ERNW
providing security.

Virtual Secure Mode: Initialization

Dominik Phillips
dphillips@ernw.de✉

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 section described the process for VSM initialization activities performed by the Windows loader when Windows 10 is booted. The Windows loader is the Windows boot entity that initializes VSM with respect to values of configuration parameters. Figure 1 depicts the booting process of Windows 10 with VSM enabled. In this process, each entity verifies the integrity of, and loads, the next entity in the booting chain. The Unified Extensible Firmware Interface (UEFI) firmware with Secure Boot enabled extends the trust chain securing the booting process of Windows. It serves as the first root of trust in this chain. The UEFI firmware verifies the integrity of and loads the boot manager. The following activities of the booting process can be structured into four phases:

- Phase 1: The boot manager loads the Windows loader. The Windows loader then loads the hypervisor loader ([1] in Figure 1);
- Phase 2: The hypervisor loader loads the Hyper-V hypervisor. Once Hyper-V is loaded, execution control is switched back to the Windows loader ([2] in Figure 1);
- Phase 3: The Windows loader loads the secure kernel ([3] in Figure 1);

- Phase 4: The Windows loader loads the normal kernel. The secure and the normal kernel load Windows 10 to its full extent, making it ready for use ([4] in Figure 1).

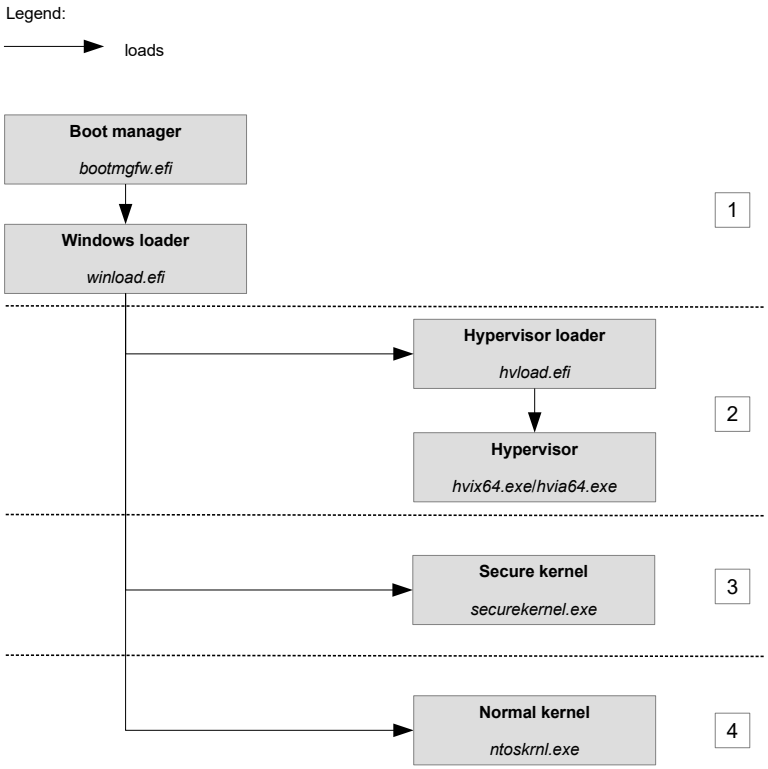

Figure 1: The booting process of Windows 10 with VSM enabled

The Windows loader starts the VSM initialization process. On an UEFI-enabled platform, the Windows loader is implemented in the *%SystemRoot%\System32\winload.efi* executable. The *OslPrepareTarget* function implemented as part of the Windows loader performs VSM initialization activities. TO DO Section 2.1.1-Section 2.1.5 discuss such activities performed in relevant functions invoked by *OslPrepareTarget*: *OslSetVsmPolicy*, *OslArchHypervisorSetup*, *OslArchHypercallSetup*, *OslFwProtectSecureBootVariables*, and *OslVsmSetup*. These functions are invoked by *OslPrepareTarget* in the order as presented in this section. When these functions are finished executing, the Windows loader loads the secure and the normal kernel, which then instantiate Isolated User Mode (IUM) applications. TO DO Section 2.1.6 discusses the requirements for an executable to be instantiated as an IUM application.

2 OslSetVsmPolicy

OslSetVsmPolicy processes configuration parameters for enabling or disabling the core VSM entities and configuring the VSM features Hypervisor Code Integrity (HVCI) and Credential Guard. The parameters for configuring HVCI and Credential Guard are stored in the system's registry. Figure 2 depicts pseudo-code of the implementation of OslSetVsmPolicy.

OslSetVsmPolicy first invokes *BlSecureBootGetBootPrivateVariable*. This function evaluates whether the UEFI variable *VbsPolicyDisabled* is defined. If defined, the core VSM entities and the VSM features will not be initialized.

If *VbsPolicyDisabled* is not set, *OslSetVsmPolicy* parses the registry key *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard* (see Figure 3). The *OslGetVsmEnabled* function evaluates the values stored in the keys *Scenarios\HypervisorEnforcedCodeIntegrity\Enabled*, *Scenarios\HypervisorEnforcedCodeIntegrity*

```
OslSetVsmPolicy( [...] )
{

    if ( BlSecureBootGetBootPrivateVariable("VbsPolicyDisabled", [...] ))
    {
        if ( OslGetVsmEnabled( [...] ))
        {
            [...]
            OslHiveReadWriteControlDword("DeviceGuard", "RequirePlatformSecurityFeatures", [...] );

            OslHiveReadWriteControlDword("DeviceGuard", "Mandatory", [...]);

            OslGetVbsHvciConfiguration( [...] );

            OslHiveReadWriteControlDword("DeviceGuard", "RequireMicrosoftSignedBootChain", [...] );
            [...]

        }
        BlVsmSetSystemPolicy( [...] );
        [...]

    }
    [...]
}
```

Figure 2: Pseudo-code of the implementation of OslSetVsmPolicy

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard
├─ Scenarios
│  └─ HypervisorEnforcedCodeIntegrity
│        ├─ Enabled
│        ├─ Level
│        └─ Locked
├─ EnableVirtualizationBasedSecurity
├─ Locked
├─ HyperVVirtualizationBasedSecurityOptOut
├─ RequirePlatformSecurityFeatures
├─ Mandatory
└─ RequireMicrosoftSignedBootChain
```

Figure 3: The layout of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard

Locked, *EnableVirtualizationBasedSecurity*, *Locked*, and *HyperVVirtualizationBasedSecurityOptOut*.

In addition to evaluating the above registry values, *OslGetVsmEnabled* evaluates the value of the *BcdOSLoaderInteger_HypervisorLaunchType* variable. This variable is stored in the system's Boot Configuration Database (BCD). It has to be set for the hypervisor to be loaded.[1]

When *OslGetVsmEnabled* is finished executing, *OslSetVsmPolicy* processes further configuration information. This information is stored in the registry values *RequirePlatformSecurityFeatures*, *Mandatory*, and *RequireMicrosoftSignedBootChain* (see Figure 3 and the invocations of *OslHiveReadWriteControlDword* in Figure 2). In addition, *OslSetVsmPolicy* invokes *OslGetVbsHvciConfiguration*. This function evaluates the *HypervisorEnforcedCodeIntegrity* registry value. This value is used for initializing the VSM feature HVCI.

Once *OslSetVsmPolicy* is finished processing the configuration parameters stored in the registry, it passes these parameters to the function *BlVsmSetSystemPolicy*. This function then evaluates the value stored in the *BcdOSLoaderInteger_SafeBoot* variable. This variable is stored in the system's BCD. It can have one of the following values: *SafemodeMinimal*, *SafemodeNetwork*, or *SafemodeDsRepair*.[2]. The VSM initialization procedure continues only if *BcdOSLoaderInteger_SafeBoot* has the value of *SafemodeMinimal*.

BlVsmSetSystemPolicy then initializes the variable *BlVsmpSystemPolicy*. This variable stores the configuration parameters previously read from the registry in the form of flags that are part of a bitmask value. Before storing

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/aa362670(v=vs.85).aspx [Retrieved: 25/4/2018]

[2] https://msdn.microsoft.com/en-us/library/windows/desktop/aa362656(v=vs.85).aspx [Retrieved: 25/4/2018]

them in *BlVsmpSystemPolicy*, *BlVsmSetSystemPolicy* compares the configuration parameters with configuration parameters stored in the UEFI variable *VbsPolicy*. In case of a mismatch, the values stored in the UEFI variable are used for initializing the core VSM entities and VSM features. This indicates that UEFI serves as the root of trust for evaluating at system start-up the integrity of configuration parameters stored in the registry. The configuration parameters stored in *VbsPolicy* are defined at the previous system shutdown and reflect the VSM configuration at that time.

VbsPolicy is not defined when Secure Boot is disabled. In the scenario where the VSM features HVCI and Credential Guard are not configured to operate, a default set of configuration parameters are stored in *BlVsmpSystemPolicy*. These parameters are for initializing only the core VSM entities. This indicates that Secure Boot is not a requirement for initializing these entities. Secure Boot is a requirement for initializing the VSM features Credential Guard and HVCI.

In the scenario where Credential Guard, and/or HVCI are configured to be enabled, and Secure Boot is not enabled, the core VSM entities, HVCI, and/or Credential Guard are not initialized.

Once populated with configuration parameters, *BlVsmpSystemPolicy* is stored in the *LOADER_PARAMETER_BLOCK* structure ([RSI12], Chapter 13). The Windows loader passes this structure to the normal and secure kernel when loaded.

3 OslArchHypervisorSetup

OslArchHypervisorSetup loads and executes the hypervisor loader. This section discusses the integrity verification process conducted as part of this activity. The integrity of the hypervisor loader executable is verified using the Authenticode digital signing technology. The hypervisor loader has to be signed by Microsoft. This section focuses on the cryptographic requirements that the digital Authenticode signature of the hypervisor loader must fulfill such that the loader is considered authentic.

On an UEFI-enabled platform, the hypervisor loader is implemented in the *%SystemRoot%\System32\hvloader.efi* executable (i.e., image). The Windows loader loads this executable in the *ImgpLoadPEImage* function. Figure 4 (label 1) depicts the functions preceding *ImgpLoadPEImage*. Once the hypervisor loader is loaded, it is executed in the *Archpx64TransferTo64BitApplicationAsm* function. Figure 4 (label 2) depicts the functions preceding *Archpx64TransferTo64BitApplicationAsm*.

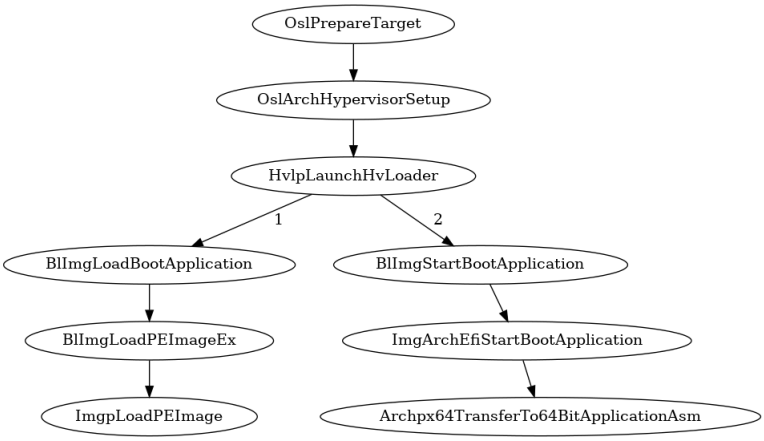

Figure 4: Function stack: Windows loader loads the hypervisor loader

ImgpLoadPEImage invokes three functions that are relevant to the image verification process: *ImgpGetScenarioFromImageFlags*, *ImgpGetHashAlgorithmForScenario*, and *ImgpValidateImageHash*.

Before the function *ImgpLoadPEImage* initiates the image verification process it must identify and store the signing requirements in configuration variables. This involves identifying and storing the required extended key usage (EKUs) that have to be present in the certificate of the signer of the image. It also involves identifying and storing the required hash algorithm used for signing certificates stored as part of the Authenticode signature.

ImgpGetScenarioFromImageFlags (see Figure 5, [1]) extracts a value indicating the required hash algorithm, such as Secure Hash Algorithm (SHA)-1, and stores it as an integer value. This work refers to this integer value as signing scenario. The signing scenario is stored as a flag that is part of a bitmask value. This work refers to this bitmask value as image loading parameters.

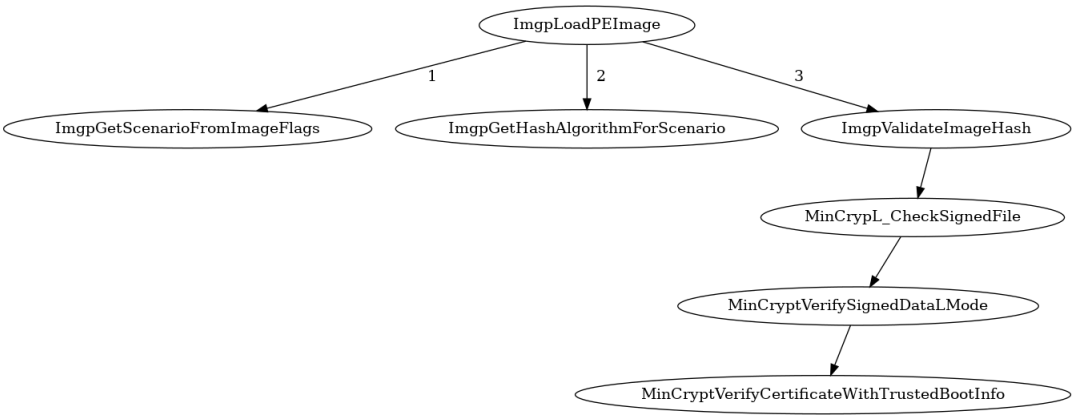

Figure 5: Function stack: Windows loader loads the hypervisor loader

In addition to *ImgpGetScenarioFromImageFlags*, *ImgpLoadPEImage* extracts from the image loading parameters a value indicating the EKUs that have to be present in the certificate of the signer of the image. This work refers to this value as EKU flag and to the associated EKUs as required signer's EKUs. A certificate's EKUs express the purposes for which the public key that is stored as part of certificate may be used. Code signing certificates issued by Microsoft contain specific EKUs that describe the use of the certificates' public keys for verifying signatures of specific executables. For example, the *1.3.6.1.4.1.311.10.3.37* (Isolated User Mode) EKU is stored in part of certificates used for verifying the signatures of IUM applications.

The function *ImgpGetHashAlgorithmForScenario* (see Figure 5, label [2]) maps the signing scenario extracted by *ImgpGetScenarioFromImageFlags* to a code identifying the required hash algorithm. This work refers to this code as hash code. For example, *ImgpGetScenarioFromImageFlags* extracts a signing scenario of *0* from the image loading parameters of the hypervisor loader. *ImgpGetHashAlgorithmForScenario* maps the signing scenario of *0* to the hash code *0x800c* if Secure Boot is enabled. This function maps the signing scenario of *0* to the hash code *0x8004* if Secure Boot is disabled. The hash codes *0x8004* and *0x800c* identify the SHA-1 and SHA-256 hash algorithms, respectively.

The *ImgpLoadPEImage* function initiates the integrity verification process by invoking the *ImgpValidateImageHash* function (see Figure 5, [3]). Based on the EKU flag, *ImgpValidateImageHash* stores the required signer's EKUs, extracted by *ImgpLoadPEImage*, in a variable. When the hypervisor loader is loaded, the required signer's EKU is *1.3.6.1.4.1.311.10.3.6*. In addition, it stores the EKU *1.3.6.1.5.5.7.3.3* in this variable. This is an EKU that has to be present in all code signing certificates issued by Microsoft. The hypervisor loader is signed by Microsoft.

The *MinCryptVerifyCertificateWithTrustedBootInfo* function (see Figure 5, [3]), invoked by *ImgpValidateImageHash*, verifies whether the required signer's EKUs are present in the certificate of the signer of the image being loaded. If a required EKU is missing, *MinCryptVerifyCertificateWithTrustedBootInfo* returns the error code *0xc0000428* (*INVALID_IMAGE_HASH*).[3] The certificate of the signer of the image being loaded is also verified against its root certificate. The root certificate is hardcoded in the Windows loader executable. The fact that hardcoded contents

[3]https://msdn.microsoft.com/en-us/library/cc704588.aspx [Retrieved: 25/4/2018]

of Windows loader executable are used for verification of the certificates of the signer of the hypervisor loader, shows that the root of trust for verifying the integrity of the loader is the Windows loader itself.

We analyzed the possibility for mitigating the previously described integrity verification process by modifying the system's boot configuration. This can be done by issuing the command *bcdedit /set nointegritychecks on*. The command sets the variable *BcdLibraryBoolean_DisableIntegrityChecks* to *1*. This variable is stored in the system's BCD. We observed that when the hypervisor loader is loaded (see [1] in Figure 4), the value of *BcdLibraryBoolean_DisableIntegrityChecks* is evaluated, however, ignored. The integrity of the hypervisor loader is always verified.

Once the hypervisor loader is loaded, execution control is transferred to it by executing *Archpx64TransferTo64BitApplicationAsm* (see [2] in Figure 4). The hypervisor loader then loads the Hyper-V executable (*hvix64.exe* or *hvax64.exe*). The image integrity verification process implemented in the hypervisor loader is conceptually identical to the one implemented in the Windows loader. The hypervisor has to be signed by Microsoft. The integrity of the Hyper-V executable is verified using the Authenticode digital signing technology.

4 OslArchHypercallSetup

OslArchHypercallSetup maps a page aligned at a guest physical memory address to a guest virtual address. This page is allocated to the partition hosting the normal and secure kernel and is populated by the hypervisor with code. This code is for the kernels to invoke hypervisor services, referred to as hypercalls. The page storing the code is referred to as the hypercall page.

OslArchHypercallSetup invokes *BlMmMapPhysicalAddressEx*. This function performs the mapping of the guest physical address, at which the hypercall page is aligned, to a guest virtual address. Figure 6 depicts the guest virtual address at which a hypercall page is aligned (*fffff802'206ea000* in Figure 6).

```
rcx=00000000`001c3d10
[...]
winload!BlMmMapPhysicalAddressEx:

kd> dps 00000000`001c3d10 L1
00000000`001c3d10  fffff802`206ea000
```

Figure 6: Guest virtual address of a hypercall page

When *OslArchHypercallSetup* is finished executing, the guest virtual address of the hypercall page is stored in the *HvlpHypercallCodePageVa* variable. Once the hypercall page is populated by Hyper-V, *HvlpHypercallCodePageVa* is stored in the *LOADER_PARAMETER_BLOCK* structure ([RSI12], Chapter 13). This structure is passed to the normal and secure kernel when they are loaded and executed. Figure 7 depicts the content of a populated hypercall page, aligned at the virtual address *fffff802'206ea000*. It is extracted from the context of the normal kernel, once it has been loaded by the Windows loader.

```
0: kd> u poi(nt!HvlpHypercallCodeVa)
fffff802`206ea000 0f01c1          vmcall
fffff802`206ea003 c3              ret
fffff802`206ea004 8bc8            mov     ecx,eax
fffff802`206ea006 b811000000      mov     eax,11h
fffff802`206ea00b 0f01c1          vmcall
fffff802`206ea00e c3              ret
fffff802`206ea00f 488bc1          mov     rax,rcx
fffff802`206ea012 48c7c111000000  mov     rcx,11h
[...]
```

Figure 7: The content of a hypercall page

5 OslFwProtectSecureBootVariables

Among other parameters, *OslFwProtectSecureBootVariables* verifies configuration parameters of the VSM feature Credential Guard. The verified parameters are stored in the system's registry, and their values are verified

against their counterparts stored as UEFI variables. In case of a mismatch, the values stored as UEFI variables are used for initialization of Credential Guard. The configuration parameters stored as UEFI variables are written into the UEFI context at the first system shutdown after enabling VSM and reflect the Credential Guard configuration at that time. UEFI serves as the root of trust for evaluating at system start-up the integrity of configuration parameters of Credential Guard stored in the registry. Secure Boot is a requirement for Credential Guard. If Secure Boot is not enabled, Credential Guard is not initialized (see Section 2)

OslFwProtectSecureBootVariables invokes the *OslFwProtectSecConfigVars* function. Figure 12 depicts pseudo-code of the implementation of *OslFwProtectSecConfigVars*. *OslFwProtectSecConfigVars* evaluates the values of the registry keys *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA\RunasPPL* and *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA\LsaCfgFlags* against their counterparts stored as UEFI variables.

OslFwProtectSecConfigVars first evaluates the value stored in the key *RunasPPL* (*OslHiveReadWriteControlDword* and *EfiGetVariable* in Figure 8). *RunasPPL* configures the Local Security Authority (LSA) process to be executed under Protected Process Light (PPL) protection.[4] PPL is a security mechanism protecting the memory space of a process from accesses by other untrusted processes. If the *RunasPPL* registry key is set and Secure Boot is enabled, the *OslFwProtectSecConfigVars* compares the value read from the registry with the UEFI variable *Kernel_Lsa_Ppl_Config*.

```
OslFwProtectSecConfigVars( [...] )
{
    [...]

    OslHiveReadWriteControlDword( "Lsa", "RunasPPL", [...] );
    [...]

    EfiGetVariable( "Kernel_Lsa_Ppl_Config", [...] );
    [...]

    if ( BlSecureBootGetBootPrivateVariable("Kernel_Lsa_Cfg_Flags_Cleared", [...]) )
    {
        [...]

        if ( OslHiveReadWriteControlDword( [...], "Lsa", "LsaCfgFlags", &getRegValue ) )
        {

            if ( EfiGetVariable( "Kernel_Lsa_Cfg_Flags", &getEfiValue, [...] ))
            {
                [...]
            }
            else
            {
                [...]
                if ( ( getRegValue | getEfiValue ) == getEfiValue )
                [...]
            }
            [...]
        }
    }
    [...]
}
```

Figure 8: Pseudo-code of the implementation of OslFwProtectSecConfigVars

OslFwProtectSecConfigVars then evaluates the value stored in the key *LsaCfgFlags*.[5] Among other things, the *LsaCfgFlags* registry key indicates if Credential Guard should be enabled. The value stored in the *LsaCfgFlags* registry key (*OslHiveReadWriteControlDword* and *getRegValue* in Figure 8) is compared with the value stored in the UEFI variable *Kernel_Lsa_Cfg_Flags* (*EfiGetVariable* and *getEfiValue* in Figure 8). The comparison is done with a logical OR operation (*getRegValue | getEfiValue* in Figure 8). In addition, *OslFwProtectSecConfigVars* evaluates the value of the UEFI variable *Kernel_Lsa_Cfg_Flags_Cleared*. If this variable is set, the configuration parameters of Credential Guard will be cleared and Credential Guard will not be initialized.

[4] https://docs.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection [Retrieved: 25/4/2018]

[5] https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-manage [Retrieved: 25/4/2018]

6 OslVsmSetup

Figure 9 depicts pseudo-code of the implementation of *OslPrepareTarget* invoking the *OslVsmSetup* function. The primary task of *OslVsmSetup* is to load and execute the secure kernel and its modules. After the secure kernel is loaded, the Windows loader loads the normal kernel. The secure and the normal kernel then load Windows 10 to its full extent, making it ready for use (see Figure 1).

```
[...]
if ( !_bittest(&BlVsmpSystemPolicy, 0xBu)
    && (HvlpResourceInitialized
    || (unsigned __int8)HviIsHypervisorMicrosoftCompatible( [...] )) )
{
    NTSTATUS = OslVsmSetup( [...] );
}
[...]
```

Figure 9: Pseudo-code of the implementation of OslPrepareTarget invoking OslVsmSetup

The secure kernel is implemented in the *%SystemRoot%\System32\securekernel.exe* executable. The secure kernel is loaded if the 11-th bit in the *BlVsmpSystemPolicy* variable is not set (*bittest* and *0xBu* in Figure 9). In addition, the *HvlpResourceInitialized* variable needs to be set. *HvlpResourceInitialized* is set only when the hypervisor loader has loaded the Hyper-V hypervisor. If this variable is not set, for the secure kernel to be loaded, the presence of Hyper-V has to be determined. Hyper-V sets the 31-st bit of the value stored in the *ecx* register ([Mic17], Section 2.2). The *HviIsHypervisorMicrosoftCompatible* function verifies that this bit is set.

OslVsmSetup invokes the *OslpVsmLoadModules* and *OslLoadImage* functions to load and verify the integrity of the *securekernel.exe* as well as its required modules. The integrity of *securekernel.exe* and its required modules is verified using the Authenticode digital signing technology. The secure kernel has to be signed by Microsoft. The verification of cryptographic requirements, such as EKUs that have to be present in the certificate of the signer of the image, is conceptually identical to the one described in Section 3.

Before *OslpVsmLoadModules* is invoked, *OslVsmSetup* evaluates the value stored in the *BCDE_OSLOADER_TYPE_VSM_LAUNCH_TYPE* variable. This variable is stored in the system's BCD. It can have the value *Off* or *Auto*. *OslpVsmLoadModules* is invoked only if *BCDE_OSLOADER_TYPE_VSM_LAUNCH_TYPE* has the value of *Auto*.

7 Instantiation of IUM Applications

Once the normal and the secure kernel are loaded and executed, they instantiate IUM applications (i.e., trustlets). For an IUM application to be instantiated, the normal kernel invokes the *NtCreateUserProcess* function. Among other things, this function initializes relevant kernel structures for process management, such as the structure of type *EPROCESS*. This structure contains relevant process information, such as process ID. *NtCreateUserProcess* collaborates with the secure kernel to finish instantiating the IUM application.

Third parties cannot instantiate an application as a trustlet without fulfilling certain requirements. The use of functions implemented as part of the Windows API and exposed to third parties, such as *CreateUserProcess*, do not instantiate executables as IUM applications. This is because such functions do not set a concrete flag used by the normal kernel for instantiating an executable as an IUM application. This flag is referred to as the IUM application flag in this work. In order to instantiate an application as an IUM application, third parties need to implement custom executable loaders that directly invoke *NtCreateUserProcess* such that the IUM application flag is set.

In addition to the IUM application flag being set, for an application to be instantiated as a trustlet, it has to be properly signed. Each executable implementing an IUM application has to be signed by Microsoft using the Authenticode digital signing technology. The certificate issued by the signer of the IUM application has to possess the following EKUs (textual EKU descriptions provided in round brackets, see Section 3):

- *1.3.6.1.5.5.7.3.3* (*Code Signing*)

- *1.3.6.1.4.1.311.10.3.6* (*Windows System Component Verification*)
- *1.3.6.1.4.1.311.10.3.37* (*Isolated User Mode*)
- *1.3.6.1.4.1.311.10.3.24* (*Protected Process Verification*)

The EKU Isolated User Mode is used specifically for marking certificates that can be used only for verifying signatures of executables implementing IUM applications. As part of the Authenticode signature verification process of a given IUM application, the EKUs stored in the certificate used for signing the application are evaluated against EKUs hardcoded in the *ci.dll* library file. In case of a mismatch, the application is considered not authentic.

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.

[RSI12] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. 2012. Microsoft Press.