3 in Figure 1). Once the system service index is set, the syscall instruction switches the execution context to the secure kernel (see line 4 in Figure 1 and line 8 in Figure 2).

Once the execution context is switched to the secure kernel, it invokes the *KiSystemCall64* routine. The address of this function is stored in the model-specific register (MSR) *0xC0000082* when the *ShvlpInitProcessor* function is invoked (see Figure 3). This function is invoked during the initialization of the secure kernel. For comparison purposes, Figure 4 depicts the use of the model-specific register (MSR) *0xC0000082* for the same purpose in the context of traditional system calls. The *KiInitializeBootStructures* is invoked during the initialization of the normal kernel.

```
ShvlpInitProcessor( [...] )
{
    [...]
    __writemsr(0xC0000082, KiSystemCall64);
    [...]
}
```

Figure 3: MSR 0xC0000082 storing KiSystemCall64 (secure kernel)

KiSystemCall64 routes invocations of a given IUM system call to the corresponding handler function implementing the actual service functionality. Figure 5 depicts the implementation of *KiSystemCall64*. This function first evaluates whether the highest bit of the system service index is set (see line 5 in Figure 5). If set, *KiSystemCall64* loads the address of the kernel structure *SkiSecureServiceTable* (see line 13 in Figure 5). This structure is an array of functions implementing the service functionalities of IUM system calls. Figure 6 depicts the implementation of *SkiSecureServiceTable*. After loading*SkiSecureServiceTable*, *KiSystemCall64* executes the handler

```
KiInitializeBootStructures( [...] )
{
    [...]
    __writemsr(0xC0000082, KiSystemCall64);
    [...]
}
```

Figure 4: MSR 0xC0000082 storing KiSystemCall64 (normal kernel)

```
1  securekernel!KiSystemCall64:
2  [...]
3  mov     ebx,eax
4  and     eax,0FFFh
5  test    ebx,8000000h
6  je      securekernel!KiSystemServiceStart+0x28
7
8  securekernel!KiSystemServiceStart+0x13:
9  cmp     eax,dword ptr [securekernel!SkiSecureServiceLimit]
10 jae     securekernel!KiSystemServiceExit+0x145
11
12 securekernel!KiSystemServiceStart+0x1f:
13 lea     r10,[securekernel!SkiSecureServiceTable]
14 jmp     securekernel!KiSystemServiceStart+0x3b
15
16 securekernel!KiSystemServiceStart+0x28:
17 lea     r10,[securekernel!IumSyscallDispatchTable]
18 cmp     eax,dword ptr [securekernel!IumSyscallDescriptorLimit]
19 jae     securekernel!KiSystemServiceExit+0x145
20
21 securekernel!KiSystemServiceStart+0x3b:
22 movsxd  r11,dword ptr [r10+rax*4]
23 mov     rax,r11
24 sar     r11,4
25 add     r10,r11
26 nop     dword ptr [rax]
27 and     eax,0Fh
28 sub     eax,4
29 jle     securekernel!KiSystemServiceCopyEnd
30 [...]
31 securekernel!KiSystemServiceCopyEnd:
32 mov     eax,ebx
33 call    r10
34 [...]
```

Figure 5: The implementation of KiSystemCall64

function indexed by the system service index identifying the invoked IUM system call (see line 22 and line 33 in Figure 5).

If the highest bit of the system service index is not set, that is, if the evaluation at line 5 in Figure 5 fails, the *KiSystemCall64* routes an invocation of a normal-mode service. Section 5 discusses normal-mode services.

Since only IUM applications may invoke IUM system calls, the conditions for instantiating IUM applications by third parties apply also to invoking IUM system calls by these parties.

3 Hypercalls

Hypercalls implement services that the hypervisor exposes to partitions. This involves critical system services enabling the operation of virtual systems, such as memory management services. The hypercalls implemented by the Hyper-V hypervisor are listed in ([Mic17], Appendix A). Each Hyper-V hypercall can be uniquely identified by an identification number, referred to as a call code.

Partitions can invoke hypercalls only from kernel-mode. In a VSM-enabled Windows environment, this includes the execution context of the normal and the secure kernel. The *winhv* and *winhvr* drivers implement wrapper functions enabling the straightforward invocation of hypercalls. For example, the functions implement assignment of call codes and management of hypercall input and output values. The activities that need to be performed for a Hyper-V hypercall to be executed are documented in ([Mic17], Section 3).

```
[...]

TABLERO:0000000140089000 SkiSecureServiceTable dq offset IumCreateSecureDevice
TABLERO:0000000140089000                                        ; DATA XREF: SkiSystemStartup+D6?o
TABLERO:0000000140089000                                        ; KiSystemCall64+C3?o
TABLERO:0000000140089008                 dq offset IumCreateSecureSection
TABLERO:0000000140089010                 dq offset IumCrypto
TABLERO:0000000140089018                 dq offset IumDmaMapMemory
TABLERO:0000000140089020                 dq offset IumFlushSecureSectionBuffers
TABLERO:0000000140089028                 dq offset IumGetDmaEnabler
TABLERO:0000000140089030                 dq offset IumGetExposedSecureSection
TABLERO:0000000140089038                 dq offset IumGetIdk
TABLERO:0000000140089040                 dq offset IumMapSecureIo
TABLERO:0000000140089048                 dq offset IumOpenSecureSection
TABLERO:0000000140089050                 dq offset IumPostMailbox
TABLERO:0000000140089058                 dq offset IumProtectSecureIo
TABLERO:0000000140089060                 dq offset IumQuerySecureDeviceInformation
TABLERO:0000000140089068                 dq offset IumSecureStorageGet
TABLERO:0000000140089070                 dq offset IumSecureStoragePut
TABLERO:0000000140089078                 dq offset IumUnmapSecureIo
TABLERO:0000000140089080                 dq offset IumUpdateSecureDeviceState

[...]
```

Figure 6: The implementation of SkiSecureServiceTable

A crucial prerequisite for Hyper-V hypercalls to be invoked is the existence of the hypercall page in the context of the partition. A hypercall page is a memory page that stores code for invoking hypercalls as per the Hyper-V specification. This page is exposed by the hypervisor to each partition. During the initialization process, each partition reserves a memory page and stores its guest physical address (GPA) in the MSR *0x40000001* ([Mic17], Section 3.13). Hyper-V then populates this page with code. A populated hypercall page cannot be modified in order to prevent unauthorized modifications of the code stored in it. Figure 7 depicts the contents of a MSR *0x40000001*.

```
[...]

kd> rdmsr 0x40000001
msr[40000001] = 00000000`0020e003

[...]
```

Figure 7: The contents of a MSR 0x40000001

When the hypercall page is loaded in the context of a partition, the kernel running in the partition can invoke hypercalls. This typically involves activities such as loading the hypercall page, allocating memory buffers for hypercall input and output values, and setting these values. Finally, the code stored in the hypercall page is executed so that the execution context is switched to the hypervisor. For example, the *WinHvpHypercall* function of the *winhvr* driver results in the execution of code stored in the hypercall page.

Figure 8 depicts the contents of a hypercall page accessed in the *WinHvpHypercall* function. It contains page-aligned code for invoking hypercalls, padded with "no operation" (*nop*) instructions. The page contains several sets of instructions ending with the instruction sequence "*vmcall ret*". The vmcall instruction is implemented in Intel processors and it switches execution context to the hypervisor.

The sets of instructions stored in the hypercall page can be understood as trampolines for abstracting the switching of execution context to the hypervisor in different scenarios. These trampolines accommodate the execution of any hypercall, and of the hypercalls with call codes *0x11* and *0x12*, on both 32-bit and 64-bit platforms. The instructions preceding the *vmcall* instructions (if any) save the contents of the *eax*, or the *rcx*, register and store a hypercall call code in this register. The *eax*, or the *rcx*, register stores a hypercall call code on 32-bit and 64-bit platforms, respectively. The use of specific registers for storing hypercall input and output values, as well as call codes, is documented in ([Mic17], Section 3.7) and ([Mic17], Section 3.8).

The sets of instructions in the hypercall page where the values *0x11* and *0x12* are stored in the *eax*, or the

```
[...]

kd> u fffff800`b3948000 L20
fffff800`b3948000 0f01c1          vmcall
fffff800`b3948003 c3              ret
fffff800`b3948004 8bc8            mov     ecx,eax
fffff800`b3948006 b811000000      mov     eax,11h
fffff800`b394800b 0f01c1          vmcall
fffff800`b394800e c3              ret
fffff800`b394800f 488bc1          mov     rax,rcx
fffff800`b3948012 48c7c111000000  mov     rcx,11h
fffff800`b3948019 0f01c1          vmcall
fffff800`b394801c c3              ret
fffff800`b394801d 8bc8            mov     ecx,eax
fffff800`b394801f b812000000      mov     eax,12h
fffff800`b3948024 0f01c1          vmcall
fffff800`b3948027 c3              ret
fffff800`b3948028 488bc1          mov     rax,rcx
fffff800`b394802b 48c7c112000000  mov     rcx,12h
fffff800`b3948032 0f01c1          vmcall
fffff800`b3948035 c3              ret
fffff800`b3948036 90              nop
fffff800`b3948037 90              nop
fffff800`b3948038 90              nop

[...]
```

Figure 8: The contents of a hypercall page

rcx, register are used for invoking the hypercalls with call codes *0x11* and *0x12*. These hypercalls are used for invoking normal-mode and secure services. Section 4 and Section 5 discuss these services. The first set of instructions, containing only the "*vmcall ret*" instruction sequence, is used for invoking any other hypercall.

When the vmcall instruction is executed, the execution context is switched to Hyper-V. The hypervisor then performs access control checks. If a given hypercall is protected by access control, the partition invoking it has to possess the required privilege ([Mic17], Section 3.11). If the hypercall is to be executed, Hyper-V loads an array that contains entries of a fixed size. Each entry is indexed by a hypercall call code and contains a pointer to a function implementing the functionality of the hypercall identified by the call code. Hyper-V then executes the function indexed by the call code of the invoked hypercall. After this, the execution context is switched back to the kernel that has invoked the hypercall. Figure 9 depicts a portion of the array containing functions implementing hypercall functionalities indexed by call codes (see, for example, *sub_FFFFF800002129D8* [function] and *5Dh* [call code] in Figure 9). This array is implemented as part of *hvix64.exe*.

```
[...]

CONST:FFFFF80000C008B8          dq offset sub_FFFFF800002129D8
CONST:FFFFF80000C008C0          db  5Dh ; ]
CONST:FFFFF80000C008C1          db    0
CONST:FFFFF80000C008C2          db    0
CONST:FFFFF80000C008C3          db    0
CONST:FFFFF80000C008C4          db    8
CONST:FFFFF80000C008C5          db    0

[...]

CONST:FFFFF80000C008CF          db    0
CONST:FFFFF80000C008D0          dq offset sub_FFFFF800002BF034
CONST:FFFFF80000C008D8          db  5Eh ; ^
CONST:FFFFF80000C008D9          db    0
CONST:FFFFF80000C008DA          db    0
CONST:FFFFF80000C008DB          db    0
CONST:FFFFF80000C008DC          db  10h
CONST:FFFFF80000C008DD          db    0

[...]
```

Figure 9: Functions implementing hypercall functionalities

4 Secure Services

The secure kernel exposes services to the normal kernel, referred to as secure services in this work. They implement security-critical kernel operations that are executed in the secure, isolated environment. For a secure service to be invoked by the normal kernel, the kernel has to switch from Virtual Trust Level (VTL) 0 to VTL 1. This process is known as VTL call. In its essence, a VTL call is an execution context switch from a lower to a higher VTL. ([Mic17], Section 15.6.1) provides details on the VTL call process.

VTL calls are performed by the normal kernel issuing a hypercall with call code *0x11* – the *HvCallVtlCall* hypercall ([Mic17], Section 17). The normal kernel issues *HvCallVtlCall* by invoking the function chain *VslpEnterIumSecureMode* → *HvlSwitchToVsmVtl1* → *HvlpVsmVtlCallVa*. The *VslpEnterIumSecureMode* function is invoked in the functions implemented as part of the normal kernel that require a secure service. *HvlpVsmVtlCallVa* is a variable storing a function referencing the trampoline of the hypercall page for invoking the hypercall with call code *0x11*. Figure 10 depicts this trampoline executed in the *HvlSwitchToVsmVtl1* function.

```
[...]

nt!HvlSwitchToVsmVtl1+0xa3:
fffff802`289cc633 ffd0            call    rax
0: kd> t
fffff802`2873f00f 488bc1          mov     rax,rcx
0: kd> p
fffff802`2873f012 48c7c111000000  mov     rcx,11h
0: kd> p
fffff802`2873f019 0f01c1          vmcall

[...]
```

Figure 10: Issuing a VTL call

Each secure service can be uniquely identified by an identification number, referred to as secure service call number (SSCN). In the context of the normal kernel, a SSCN is specified as the second parameter of *VslpEnterIumSecureMode*. The SSCN is then passed to the secure kernel as part of a data structure stored in the *rdx* register when a VTL call is issued. This structure is referred to as the VTL call data structure in this work. The table presented in the section 'Secure Services' of the Appendix lists the functions implemented as part of the normal kernel (column 'Function') that invoke secure services identified by SSCNs (column ' SSCN').

In addition to secure services, a VTL call supports the specification of other operations that can be executed by the secure kernel. Each operation is uniquely identified by an operation code, which is stored in the VTL call data structure. The operations are:

- managing the execution of a thread relevant to the secure kernel (operation code – *0x0*): Section 5 discusses more on this topic;
- invocation of a secure service (operation code – *0x01*);
- flushing the transaction lookaside buffer (TLB) (operation code – *0x02*): With respect to the design of VSM, the flushing of the TLB is considered a security-critical activity and is therefore executed in the secure environment. The TLB is involved in translations between virtual and physical addresses.

Figure 11 depicts the contents of the VTL call data structure when a VTL call is issued. In Figure 11, *0x01* is an operation code, indicating invocation of a secure service, and *0xD1* is a SSCN.

In the context of the secure kernel, a VTL call is processed in the function *IumInvokeSecureService*, invoked by the *SkCallNormalMode* function. *IumInvokeSecureService* extracts the SSCN from the VTL call data structure and invokes the function(s) implementing the actual secure service identified by the SSCN. The secure kernel then continues the execution of *SkCallNormalMode*. This function invokes the trampoline of the hypercall page for invoking the hypercall with call code *0x12*. This is done for returning relevant data to the normal kernel and switching the execution context back to VTL 0. The hypercall with call code *0x12* is the *HvCallVtlReturn* hypercall ([Mic17], Section 17). It is used for switching from a higher to a lower VTL. This process is opposite to a VTL

```
[...]

5: kd> db @rdx
ffffe201`1cfcfb00  01 00 d1 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb10  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb40  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb60  00 00 00 00 00 00 00 00-4f 40 48 8d 55 38 41 b0  ........O@H.U8A.
ffffe201`1cfcfb70  af e4 b7 2b 8f dd ff ff-57 40 0f b7 48 34 66 89  ...+....W@..H4f.

[...]
```

Figure 11: Contents of the VTL call data structure

call and is referred to as VTL return. ([Mic17], Section 15.7.1) provides details on the VTL return process. In *SkCallNormalMode*, the trampoline for invoking *HvCallVtlReturn* is stored in the *ShvlpVtlReturn* variable (see Figure 12). This variable is populated during the initialization of the secure kernel, in the *ShvlpInitializeVsmCodeArea* function.

```
[...]

.text:000000014005FB95                  mov     cr2, rdx
.text:000000014005FB98                  mov     ecx, 1
.text:000000014005FB9D
.text:000000014005FB9D InvokeReturnHcall:
.text:000000014005FB9D                  call    cs:ShvlpVtlReturn

[...]
```

Figure 12: Execution of the HvCallVtlReturn hypercall in SkCallNormalMode

5 Normal-mode Services

The normal kernel exposes services to the secure kernel, referred to as normal-mode services in this work. These services implement kernel operations that are not implemented by the secure kernel, however, are necessary for this kernel or the IUM applications that it hosts to function. The secure kernel implements only a limited set of security-critical functionalities. This is because this kernel is designed to expose a minimal interface. It has a significantly smaller codebase than the one of the normal kernel. This reduces the risk of breaches due to design or implementation errors.

Example normal-mode services include semaphore and process management, and registry and filesystem input/output. The traditional system calls implemented as part of the normal kernel are invoked as normal-mode services by the secure kernel.

In the context of the secure kernel, normal-mode services that are implemented as system calls in the normal kernel, are invoked by executing functions with names starting with *Nt* or *Zw*. These functions may be invoked by IUM applications requesting kernel functionalities or the secure kernel itself. The functions with names starting with *Zw* invoke the *KiServiceInternal* function. The system service index is stored in the *eax* register. Figure 13 depicts the invocation of *KiServiceInternal* by the function *ZwTerminateProcess* such that the system service index is *0x2C*. This is the index of the *NtTerminateProcess* system call implemented in the normal kernel.

KiServiceInternal invokes *KiSystemServiceStart*, a code segment of the *KiSystemCall64* function (see Figure 5, Section 2). In *KiSystemServiceStart*, the secure kernel loads the variable *IumSyscallDispatchTable* (which is different than the one in the normal kernel). This is because the highest bit of the system service index is set (see line 17 in Figure 5). *IumSyscallDispatchTable* potentially contains pointers to functions implemented as part of the *IumSyscallDispEntries* array. *IumSyscallDispEntries* stores pointers to functions with prefix *Nt*, indexed by a system service index. Figure 14 depicts a portion of the contents of *IumSyscallDispEntries*.

```
ZwTerminateProcess proc near

[...]

mov     eax, 2Ch
jmp     KiServiceInternal
retn
ZwTerminateProcess endp
```

Figure 13: ZwTerminateProcess invoking KiServiceInternal

```
[...]

.data:0000000140078580 IumSyscallDispEntries dq offset NtWorkerFactoryWorkerReady
.data:0000000140078580
.data:0000000140078580
.data:0000000140078588               db    1
.data:0000000140078589               db    0

[...]

.data:000000014007858F               db    0
.data:0000000140078590               dq offset NtWaitForSingleObject
.data:0000000140078598               db    4
.data:0000000140078599               db    0

[...]

.data:000000014007859F               db    0
.data:00000001400785A0               dq offset NtReleaseSemaphore
.data:00000001400785A8               db   0Ah
.data:00000001400785A9               db    0

[...]
```

Figure 14: Contents of IumSyscallDispEntries

After loading *IumSyscallDispatchTable*, *KiSystemServiceStart* invokes the function with prefix *Nt* indexed by the system service index stored in the *eax* register. The functions with prefix *Nt* invoke stubs for executing normal-mode services. These stubs are implemented in functions with names starting with *Nk*. For example, *NtSetEvent* invokes *NkSetEvent*.

Figure 15 depicts the process of executing normal-mode services by functions with prefix *Nk*. Figure 15 depicts the concrete example of *NkTerminateProcess* executing the system call *NtTerminateProcess* as a normal-mode service. *NkTerminateProcess* invokes *IumGenericSyscall* such that the first parameter is a system service index with the highest bit set. *NkTerminateProcess* sets the first parameter of *IumGenericSyscall* to *0x8000002C*. *0X2C* is the system service index of the *NtTerminateProcess* system call implemented in the normal kernel. *IumGenericSyscall* invokes *SkSyscall* such that its first parameter is the system service index (*SysCallID* in Figure 15). *SkSyscall* sets the highest bit of the system service index to *0* (*SysCallID&0x7FFFFFFF* in Figure 15). The system service index is then passed to the *SkCallNormalMode* function as part of a data structure (*param* in Figure 15).

SkCallNormalMode executes a VTL return; that is, it switches from VTL 1 to VTL 0 (see Section 4). *SkCallNormalMode* passes the data structure provided by *SkSyscall* to VTL 0 (*param* in Figure 15). This structure is referred to as the VTL return data structure in this work. *SkCallNormalMode* executes a VTL return by invoking the hypercall with call code *0x12* (see Section 4).

In the context of the normal kernel, normal-mode services requested by IUM applications or by the secure kernel are handled in the *VslpEnterIumSecureMode* function. The *VslpDispatchIumSyscall* function, invoked by *VslpEnterIumSecureMode*, executes normal-mode services implemented as system calls in the normal kernel. The *PsDispatchIumService* function, invoked by *VslpEnterIumSecureMode*, executes other normal-mode services.

VslpDispatchIumSyscall and *PsDispatchIumService* are executed in the context of worker threads. These threads act as agents of entities running in the secure environment for executing normal-mode services. Normal-mode services requested by the secure kernel are executed in the context of a thread owned by the Secure System process. Normal-mode services requested by IUM applications are executed in the context of threads owned by these applications. Figure 16 depicts the invocation of *VslpDispatchIumSyscall* in the context of threads owned

```
NkTerminateProcess ( [...] )
{
    return IumGenericSyscall(0x8000002C, [...] );
}

IumGenericSyscall(SysCallID, [...] )
{
    [...]

    return SkSyscall(SysCallID, [...] );
}

SkSyscall(SysCallID, [...] )
{
    [...]

    SysCallID = SysCallID & 0x7FFFFFFF;

    IumApi_NtGENERIC ( [...] );
    IumApi_NtGENERIC ( [...] );

    [...]

    WORD(param) = SysCallID;
    SkCallNormalMode(&param);
    IumApi_NtGENERIC ( [...] );

    [...]
}
```

Figure 15: Executing normal-mode services by functions with prefix Nk (NkTerminateProcess)

by the Secure System process (*[1]* in Figure 16) and the *BioIso.exe* IUM application (*[2]* in Figure 16). Next, the operation of the worker thread owned by *BioIso.exe* is discussed.

[1]

```
Breakpoint 0 hit
nt!VslpDispatchIumSyscall:
fffff802`fdf616e0 56              push    rsi
0: kd> !thread
THREAD ffffbc829a699340  Cid 0004.0008  Teb: 0000000000000000 Win32Thread: 0000000000000000 RUNNING on processor 0
Not impersonating
Owning Process            ffffbc829a620040       Image:         System
Attached Process          ffffbc829cfe1040       Image:         Secure System

[...]

Priority 31 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child                                                           : Call Site
ffff8f80`eb3bb9a8 fffff802`fdfb546f : fffff802`fe13aea0 00000000`00000001 ffffbc82`9a699340 fffff802`00000000 : nt!VslpDispatchIumSyscall
[...]
ffff8f80`eb3bbb90 fffff802`fdf6a456 : fffff802`fe15d180 ffffbc82`9a699340 fffff802`fdeb747c ffffffff`00057fc8 : nt!PspSystemThreadStartup+0x41
ffff8f80`eb3bbbe0 00000000`00000000 : ffff8f80`eb3bc000 ffff8f80`eb3b5000 00000000`00000000 00000000`00000000 : nt!KiStartSystemThread+0x16
```

[2]

```
Breakpoint 0 hit
nt!VslpDispatchIumSyscall
fffff802`fdf616e0 56              push    rsi
3: kd> !thread
THREAD ffffbc82a0853080  Cid 09bc.09c4  Teb: 00000007eeccb000 Win32Thread: 0000000000000000 RUNNING on processor 3
Not impersonating
Owning Process            ffffbc82a0864800       Image:         BioIso.exe

[...]

Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child                                                           : Call Site
ffff8f80`ef2217e8 fffff802`fdfb546f : 00000000`00000001 00000000`00000001 ffffbc82`a0853080 ffffbc82`a0864800 : nt!VslpDispatchIumSyscall
[...]
ffff8f80`ef221a80 00007ffb`53a55e70 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : nt!KiStartUserThreadReturn
00000007`eeacfa78 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!RtlUserThreadStart
```

Figure 16: Invocation of VslpDispatchIumSyscall

The thread enters the *VslpEnterIumSecureMode* function. This function issues VTL calls in a loop, by executing the hypercall with call code *0x11*. These VTL calls are issued by invoking *HvlSwitchToVsmVtl1*, such that the SSCN is set to *0* and the operation code is set to *0x0* (see Section 4). At a given point in time, the data returned from the VTL call contains either a system service index or a normal-mode service code. Normal-mode service codes are used for uniquely identifying normal-mode services that are not implemented as system calls in the normal kernel.

If the returned data contains a system service index, the *VslpDispatchIumSyscall* function invokes the corresponding system service routine. If the returned data contains a normal-mode service code, the *PsDispatchI-*

umService function invokes the corresponding normal service. *PsDispatchIumService* implements multiple condition blocks for invoking specific functions for a given normal service code.

Figure 17 depicts the presence of a system service index in the data returned from a VTL call issued in *VslpEnterIumSecureMode*. The system service index *0x48*, which specifies the *NtCreateEvent* system call, results in *VslpDispatchIumSyscall* invoking the *NtCreateEvent* system service routine. This routine is implemented in the normal kernel.

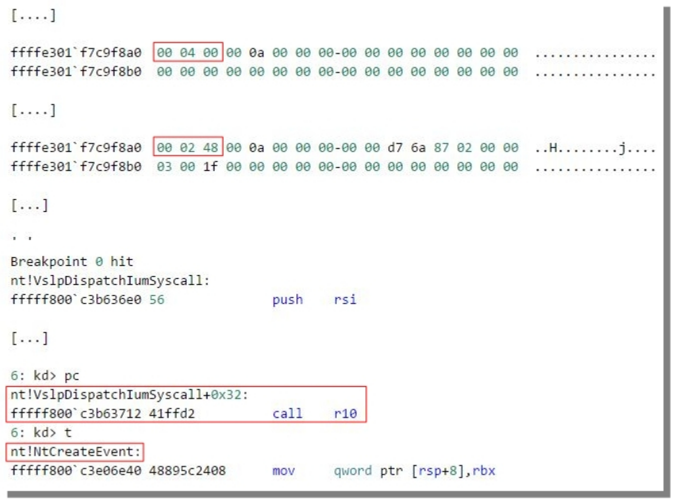

Figure 17: VslpDispatchIumSyscall invoking the NtCreateEvent system call

Once a normal-mode service is handled in *VslpDispatchIumSyscall* or *PsDispatchIumService*, the loop issuing VTL calls with operation code *0x0* is continued. At some point, the worker threads owned by *BioIso.exe* is put to sleep and terminated.

Appendix

Secure Services

Function	SSCN
DbgkCopyProcessDebugPort	*0xB*
HvlCollectLivedump	*0xE9*
HvlInitializeProcessor	*0x2*
HvlNotifyDebugDeviceAvailable	*0xF0*
HvlpGetSecurePageList	*0x802*
HvlPrepareForRootCrashdump	*0xEC*
HvlPrepareForSecureHibernate	*0xEB*
HvlpStartSecurePageListIteration	*0x800*
KeBalanceSetManager	*0xD1*
KeCopyPrivilegedPage	*0xE4*
KeRequestTerminationThread	*0x8*
KeReservePrivilegedPages	*0xD2*
KeSecureProcess	*0x6*
KeSetPagePrivilege	*0xE6/0xE3/0xE5*
KeUnsecureProcess	*0x1B*
MiApplyDynamicRelocations	*0xD3*
MiFlushEntireTbDueToAttributeChange	*0x0*
NtDebugActiveProcess	*0xB*
NtRemoveProcessDebug	*0xB*
PopAllocateHiberContext	*0x1F*
PspInitPhase3	*0x3*
PspUserThreadStartup	*0x0*
VslAbortLiveDump	*0x28*
VslCloseSecureHandle	*0x1B*
VslConfigureDynamicMemory	*0x21*
VslConnectSwInterrupt	*0x22*
VslCreateSecureAllocation	*0x13*
VslCreateSecureImageSection	*0x16*
VslCreateSecureProcess	*0x5*
VslCreateSecureThread	*0x7*
VslEnableOnDemandDebugWithResponse	*0x10*
VslEndSecurePageIteration	*0x801*
VslExchangeEntropy	*0x1E*
VslFastFlushSecureRangeList	*0xE1*
VslFillSecureAllocation	*0x14*
VslFinalizeLiveDumpInSk	*0x27/0x28*
VslFinalizeSecureImageHash	*0x17*
VslFinishSecureImageValidation	*0x18*
VslFlushSecureAddressSpace	*0xE0*
VslFreeSecureHibernateResources	*0x20*
VslGetNestedPageProtectionFlags	*0xE7*
VslGetOnDemandDebugChallenge	*0xF*
VslGetSecurePebAddress	*0xC0*
VslGetSecureTebAddress	*0xC*
VslGetSetSecureContext	*0xE*
VslIsTrustletRunning	*0x12*
VslIumEfiRuntimeService	*0xE8*
VslIumEtwEnableCallback	*0xD4*
VslLiveDumpQuerySecondaryDataSize	*0x23*
VslMakeCodeCatalog	*0x15*
VslNotifyShutdown	*0xEE*
VslpAddLiveDumpBufferChunk	*0x25*
VslpConnectedStandbyPoCallback	*0x29*
VslpConnectedStandbyWnfCallback	*0x29*
VslpIumPhase0Initialize	*0xD0*
VslpIumPhase4Initialize	*0x1*
VslpKsrEnterIumSecureMode	*0xF1*
VslPrepareSecureImageRelocations	*0x19*
VslpSetupLiveDumpBuffer	*0x26*
VslQuerySecureKernelProfileInformation	*0x2A*
VslRegisterLogPages	*0xEA*
VslRegisterSecureSystemProcess	*0x4*
VslRelocateImage	*0x1A*
VslReportBugCheckProgress	*0xED*
VslRetrieveMailbox	*0x11*
VslRundownSecureProcess	*0xA*
VslSetupLiveDumpBufferInSk	*0x24/0x28*
VslSlowFlushSecureRangeList	*0xE2*
VslTerminateSecureThread	*0x9*
VslTransferSecureImageVersionResource	*0x1D*
VslValidateDynamicCodePages	*0x1C*
VslValidateSecureImagePages	*0xC1*

References

[Mic17] Microsoft. Hypervisor Top Level Functional Specification. 2017. Version 5.0b; https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs.