Windows Defender Application Control: Initialization

Dominik Phillips✉ Aleksandar Milenkoski
dphillips@ernw.de *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 Device Guard Image Integrity, 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 describes the process for initializing Windows Defender Application Control (WDAC) performed by the Windows loader and the kernel when Windows 10 is booted (see Figure 1).

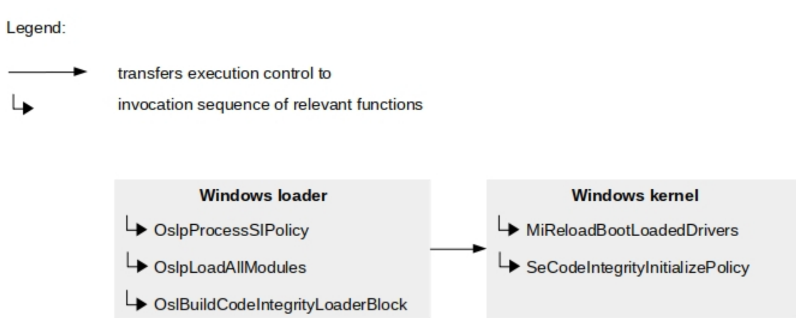

Figure 1: WDAC initialization

Windows loader The *OslPrepareTarget* function implemented as part of the Windows loader performs WDAC initialization activities. These activites are performed by the functions: *OslpProcessSIPolicy*, *OslpLoadAllModules*, and *OslBuildCodeIntegrityLoaderBlock*. These functions are invoked by *OslPrepareTarget*.

The *OslpProcessSIPolicy* function initializes and loads the WDAC policy in the context of the Windows loader. This involves verifying the integrity of the WDAC policy, if signed. Once *OslpProcessSIPolicy* is finished executing, the WDAC policy may be used for image verification by the Windows loader. Among other images, the Windows loader verifies the integrity of the *ci.dll* file.

The functions *OslBuildCodeIntegrityLoaderBlock* and *OslpLoadAllModules* populate with WDAC initialization parameters the *CodeIntegrityLoaderBlock* (see Figure 17) and *LoadOrderListHead* fields ultimately referenced by the *_LOADER_PARAMETER_BLOCK* structure ([RSI12], Chapter 13). The *_LOADER_PARAMETER_BLOCK* structure is ultimately passed to the Windows kernel at execution transfer between the Windows loader and the kernel. Once *_LOADER_PARAMETER_BLOCK* is populated with WDAC initialization parameters, the Windows loader transfers the execution control to the Windows kernel. To this end, it executes the *OslArchTransferToKernel* function.

Windows kernel Once the Windows loader has transferred the execution control to the kernel, it uses the populated *_LOADER_PARAMETER_BLOCK* structure to initialize WDAC in the context of the kernel. The kernel is initialized in two phases: Phase 0 and Phase 1 ([RSI12], Chapter 13). The kernel invokes in Phase 0 the *MiReloadBootLoadedDrivers* function. This function allocates a memory region in the virtual address space assigned to the kernel for the *ci.dll* file. The starting address of this space is referred to as the image base address of *ci.dll*.

Once Phase 0 is finished, the kernel starts Phase 1. In this phase, the kernel continues initializing WDAC. This involves for example, invoking the *SeCodeIntegrityInitializePolicy* function, which initializes the WDAC policy. Once *SeCodeIntegrityInitializePolicy* is finished executing, the WDAC policy may be used for image verification by the Windows kernel.

2 Windows Loader: OslpProcessSIPolicy

OslpProcessSIPolicy loads and processes the *SIPolicy.p7b* file, that is, the WDAC policy. If the WDAC policy is signed, *OslpProcessSIPolicy* verifies the integrity of the policy. This section discusses this verification process. The *SIPolicy.p7b* is in the (Public Key Cryptography Standards) PKCS#7 file format.[1] This format allows for specifying file-specific cryptographic data, such as digital signatures. Figure 2 depicts the Abstract Syntax Notation One (ASN.1) format of a digitally signed PKCS#7 file. The *SignedData* data structure contains the overall data content, including related cryptographic data. This section focusses on the *digestAlgorithms*, *contentInfo*, *certificates*, and *signerInfos* fields of *SignedData*.

```
SignedData ::= SEQUENCE {
    version: Version,
    digestAlgorithms: DigestAlgorithmIdentifiers,
    contentInfo: ContentInfo,
    certificates:
        [0] ExtendedCertificatesAndCertificates,
    crls:
        [1] CertificateRevocationLists,
    signerInfos: SignerInfos
}
```

Figure 2: ASN.1 format of a PKCS#7 file

contentInfo stores the user-generated file rules and policy rule options in binary format. This work refers to these file rules and policy rule options as WDAC content. *OslpProcessSIPolicy* verifies the integrity of the WDAC content.

[1] https://tools.ietf.org/html/rfc2315 [Retrieved: 13/9/2018]

certificates stores the certificate chain used to sign WDAC content. The certificates are stored in the X.509 format.

signerInfos stores values that describe the certificate of the signer of the WDAC content, the hash value of the WDAC content, and the signed hash of the WDAC content. Some fields referenced by signerInfos are:

- *issuerAndSerialNumber*, which stores the issuer and the serial number of the signing certificate;
- *encryptedDigest*, which stores the signed hash of the WDAC content;
- *digestAlgorithm*, which stores the hash algorithm used to calculate the hash value of the WDAC content; and
- *authenticatedAttributes*, which stores, among other things, the hash value of the WDAC content.

Figure 3 depicts a portion of a signed WDAC policy as viewed with the *openssl* utility.

```
PKCS7:
  [...]
   contents:
     type: undefined (1.3.6.1.4.1.311.79.1)
     d.other: OCTET STRING:
       0000 - 02 00 00 00 0e 37 44 a2-c9 44 06 4c b5 51 f6   .....7D..D.L.Q.
       000f - 01 6e 56 30 76 e4 f7 07-2e 4c 19 20 4d b7 c9   .nV0v....L. M..
       001e - 6f 44 a6 c5 a2 34 00 00-c1 82 00 00 00 00 00   oD...4.........
       [...]
       012c - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
       013b - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
       014a - 00 00 00 00 00 00 03 00-00 00                  ..........

   cert:
       cert_info:
         version: 2
         serialNumber: 0x1700000002AE2B61EAE06796EA000000000002
         signature:
           algorithm: sha256WithRSAEncryption (1.2.840.113549.1.1.11)
           parameter: NULL
         issuer: DC=internal, DC=test, CN=test-WIN-97VOE5A4O8L-CA
         validity:
           notBefore: Aug  3 14:06:27 2018 GMT
           notAfter: Aug  3 14:06:27 2019 GMT
         subject: CN=testDGSigningCert
         [...]
```

Figure 3: Portion of a signed WDAC policy

OslpProcessSIPolicy first invokes the *BlSIPolicyCheckPolicyOnDevice* function, which invokes *BlSIPolicyReadPolicies*. *BlSIPolicyReadPolicies* loads *SIPolicy.p7b* and returns the size and ASN.1 formatted WDAC policy. The former is stored at offset *0x30*, and the latter at *0x28* of the *rsp* register (see Figure 4).

WDAC is considered disabled if no WDAC policy is returned by *BlSIPolicyReadPolicies*. If a WDAC policy is returned, WDAC is considered enabled. Only users with administrative privileges can delete a WDAC policy and therefore, disable WDAC. When *BlSIPolicyReadPolicies* is finished executing, *BlSIPolicyCheckPolicyOnDevice* invokes *BlSIPolicyParsePolicyData*. This function processes the loaded WDAC policy.

Before *BlSIPolicyParsePolicyData* processes the WDAC policy, it verifies its integrity. The *MinCryptVerifySignedDataLMode* function initiates the verification of the WDAC policy. *MinCryptVerifySignedDataLMode* receives as parameters the size of the WDAC policy and the ASN.1 formatted WDAC policy. Figure 5 depicts the invocation of *MinCryptVerifySignedDataLMode*. The integrity verification of the WDAC policy can be structured into two phases. In the first phase, the certificate of the signer of the WDAC policy is verified. In the second phase, the integrity of the WDAC policy itself is verified.

MinCryptVerifySignedDataLMode first invokes the *MinCryptVerifyCertificateWithRootInfo* function. *MinCryptVerifyCertificateWithRootInfo* verifies the certificate of the signer of the WDAC policy signer certificate against its root certificate. The verified certificate is stored in the certificates field of the *SignedData* structure. *MinCryptVerifyCertificateWithRootInfo* uses the root certificates embedded in the Windows loader, in the *RootTable* structure. The fact that certificates embedded in the Windows loader are used for verifying the certificate used to sign the WDAC policy shows that the root of trust for verifying the integrity of the WDAC policy is the Windows loader itself.

```
kd> dd poi(@rsp + 0x30) L1
00000000`001a6dc8  00000903

kd> db poi(poi(@rsp + 0x28)) L903
fffff802`26b7fee0  30 82 08 ff 06 09 2a 86-48 86 f7 0d 01 07 02 a0  0.....*.H.......
fffff802`26b7fef0  82 08 f0 30 82 08 ec 02-01 01 31 0f 30 0d 06 09  ...0......1.0...
fffff802`26b7ff00  60 86 48 01 65 03 04 02-01 05 00 30 82 01 67 06  `.H.e......0..g.
fffff802`26b7ff10  09 2b 06 01 04 01 82 37-4f 01 a0 82 01 58 04 82  .+.....7O....X..
[...]
fffff802`26b80070  00 00 03 00 00 00 a0 82-05 84 30 82 05 80 30 82  ..........0...0.
fffff802`26b80080  04 68 a0 03 02 01 02 02-13 17 00 00 00 02 ae 2b  .h.............+
fffff802`26b80090  61 ea e0 67 96 ea 00 00-00 00 00 02 30 0d 06 09  a..g........0...
fffff802`26b800a0  2a 86 48 86 f7 0d 01 01-0b 05 00 30 52 31 18 30  *.H........0R1.0
fffff802`26b800b0  16 06 0a 09 92 26 89 93-f2 2c 64 01 19 16 08 69  .....&...,d....i
fffff802`26b800c0  6e 74 65 72 6e 61 6c 31-14 30 12 06 0a 09 92 26  nternal1.0.....&
fffff802`26b800d0  89 93 f2 2c 64 01 19 16-04 74 65 73 74 31 20 30  ...,d....test1 0
fffff802`26b800e0  1e 06 03 55 04 03 13 17-74 65 73 74 2d 57 49 4e  ...U....test-WIN
fffff802`26b800f0  2d 39 37 56 4f 45 35 41-34 4f 38 4c 2d 43 41 30  -97VOE5A4O8L-CA0
fffff802`26b80100  1e 17 0d 31 38 30 38 30-33 31 34 30 36 32 37 5a  ...180803140627Z
fffff802`26b80110  17 0d 31 39 30 38 30 33-31 34 30 36 32 37 5a 30  ..190803140627Z0
fffff802`26b80120  1c 31 1a 30 18 06 03 55-04 03 13 11 74 65 73 74  .1.0...U....test
fffff802`26b80130  44 47 53 69 67 6e 69 6e-67 43 65 72 74 30 82 01  DGSigningCert0..
[...]
```

Figure 4: Loaded SIPolicy.p7b

```
winload!MinCryptVerifySignedDataLMode:
00000000`007bd1e8 4055            push    rbp

kd> r
rax=00000000001a6a98 rbx=0000000000000903 rcx=fffff80226b7fee0
rdx=0000000000000903 rsi=00000000001a6bc0 rdi=000000000089c600
[...]

kd> kc
 # Call Site
00 winload!MinCryptVerifySignedDataLMode
01 winload!MinCrypL_CheckSignedData
02 winload!BlSIPolicyParsePolicyData
03 winload!BlSIPolicyCheckPolicyOnDevice
04 winload!OslpProcessSIPolicy
05 winload!OslPrepareTarget
06 winload!OslpMain
07 winload!OslMain
08 0x0
```

Figure 5: Invocation of MinCryptVerifySignedDataLMode

It is important to emphasize that in the scenario, where the WDAC policy is signed with a certificate that cannot be verified against a root certificate stored in *RootTable*, the certificate is considered valid without verification against an alternative root certificate.

Once *MinCryptVerifyCertificateWithRootInfo* is finished executing, the WDAC policy, that is, the WDAC content, is verified. To this end, *MinCryptVerifySignedDataLMode* first invokes the *MinCryptHashMemory* function. *MinCryptHashMemory* computes the hash value of the WDAC content, which stored in the *contentInfo* field of the *SignedData* structure. The algorithm used to calculate the hash value of the WDAC content is stored in *digestAlgorithms*.

MinCryptVerifySignedDataLMode then invokes the *I_MinCryptVerifySignerAuthenticatedAttributes* function. This function verifies the computed hash value against the hash value stored in *authenticatedAttributes*. Finally, *MinCryptVerifySignedDataLMode* invokes *MinCryptVerifySignedHash* in order to verify the signed hash of the WDAC content stored in *encryptedDigest*. To this end, it uses the previously verified signer certificate and the verified computed hash value. Only if the verifications performed by *I_MinCryptVerifySignerAuthenticatedAttributes* and *MinCryptVerifySignedHash* are successful, the WDAC content is considered authentic.

3 Windows Loader: OslpLoadAllModules

OslpLoadAllModules performs image loading and integrity verification activities. *OslpLoadAllModules* invokes *OslLoadDrivers* for loading driver executables, and *OslLoadImage* for loading any other type of image. The Windows loader loads the *ci.dll* library file in the *LoadImports* function, invoked by *OslLoadImage*. All of the previously mentioned functions ultimately invoke *BlImgLoadPEImageEx*, which performs image loading and integrity verification. Figure 6 depicts the *BlImgLoadPEImageEx* function loading *ci.dll* and its image base address (*fffff803'99b1e000*).

```
winload!BlImgLoadPEImageEx:
00000000`007eb9e4 488bc4          mov     rax,rsp

kd> r
[...]
r8=fffff80397feebf0  r9=00000000001a6810 r10=0000000000000000
[...]

kd> du @r8
fffff803`97feebf0  "\Windows\system32\CI.dll"

kd> dps 00000000001a6810 L1
00000000`001a6810  fffff803`99b1e000

kd> !dh -e fffff803`99b1e000
_IMAGE_EXPORT_DIRECTORY fffff80399ba3000 (size: 00000130)
Name: CI.dll
Characteristics: 00000000 Ordinal base: 1.
Number of Functions: 11. Number of names: 8. EAT: fffff80399ba3028.
   ordinal hint target           name
        4    0 FFFFF80399B41650 CiCheckSignedFile
        5    1 FFFFF80399B41700 CiFindPageHashesInCatalog
        6    2 FFFFF80399B41780 CiFindPageHashesInSignedFile
        7    3 FFFFF80399B41790 CiFreePolicyInfo
        8    4 FFFFF80399B41520 CiGetPEInformation
        9    5 FFFFF80399B40110 CiInitialize
       10    6 FFFFF80399B4C3D0 CiValidateFileObject
       11    7 FFFFF80399B415D0 CiVerifyHashInCatalog
        1      FFFFF80399B4C9E0 [NONAME]
        2      FFFFF80399B51AA0 [NONAME]
        3      FFFFF80399B51C80 [NONAME]
```

Figure 6: The image base address of ci.dll

Once *ci.dll* is loaded, its image base address is stored in a linked list referenced by the *LoadOrderListHead* variable. This variable is stored in the *_LOADER_PARAMETER_BLOCK* structure. Figure 7 depicts a portion of *_LOADER_PARAMETER_BLOCK* and the *LoadOrderListHead* variable referencing the image base address of *ci.dll*.

```
kd> dps poi(winload!OslLoaderBlock)
[...]
fffff801`ec894fd0  fffff801`ec94a6b0
[...]

kd> dl fffff801`ec94a6b0
[...]
fffff801`ec897a70  00000000`00000000 00000000`00000000
fffff801`ec898a60  fffff801`ec899a40 fffff801`ec897a60
fffff801`ec898a70  00000000`00000000 00000000`00000000
[...]

kd> dps fffff801`ec899a40 + 0x30 L1
fffff801`ec899a70  fffff803`99b1e000
```

Figure 7: A portion of _LOADER_PARAMETER_BLOCK and LoadOrderListHead

Once the Windows loader has transferred execution control to the kernel, it uses the populated *LoadOrderListHead* variable to pass the image base address of *ci.dll* (*fffff803'99b1e000*) to the Windows kernel for allocation of *ci.dll* in kernel's context.

4 Windows Loader: OslBuildCodeIntegrityLoaderBlock

OslBuildCodeIntegrityLoaderBlock first populates the *_LOADER_PARAMETER_CI_EXTENSION* structure with WDAC initialization parameters. These parameters are used by the kernel to further initialize WDAC. A reference to

_LOADER_PARAMETER_CI_EXTENSION and its size are stored in the *_LOADER_PARAMETER_EXTENSION* structure, in the *CodeIntegrityLoaderBlockSize* and the *CodeIntegrityLoaderBlock*, respectively (see Figure 8). The *_LOADER_PARAMETER_EXTENSION* structure is referenced by the *Extension* variable. This variable is stored in *_LOADER_PARAMETER_BLOCK*, at offset *0xF0* (see Figure 8).

```
typedef struct _LOADER_PARAMETER_BLOCK {
    [...]
    PLOADER_PARAMETER_EXTENSION         Extension;                              // 0xF0
    [...]
} LOADER_PARAMETER_BLOCK, * PLOADER_PARAMETER_BLOCK


typedef struct _LOADER_PARAMETER_EXTENSION {
    [...]
    PLOADER_PARAMETER_CI_EXTENSION      CodeIntegrityLoaderBlock;               // 0x9D8
    ULONG32                             CodeIntegrityLoaderBlockSize;           // 0x9E0
    [...]
} LOADER_PARAMETER_EXTENSION, * PLOADER_PARAMETER_EXTENSION;


typedef struct _LOADER_PARAMETER_CI_EXTENSION {
    [...]
    UINT8                       CodeIntegrityPolicyHash[32];                    // 0x0020
    ULONG32                     CodeIntegrityPolicyType                         // 0x1338
    ULONG32                     CodeIntegrityPolicySize                         // 0x133c
    UINT8                       CodeIntegrityPolicy[CodeIntegrityPolicySize]    // 0x1340
    [...]
} LOADER_PARAMETER_CI_EXTENSION, * PLOADER_PARAMETER_CI_EXTENSION;
```

Figure 8: Relevant _LOADER_PARAMETER_* structures

The *OslBuildCodeIntegrityLoaderBlock* function populates *_LOADER_PARAMETER_CI_EXTENSION* with WDAC initialization parameters, such as:

- *CodeIntegrityPolicyHash*: This parameter stores the hash value of the WDAC content. This hash is calculated in the *OslpCalculateCodeIntegrityPolicyHash* function, invoked by *OslBuildCodeIntegrityLoaderBlock*;
- *CodeIntegrityPolicySize*: This parameter stores the size of the WDAC content; and
- *CodeIntegrityPolicy*: This parameter stores the WDAC content extracted from *contentInfo*.

After *OslBuildCodeIntegrityLoaderBlock* has finished executing, the Windows loader transfers the execution control to the kernel. The kernel uses the populated *_LOADER_PARAMETER_CI_EXTENSION* structure, ultimately referenced by *_LOADER_PARAMETER_BLOCK* to further initialize WDAC.

5 Windows Kernel: MiReloadBootLoadedDrivers

After execution control has been transferred to the kernel, it invokes the *InitBootProcessor* function. This function is responsible for conducting relevant tasks, for example, initializing memory management functionalities. *InitBootProcessor* ultimately invokes the memory management routine *MmInitSystem*. This routine, in turn, invokes *MiReloadBootLoadedDrivers*. This function allocates *ci.dll* in the context of the kernel based on the image base address of *ci.dll* (see, for example, *fffff803`99b1e000* in Figure 6), passed by the Windows loader.

MiReloadBootLoadedDrivers invokes the *MiUpdateThunks* function, which allocates *ci.dll* in the context of the kernel. Figure 9 depicts the invocation of *MiUpdateThunks*. The second parameter of *MiUpdateThunks* (*rdx* in Figure 9) is the image base address of *ci.dll* passed by the Windows loader, whereas the third (*r8* and *fffff808`c5fd0000* in Figure 9) is an address in the context of the kernel, where *ci.dll* is to be allocated.

Once *ci.dll* is allocated in the kernel's context, the kernel invokes the *SepInitializeCodeIntegrity* function. This function initializes the interface exposed by *ci.dll*, after which the kernel can use code integrity functionalities.

It is important to emphasize that the integrity of *ci.dll* is verified by the Windows loader. This shows that the root of trust for verifying the integrity of *ci.dll* is the Windows loader.

```
nt!MiUpdateThunks:
fffff803`9961e8c0 48895c2408      mov     qword ptr [rsp+8],rbx

kd> r
[...]
rdx=fffff80399b1e000 rsi=fffff80399b1e000 rdi=0000000000000006
 r8=fffff808c5fd0000  r9=00000000000a0000 r10=0000000000000000

kd> lm v m CI
start             end                 module name
fffff808`c5fd0000 fffff808`c6070000   CI         (deferred)
    Image path: CI.dll
    Image name: CI.dll
    Timestamp:        Tue Mar  6 06:25:49 2018 (5A9E265D)
    CheckSum:         0009D5DB
    ImageSize:        000A0000
    File version:     10.0.14393.2155
    Product version:  10.0.14393.2155
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.7 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      Microsoft Corporation
        ProductName:      Microsoft® Windows® Operating System
        InternalName:     ci.dll
        OriginalFilename: ci.dll
        ProductVersion:   10.0.14393.2155
        FileVersion:      10.0.14393.2155 (rs1_release_1.180305-1842)
        FileDescription:  Code Integrity Module
        LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

Figure 9: Relocated ci.dll file

6 Windows Kernel: SeCodeIntegrityInitializePolicy

After *ci.dll* has been allocated in the kernel's context and the interface exposed by it is available to the kernel, the kernel initializes the WDAC policy. The *SeCodeIntegrityInitializePolicy* function initializes the WDAC policy. This involves storage of the WDAC policy in the context of the kernel.

SeCodeIntegrityInitializePolicy receives as parameter the *_LOADER_PARAMETER_BLOCK* structure, populated and passed by the Windows loader (*KeLoaderBlock* in Figure 10). This structure ultimately references *_LOADER_PARAMETER_CI_EXTENSION* (*CodeIntegrityLoaderBlock* in Figure 10), which, among other things, stores *CodeIntegrityPolicy* and *CodeIntegrityPolicyHash*. *CodeIntegrityPolicy* stores the WDAC content itself.

```
SeCodeIntegrityInitializePolicy(KeLoaderBlock)
{
    [...]
    Extension = *(_LOADER_PARAMETER_EXTENSION *)(KeLoaderBlock + 0xF0);
    [...]
    CodeIntegrityLoaderBlock = *(_LOADER_PARAMETER_CI_EXTENSION *)(Extension + 0x9D8)
    [...]
    if ( CiInitializePolicy )
    {
        [...]
        CiInitializePolicy(CodeIntegrityLoaderBlock, [...]);
        [...]
    }
    [...]
}
```

Figure 10: Pseudo-code of the implementation of SeCodeIntegrityInitializePolicy

SeCodeIntegrityInitializePolicy invokes the *CiInitializePolicy* function. This function receives the *_LOADER_PARAMETER_CI_EXTENSION* structure as parameter. *CiInitializePolicy* populates the *ci.dll* variables *g_SiPolicyHandles* and *g_SiPolicyHash* with the values stored in the *CodeIntegrityPolicy* and *CodeIntegrityPolicyHash* variables, respectively. An analysis of the WDAC initialization functionalities showed that the hash value stored in *CodeIntegrityPolicyHash* is not used for verifying the integrity of the WDAC content stored in *CodeIntegrityPolicy*.

Figure 11 depicts a portion of a populated *g_SiPolicyHandles* variable. Once *g_SiPolicyHandles* is populated, the Windows kernel can use the WDAC content stored in *g_SiPolicyHandles* for verification purposes. The description of each of the fields of *g_SiPolicyHandles* is out of the scope of this work.

It is important to emphasize that the integrity of the WDAC content is verified by the Windows loader. This shows that the root of trust for verifying the integrity of the WDAC content is the Windows loader.

```
ci!g_SiPolicyHandles
   +0x000 PlatformID                : nt!_GUID
   +0x010 PolicyTypeID              : nt!_GUID
   [...]
   +0x02c RuleOptionFlags           : Uint4B
   [...]
   +0x068 CodeIntegrityPolicySize   : Int4B
   +0x06c CodeIntegrityPolicy       : Ptr64 Void
```

Figure 11: g_SiPolicyHandles

References

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