ERNW
providing security.

Virtual Secure Mode: Protections of 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

This work discusses implemented mechanisms for securing the VSM communication interfaces.

2 Restrictions on Issuing VTL Calls

Hyper-V restricts the issuing of VTL calls. For a VTL call to be issued, among other things, it has to be initiated from the most privileged CPU mode. For example, an entity invoking a VTL call have to execute with a Current Privilege Level (CPL) of *0*, which is assigned by the processor executing the entity. Further, the input values have to be valid. ([Mic17], Section 15.6.1.1) describes the restrictions on issuing VTL calls in greater detail.

3 Marshalling and Sanitization

The secure kernel marshalls and sanitizes the input and output data of VTL returns. An example is the invocation of functions referenced in the *IumSyscallArgFcnTable* array. These functions are invoked in the *IumApi_NtGENERIC* function. This function, in turn, is invoked in *SkSyscall* before and after a VTL return is issued by *SkCallNormalMode*. *SkSyscall* is where the secure kernel issues VTL returns to request normal-mode services. The functions referenced in the *IumSyscallArgFcnTable* array marshall and sanitize data passed to, and returned from, the normal kernel. This is a security measure for checking, controlling, and managing the data coming in, and going out of, the secure kernel in a centralized way. This significantly reduces the risk of exploiting implementation or design errors involving the malicious manipulation of this data. It also indicates that data originating from the normal kernel is not explicitly trusted by the secure kernel.

Figure 1 depicts the contents of the *IumSyscallArgFcnTable* array. The array references functions that represent data marshallers and sanitizers for data of simple types and on a per-type basis for data of complex types. The latter involve marshallers and sanitizers of data of specific data structures. The *IumArg_GENERIC* function (see Figure 1) performs generic marshalling and sanitization. *IumArg_PALPC_MESSAGE_ATTRIBUTES* and *IumArg_PPORT_MESSAGE* are examples of per-type marshallers and sanitizers of data of type *ALPC_MESSAGE_ATTRIBUTES* and *PORT_MESSAGE*.

```
.rdata:000000014006D1D0 IumSyscallArgFcnTable dq offset IumArg_GENERIC
.rdata:000000014006D1D0
.rdata:000000014006D1D8                 dq offset IumArg_PALPC_MESSAGE_ATTRIBUTES
.rdata:000000014006D1E0                 dq offset IumArg_PHANDLE
.rdata:000000014006D1E8                 dq offset IumArg_POBJECT_ATTRIBUTES
.rdata:000000014006D1F0                 dq offset IumArg_PPORT_MESSAGE
.rdata:000000014006D1F8                 dq offset IumArg_PSID
.rdata:000000014006D200                 dq offset IumArg_PWSTR
.rdata:000000014006D208                 dq offset IumArg_PUNICODE_STRING
.rdata:000000014006D210                 dq offset IumArg_PWORKER_FACTORY_DEFERRED_WORK
.rdata:000000014006D218                 dq offset IumArg_PWSTR
```

Figure 1: The IumSyscallArgFcnTable array

4 Access Control: Hypercalls

Hyper-V enforces access control over hypercall execution. For a partition to execute a hypercall protected by access control, it has to possess the required privileges. These privileges are assigned by the hypervisor to each partition in the form of flags declared as part of a bitmask. The bitmask is stored in the *HvPartitionPropertyPrivilegeFlags* Hyper-V variable ([Mic17], Section 4.2.2).

Figure 2 depicts the value of *HvPartitionPropertyPrivilegeFlags* assigned to the partition hosting the normal and the secure kernel. The *HvPartitionPropertyPrivilegeFlags* queries the value of *HvPartitionPropertyPrivilegeFlags*. This function is implemented as part of the *winhvr* driver. When the second parameter of *WinHvGetPartitionProperty* is *0x10000*, the function queries the value of *HvPartitionPropertyPrivilegeFlags* from the hypervisor (*003b800000002e7f* in Figure 2).

([Mic17], Section 4.2.2) provides information on the layout of the privilege flags that are part of *HvPartitionPropertyPrivilegeFlags* and how the flags can be interpreted. For example, the *CreatePartitions* privilege flag implements access control over the execution of the *HvCreatePartition* hypercall. The *PostMessages* privilege flag implements access control over the execution of the *HvPostMessage* hypercall.

5 Access Control: IUM System Calls

The secure kernel implements access control over the execution of IUM system calls. The IUM system calls *IumSecureStorageGet*, *IumSecureStoragePut*, *IumCreateSecureSection*, *IumGetDmaEnabler*, *IumOpenSecureSection*, and *IumProtectSecureIo* are protected by access control. These functions evaluate flag values and return

```
[...]

winhvr!WinHvGetPartitionProperty:
fffff80a`482d1e04 488bc4          mov     rax,rsp
0: kd> ?? @rdx
unsigned int64 0x10000

[...]

0: kd> dp 0xffffab81`6f5ee420
ffffab81`6f5ee420  003b8000`00002e7f 00000000`00000001
ffffab81`6f5ee430  ffffab81`6f5ee4d9 fffff80a`47a870c8
ffffab81`6f5ee440  fffff80a`47a6a5f0 ffff988a`16c90000

[...]
```

Figure 2: A value of HvPartitionPropertyPrivilegeFlags

the error code *0xC0000022* (*STATUS_ACCESS_DENIED*) if the flags are not set.[1] The evaluated flags are stored at offsets of an address stored in the *gs:8* register. The flags are declared as part of the policy options of the trustlet invoking the IUM system calls. These options are used by the secure kernel to enable or disable secure kernel functionalities for trustlet, such as execution of IUM system calls. The policy options are described in ([YIRS17], Section "Trustlet Policy Metadata"). Policy options of trustlets are signed data. A modification of this data results in invalidation of its signature and prevents the execution of the trustlets associated with the modified options.

6 Secure Data Sharing: Mailboxes

The concept of mailboxes enables trustlets to share data with entities running in the normal environment in a secure manner. Figure 3 depicts the workflow of mailbox-based data sharing.

When a trustlet has data that it needs to share with an entity running in the normal environment, it populates a mailbox with the data. A mailbox is a memory region designated for storing shared data. Each mailbox can be uniquely identified by a mailbox ID. A trustlet populates a mailbox by issuing the *IumPostMailbox* IUM system call. The first parameter of this function is the ID of the mailbox to be populated (*ID* in Figure 3), the second stores the address of the data buffer where the shared data is stored (*buf* in Figure 3), and the third is the buffer size (*size* in Figure 3).

Each trustlet may have up to eight mailboxes. Possible mailbox IDs are between *0* and *7*. The maximum size of each mailbox is *4092* bytes. *IumPostMailbox* evaluates the trustlet-provided mailbox ID and mailbox size against the previously mentioned upper values. *IumPostMailbox* then allocates heap memory. It also copies the data stored at the address that is the second parameter of *IumPostMailbox* at offset *0x4* of the allocated memory. The beginning of the allocated memory stores the size of the copied data. Finally, *IumPostMailbox* loads the address of the heap memory storing the shared data, and the size of the data, to an address referencing the mailbox indexed by the trustlet-provided mailbox ID. This is done by executing an atomic compare-and-exchange operation implemented with the *cmpxchg* instruction.[2]

The implementation of *cmpxchg* in *IumPostMailbox* indicates that the trustlet-provided data to be shared is stored in a mailbox only if the mailbox is empty; that is, if the address referencing the mailbox is zeroed out. The address referencing the mailbox with ID is at offset *0x8*ID+0x100* of the address stored at *gs:8+0x30* (offset: *0x8*ID+0x100* ← *buf* in Figure 3). *gs:8* is the address stored in the *gs:8* register at the time of issuing *IumPostMailbox*.

[1] https://msdn.microsoft.com/en-us/library/cc704588.aspx [Retrieved: 19/4/2018]

[2] https://msdn.microsoft.com/en-us/library/windows/desktop/ms683560(v=vs.85).aspx [Retrieved: 20/4/2018]

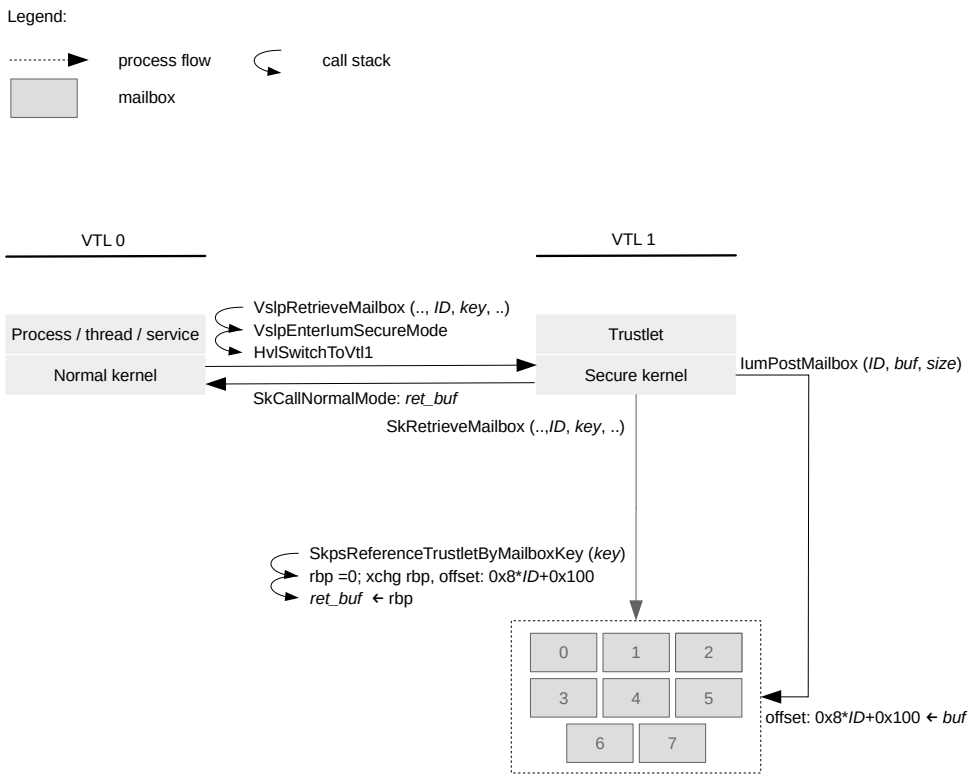

Figure 3: The workflow of mailbox-based data sharing

Once data is stored in a mailbox, entities running in the normal environment can retrieve it. This is done by invoking the *VslRetrieveMailbox* function implemented in the normal kernel. Parameters of *VslRetrieveMailbox* include the ID of the mailbox from which data is retrieved (*ID* in Figure 3) and a mailbox key (*key* in Figure 3). A mailbox key serves as a password for retrieving data stored in the mailboxes of a given trustlet. *VslRetrieveMailbox* stores the mailbox ID and the mailbox key in a buffer and issues a VTL call passing the buffer as input data.

VslRetrieveMailbox issues a VTL call by invoking the *VslpEnterIumSecureMode* and *HvlSwitchToVtl1* functions. *VslRetrieveMailbox* requests a secure service with SSCN *0x13*. This results in invoking the *IumInvokeSecureService* and *SkRetrieveMailbox* functions in the context of the secure kernel.

SkRetrieveMailbox first invokes the *SkpsReferenceTrustletByMailboxKey* function. This function identifies the trustlet whose mailboxes are protected by the mailbox key transferred from the normal environment. *SkpsReferenceTrustletByMailboxKey* searches through the attributes of running trustlets for this mailbox key. In order to search the attributes of a given trustlet, *SkpsReferenceTrustletByMailboxKey* invokes the *SkFindTrustletAttribute* function. This indicates that mailboxes keys are stored as trustlets' attributes. Attributes of a given trustlet store information associated with the trustlet and they are embedded in the executable implementing the trustlet ([Mic17], section "Trustlet Attributes", Table 3-5).

If *SkpsReferenceTrustletByMailboxKey* cannot identify a trustlet, *SkRetrieveMailbox* returns the error code *0xC000-0034* (*OBJECT_NAME_NOT_FOUND*).[3] If *SkpsReferenceTrustletByMailboxKey* identifies a trustlet, *SkRetrieveMailbox* allocates a buffer (*ret_buf* in Figure 3) and sets the value of the *rbp* register to *0* (*rbp=0* in Figure 3). It then loads into *rbp* the address stored at offset *0x8*ID+0x100* of the address stored in the *rsi* register. As mentioned earlier, this address is presumably the address referencing the mailbox indexed by ID. The loading of the address is implemented using the *xchg* instruction (*xchg rbp, offset: 0x8*ID+0x100* in Figure 3). This instruction

[3] https://msdn.microsoft.com/en-us/library/cc704588.aspx [Retrieved: 19/4/2018]

exchanges the address stored at the offset *0x8*ID+0x100* of *rsi* with the value stored in *rbp* (i.e.,*0*). This effectively zeroes out the value stored at the offset *0x8*ID+0x100* of *rsi* and populates *rbp*. As discussed earlier, this indicates that the mailbox indexed by ID has been retrieved and may be populated again.

SkRetrieveMailbox then populates the newly allocated buffer with the data stored at offset *0x4* of *rbp* (*ret_buf* ← *rbp* in Figure 3). This is the data shared by the trustlet. *SkRetrieveMailbox* then issues a VTL return by invoking *SkCallNormalMode* in order to switch to VTL 0. *SkRetrieveMailbox* passes the buffer populated with shared trustlet data to VTL 0 (*SkCallNormalMode: ret_buf* in Figure 3). This provides the data shared by a trustlet to the requesting entity that runs in the normal environment.

7 Secure Data Sharing: Secure Storage Blobs

The concept of secure storage blobs enables trustlets to share data between each other in a secure manner. A secure storage blob is a memory region designated for storing shared data. The functionalities of the secure storage blob mechanism are implemented in the IUM system calls *IumSecureStoragePut* and *IumSecureStorageGet*. *IumSecureStoragePut* is used by trustlets for storing data in secure storage blobs. *IumSecureStorageGet* is used by trustlets for retrieving data stored in secure storage blobs.

The storing data into, and retrieving data from, secure storage blobs is subject to access control. This is based on an authentication value that trustlets accessing a secure storage blob have to provide. This authentication value may be either a collaboration ID or a trustlet instance.

A trustlet sharing data via a secure storage blob may associate a collaboration ID with the blob. This allows multiple trustlets that are in possession of this ID to access the blob. Same as the mailbox key, the collaboration ID of a given trustlet is stored as part of the trustlet's attributes ([Mic17], section "Trustlet Attributes"). Alternatively to collaboration ID, a trustlet sharing data via a storage blob may associate a trustlet instance with the blob. This allows only the specific trustlet that is in possession of this instance to access the blob. A trustlet instance is a form of trustlet identity. It is a 16-byte number generated by the secure kernel and is unique to each instantiated trustlet ([Mic17], Section "Trustlet Identity"). When a trustlet is accessing a secure storage blob, if no collaboration ID is provided, the trustlet instance is used for authentication.

Analysing the *IumSecureStorageGet* function allows for better understanding the way in which access to secure storage blobs is secured. Figure 4 depicts a pseudo-code of the implementation of *IumSecureStorageGet*. The second parameter of *IumSecureStorageGet* is the address of a buffer where the data retrieved from a secure storage blob is to be stored (*buf* in Figure 4). The third parameter of *IumSecureStorageGet* is where the size of shared data is to be stored (*size* in Figure 4).

IumSecureStorageGet first invokes the *SkGetCollaborationId* function. This function attempts to extract the collaboration ID from the attributes of the trustlet retrieving shared data (*collabID* in Figure 4) by invoking *SkFindTrustletAttribute*. If a collaboration ID is not present (*if(res)* and *else* in Figure 4), then the trustlet instance is extracted. At the time of invoking *SkGetCollaborationID*, the trustlet instance is stored at offset *0x1D0* of the address stored in the *gs:8* register. *SkGetCollaborationID* then stores the extracted collaboration ID, or the trustlet instance, into a data buffer. This buffer is referred to as the authentication data buffer in this work (*auth* in Figure 4).

Once the authentication data buffer is populated, *IumSecureStorageGet* invokes *SkGetBlob*. This function first iterates through an array that stores pointers to secure storage blobs (*blob_array* in Figure 4). The address of this array is stored in a global variable of the secure kernel. Each element of the array stores a pointer to the authentication value associated with each storage blob, that is, a collaboration ID or a trustlet instance (*el->auth* in Figure 4). It also stores a pointer to the data stored in the blob.

During the iteration of the array of storage blobs, *SkGetBlob* compares the content of the previously populated authentication buffer with the authentication value associated with each storage blob. If a match is found (*authenticated = 1* in Figure 4), *SkGetBlob* copies the data stored in the blob in the buffer for storing shared data (*memmove* and *buf* in Figure 4). It also stores the size of the data in the variable for storing this size (*size* in Figure 4). The shared data is stored at offset *0x18* of the storage blob that *SkGetBlob* has granted access to (*el+0x18*

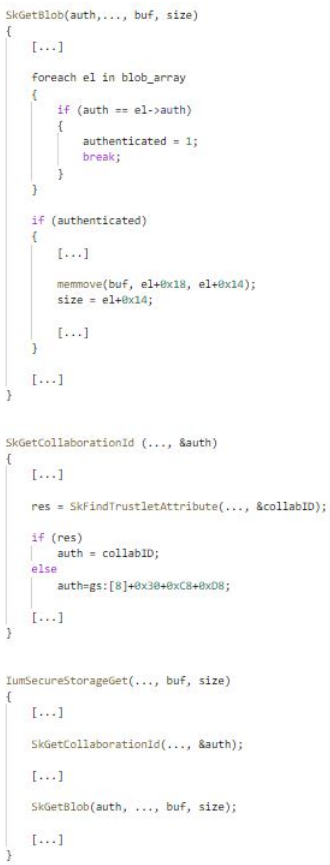

Figure 4: Pseudo-code of the implementation of IumSecureStorageGet

in Figure 4). The size of the shared data is stored at offset *0x14* of this blob (*el+0x14* in Figure 4). The data buffer storing the shared data, and the data size, are then returned to the trustlet invoking the *IumSecureStorageGet* function.

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.

[YIRS17] Pavel Yosifovic, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals, Part 1 and Part 2*. 2017. Microsoft Press.