ERNW
providing security.

ELAM: The Windows Defender ELAM Driver

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 the Trusted Platform Module (TPM), 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

In this work, we discuss the early launch anti-malware (ELAM) technology implemented in Windows 10. This technology, implemented in a driver referred to as the ELAM driver, is used for checking drivers for malware and is part of the booting process of Windows 10. Therefore, it is critical for the security of the boot process of the system. We provide an overview of the decision-making process of the ELAM driver for categorizing an executable as benign or malicious. We also provide an overview of the structure of the malware database that this driver uses, and of the verification of the integrity of this database.

The ELAM driver is initialized before the majority of the other drivers and is used for checking for malware the drivers loaded after it. An ELAM driver implements anti-malware technology, for example, detection of known malicious driver images based on searching a database of properties of such images (e.g., image hashes). This is the ELAM database of malware signatures.

Any vendor may develop an ELAM driver following the development guidelines provided by Microsoft.[1] The

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]

Windows 10 system is distributed with an ELAM driver that is part of the Microsoft's Windows Defender anti-malware technology.[2]

2 The Windows Defender ELAM Driver

The Windows Defender ELAM driver is implemented in the *%SystemRoot%\System32\drivers\WdBoot.sys* executable. It is loaded by the Windows loader and initialized in the kernel. It is unloaded by the kernel when it has checked for malware all boot drivers. Figure 1 depicts the information on the Windows Defender ELAM Driver that the kernel maintains after loading it.

```
kd> lmDvmWdBoot
Browse full module list
start             end                 module name
fffff800`0a700000 fffff800`0a710000   WdBoot     (deferred)
    Image path: WdBoot.sys
    Image name: WdBoot.sys
    [...]
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft® Windows® Operating System
    InternalName:     WdBoot
    OriginalFilename: WdBoot.sys
    ProductVersion:   4.10.14393.0
    FileVersion:      4.10.14393.0 (rs1_release.160715-1616)
    FileDescription:  Microsoft antimalware boot driver
    LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

Figure 1: Information on the Windows Defender ELAM driver

The ELAM driver performs anti-malware verification in its function *MpEbBootDriverCallback*. This is a callback function invoked when the kernel notifies the ELAM driver that a driver image is to be initialized. The kernel notifies the ELAM driver that it initializes a driver image that needs to be verified. The notification is implemented in the *PnpNotifyEarlyLaunchImageLoad* kernel function.

Figure 2 depicts a pseudo-code of the implementation of *MpEbBootDriverCallback* and relevant functions that it invokes ([1] in Figure 2). *MpEbBootDriverCallback* invokes *EbLookupProperty*, which sets an integer code indicating the category of the verified image. Adopting the Microsoft terminology on this topic, an image may be categorized as:

- a known good image (code value *BdCbClassificationKnownGoodImage* in Figure 2);
- a known bad image (code value *BdCbClassificationKnownBadImage* in Figure 2);
- a known bad image of a boot-critical driver (code value *BdCbClassificationKnownBadImageBootCritical* in Figure 2); or
- an unknown image (code value *BdCbClassificationKnownBadImage* in Figure 2).

We refer to these categories as ELAM image categories. *MpEbBootDriverCallback* passes the integer set by *EbLookupProperty* back to the kernel.Based on the image categorization done by the ELAM driver, the kernel decides whether the image of a given boot driver will be initialized in the PnpDoPolicyCheck kernel function. The kernel brings a decision based on the configuration of the ELAM group policy located at the policy path: *Computer Configuration -> Administrative Templates -> System -> Early Launch Antimalware*. For example, users

[2]https://docs.microsoft.com/en-us/windows/threat-protection/windows-defender-antivirus/windows-defender-antivirus-in-windows-10; https://blogs.technet.microsoft.com/dubaisec/2016/05/09/elam-driver/ [Retrieved: 22/9/2017]

may configure this policy such that only known good images are initialized.[3] If the ELAM group policy allows for it, the boot driver is initialized.

[1]

```
MpBinarySearch(driverProperties) { g_lookupList.find(driverProperties); }

EbLookupProperty(...)
{
    [...]

    result = MpBinarySearch(driverProperties);
    if(!result) return 0;
    else return result->classification;
}

MpEbBootDriverCallback(...)
{
  [...]

  switch EbLookupProperty(...)
  {
      case 0:  BdCbClassificationKnownGoodImage
      case 1:  BdCbClassificationKnownBadImage
      case 4:  BdCbClassificationKnownBadImageBootCritical
      default: BdCbClassificationUnknownImage
  }

  [...]
}
```

[2]

```
addLookupEntryToList(item) { g_lookupList.add(item);}

EbLoadSignatureData(...)
{
    [...]

    signaturedata = EbAuthenticateSignatureData(...);
    foreach item in signaturedata
    { addLookupEntryToList(item);}

    [...]
}
```

Figure 2: Pseudo-code of *MpEbBootDriverCallback* and functions that it invokes

EbLookupProperty performs a binary search for specific properties of the verified driver image, for example, file hash values, in the ELAM database of malware signatures (*MpBinarySearch*, *driverProperties* in Figure 2). Each entry of this database is stored in an array (*g_lookupList* in Figure 2). This array is filled when the signature database is loaded into the driver's memory space.

The loading of the ELAM signature database is done when the driver is initialized and is performed in two steps ([2] in Figure 2): First, the function *MpEbLoadSignatures* (not depicted in Figure 2) using the *ZwOpenKey* routine[4] loads the signature database. This database is stored in the form of a registry hive in the system's registry. Then, the function *EbLoadSignatureData* verifies the integrity of the signature database by invoking *EbAuthenticateSignatureData*. It also stores each entry of the database (item in Figure 2) in the previously mentioned array of database entries (*g_lookupList*, *addLookupEntryToList* in Figure 2). The stored entries are later used in *EbLookupProperty*.

Figure 3 depicts this database loading operation. The ELAM signature database is loaded in the registry as the registry hive *HKEY_LOCAL_MACHINE\ELAM* (*\REGISTRY\MACHINE\ELAM* in Figure 3). We observed that this registry hive is unloaded when the ELAM driver is unloaded. The registry hive of the ELAM signature database

[3] https://blogs.technet.microsoft.com/dubaisec/2016/05/09/elam-driver/ [Retrieved: 22/9/2017]

[4] https://msdn.microsoft.com/en-us/library/windows/hardware/ff567014(v=vs.85).aspx [Retrieved: 22/9/2017]

is stored on the file-system at *%SystemRoot%\System32\Config\elam* (see Figure 3). This enabled us to analyze the contents of the registry hive after the ELAM driver and its signature database have been unloaded.

Figure 3: The ELAM signature database in the registry

The ELAM signature database is stored as binary data. This data has a specific format structuring it into multiple entries. Each entry, among other things, consists of a malware signature, an entry type, and a trust level. A malware signature is often, but not necessarily, a hash value. An entry type is a code value indicating the type of data representing a malware signature. A trust level is a code value corresponding to the ELAM image categories known good image, known bad image, known bad image of a boot-critical driver, and unknown image. The trust level specifies the ELAM category of image associated with the malware signature.

We observed the above by parsing the signature database of the Windows Defender ELAM driver using a parser we developed. The code of this parser, implemented in the Python programming language, is placed in the Appendix, section 'ELAM Database Parser'. Figure 4 depicts the output of the parser, where *EntryType*, *TrustLevel*, and *data* mark an entry type, a trust level, and a malware signature, respectively. An example entry in the ELAM signature database is the hash value *F4 27 86 [...] 9F 8E*, which is a hash value of a known bad image.

```
[...]

Item:
    code: 800276E8
    type: <EntryType.IMAGE_HASH: 4>
    trust: <TrustLevel.KnownBadImage: 1>
    data:
        0  1  2  3  4  5  6  7
    00: 5D 67 6C 34 3D 71 9D EC ]gl4=q..
    08: 84 3C D5 A6 CD 2E 69 3E .<....i>
    10: 6D AC EE B3             m...
    comment: b''
Item:
    code: 800276E8
    type: <EntryType.IMAGE_HASH: 4>
    trust: <TrustLevel.KnownBadImage: 1>
    data:
        0  1  2  3  4  5  6  7
    00: F4 27 86 15 7B CF 37 A9 .'..{.7.
    08: 2D 31 38 E5 95 9E 3B 22 -18...;"
    10: D1 CA 9F 8E             ....
    comment: b''

[...]
```

Figure 4: The ELAM signature database

As we previously mentioned, the integrity of the ELAM signature database is verified in *EbAuthenticateSignatureData*. The signature database of the Windows Defender ELAM driver is digitally signed for preventing unauthorized modifications. Figure 5 depicts pseudo-code of the implementation of *EbAuthenticateSignatureData*. This

function verifies the signature of the ELAM signature database using the kernel implementation of the Cryptographic API: Next Generation (CNG) library. The ELAM signature database may be updated by system services, such as *Windows Update*. These services mount and modify the registry hive *HKEY_LOCAL_MACHINE\ELAM*. However, the updated signature database has to be properly signed since its integrity is verified by the ELAM driver as discussed in this work.[5]

```
EbAuthenticateSignatureData(...)
{
    [...]

    BCryptOpenAlgorithmProvider(..., "SHA1", "Microsoft Primitive Provider", ...);

    [...]

    BCryptHashData(...);

    [...]

    BCryptOpenAlgorithmProvider(..., "RSA", "Microsoft Primitive Provider", ...);

    BCryptImportKeyPair(..., "RSAPUBLICBLOB", ...);

    [...]

    BCryptVerifySignature(...);

    [...]
}
```

Figure 5: Pseudo-code of *EbAuthenticateSignatureData*

[1]

```
WdBoot!EbAuthenticateSignatureData+0x3ac:
fffff803`d5568948 488d05f1a8ffff  lea     rax,[WdBoot!g_MpPublicKeyRaw (fffff803`d5563240)]
[...]
WdBoot!EbAuthenticateSignatureData+0x3cc:
fffff803`d5568968 4889442420      mov     qword ptr [rsp+20h],rax
[...]
WdBoot!EbAuthenticateSignatureData+0x3d1:
fffff803`d556896d ff1595d6ffff    call    qword ptr [WdBoot!_imp_BCryptImportKeyPair (fffff803`d5566008)]
```

[2]

```
.rdata:00000001C0003240 g_MpPublicKeyRaw db 52h, 53h, 41h, 31h  ; Magic
.rdata:00000001C0003240                                         ; DATA XREF: EbAuthenticateSignatureData+58o
.rdata:00000001C0003240                                         ; EbAuthenticateSignatureData+3ACo
.rdata:00000001C0003240                 dd 800h                 ; BitLength
.rdata:00000001C0003240                 dd 3                    ; cbPublicExp
.rdata:00000001C0003240                 dd 100h                 ; cbModulus
.rdata:00000001C0003240                 dd 0                    ; cbPrime1
.rdata:00000001C0003240                 dd 0                    ; cbPrime2
```

Figure 6: The public key used for verification of the Windows Defender ELAM signature database

EbAuthenticateSignatureData first hashes the signature database (*BcryptHashData*[6] in Figure 5) using the Secure Hash Algorithm (SHA)-1 algorithm and the software-implemented *Microsoft Primitive Provider* (*BcryptOpenAlgorithmProvider* in Figure 5). It then imports a public key (*BCryptImportKeyPair*, *RSAPUBLICBLOB*[7] in Figure 5). Finally, *EbAuthenticateSignatureData* uses the *Microsoft Primitive Provider* and the imported public key to decrypt the signature of the ELAM signature database and compare it with the previously calculated hash value of this database (*BCryptVerifySignature*[8] in Figure 5).

[5] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]

[6] https://msdn.microsoft.com/en-us/library/windows/desktop/aa375468(v=vs.85).aspx [Retrieved: 22/9/2017]

[7] https://msdn.microsoft.com/en-us/library/windows/desktop/aa375472(v=vs.85).aspx [Retrieved: 22/9/2017]

[8] https://msdn.microsoft.com/de-de/library/windows/desktop/aa375515(v=vs.85).aspx [Retrieved: 22/9/2017]

We observed that *EbAuthenticateSignatureData* imports a public key hardcoded in the *WdBoot.sys* driver executable, stored in the variable *g_MpPublicKeyRaw*. This indicates that the root of trust for verifying the malware signature database used by the Windows ELAM driver is the driver itself. Figure 6 depicts the passing of *g_MpPublicKeyRaw* to *BcryptImportKeyPair* for importing ([1] in Figure 6). It also depicts the declaration of *g_MpPublicKeyRaw* in the *WdBoot.sys* driver executable, which we observed with the *IDA* disassembler ([2] in Figure 6).

According to the Microsoft's ELAM driver development guidelines, vendors of ELAM drivers may store the ELAM signature database and related relevant data in the registry keys *Measured*, *Policy*, and/or *Config*, under the registry hive *HKEY_LOCAL_MACHINE\ELAM*. We observed that the Windows Defender ELAM driver stores its database in *Measured*.

Appendix

ELAM Database Parser

This script can also be found in the folder *files* of the *Windows Insight* file repository, under the name *elam_-defender_parser.py*.

```
import argparse
import struct
from collections import namedtuple
import enum


from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
from Crypto.Util.number import bytes_to_long, long_to_bytes

from helperlib import print_hexII, print_hexdump, hexdump


class Item(namedtuple('Item', ['type', 'trust_code', 'data', 'comment'])):
    def __str__(self):
        l = [
            'DB entry:',
            f'\tType: {self.type!r}',
            f'\tImage category: {self.trust_code!r}',
            f'\tData:\n\t' + '\n\t'.join(hexdump(self.data, header=True)),
            f'\tComment: {self.comment}',
        ]
        return '\n'.join(l)


class EntryType(enum.IntEnum):
    THUMBPRINT_HASH = 1
    CERTIFICATE_PUBLISHER = 1
    ISSUER_NAME = 1
    IMAGE_HASH = 4
    VERSION_INFO = 7


class TrustLevel(enum.IntEnum):
    KnownGoodImage = 0
    KnownBadImage = 1
    UnknownImage_2 = 2
    UnknownImage = 3
    KnownBadImageBootCritical = 4


PUBLIC_EXPONENT = 0x010001

MODULUS = int.from_bytes(struct.pack("256B", *[
    0xb3, 0x95, 0xde, 0x5b, 0xc2, 0xe1, 0x89, 0xf7, 0x56, 0xc2, 0x20,
    0xbf, 0x27, 0xd2, 0x88, 0x1a, 0x0a, 0xac, 0xdb, 0xc7, 0x19, 0x36,
    0x7b, 0xce, 0x37, 0x83, 0xd1, 0xec, 0x42, 0xd3, 0xab, 0x30, 0x54,
    0xa5, 0x51, 0x11, 0xd8, 0xcc, 0xec, 0x80, 0xab, 0x89, 0x5a, 0xae,
    0x18, 0x71, 0x11, 0x7c, 0x85, 0x1a, 0x1a, 0x53, 0x54, 0x46, 0x3e,
    0x55, 0x5c, 0x43, 0x5d, 0x4b, 0x9f, 0xc7, 0x54, 0x57, 0x75, 0xc5,
    0x02, 0xe2, 0x63, 0xa9, 0x94, 0x56, 0xa7, 0x3b, 0xe0, 0xc3, 0xed,
    0x5f, 0x66, 0x9d, 0x60, 0x78, 0x1e, 0xac, 0x92, 0x3d, 0x48, 0xe9,
    0x51, 0x5d, 0x79, 0x2a, 0x22, 0x9a, 0x9e, 0xd3, 0xbc, 0x15, 0xbe,
    0x7a, 0x4e, 0x97, 0xe8, 0x1f, 0x9c, 0x80, 0xf5, 0xfb, 0x94, 0x0b,
    0x5f, 0xb7, 0x6f, 0x0d, 0x57, 0xa0, 0x09, 0x55, 0x68, 0x78, 0xf3,
    0x5d, 0x7b, 0x9a, 0x9b, 0x08, 0xa3, 0xa6, 0x41, 0x18, 0xf0, 0x17,
    0x11, 0x89, 0x9b, 0x71, 0x73, 0x27, 0xa2, 0x55, 0x51, 0xc0, 0xee,
    0xa5, 0x70, 0x6f, 0xb8, 0x40, 0x2a, 0x85, 0xe9, 0x91, 0x20, 0x4b,
    0x0c, 0xd2, 0x29, 0xa2, 0x01, 0x36, 0x96, 0x1c, 0xbb, 0xd5, 0xef,
    0x95, 0x68, 0x43, 0xfb, 0x77, 0x42, 0x88, 0x1a, 0xae, 0x60, 0x14,
    0xfe, 0x0b, 0x0d, 0xd3, 0x28, 0x04, 0x98, 0x15, 0x71, 0x3e, 0xba,
    0xb3, 0x80, 0x65, 0x6d, 0x2b, 0x7f, 0x30, 0xca, 0xf2, 0x6c, 0xa6,
    0x47, 0xd3, 0x3c, 0x57, 0x50, 0x0d, 0xb3, 0xbb, 0xed, 0x6d, 0x75,
    0xf2, 0x0f, 0x26, 0x29, 0xf7, 0xc6, 0xe4, 0x20, 0x5e, 0xaf, 0x87,
    0xf1, 0x8b, 0x8e, 0x57, 0x99, 0x00, 0xf3, 0x84, 0xe5, 0x25, 0x10,
    0x05, 0x2c, 0xeb, 0x77, 0xa3, 0xdb, 0xbd, 0x7e, 0xd4, 0xb5, 0x60,
    0xb6, 0x6a, 0xa0, 0x99, 0x25, 0x59, 0x2f, 0x10, 0x69, 0xf4, 0x62,
    0xe1, 0x8c, 0x2b]), 'big')


MODULUS = int.from_bytes(
    bytes.fromhex('b395de5bc2e189f756c220bf27d2881a0aacdbc719367bce3783d1ec42d3ab3054a55111d8ccec80ab895aae1871117c851a1a5354463e555c435d4b9fc7545775c502e263
a99456a73be0c3ed5f669d60781eac923d48e9515d792a229a9ed3bc15be7a4e97e81f9c80f5fb940b5fb76f0d57a009556878f35d7b9a9b08a3a64118f01711899b717327a25551c0eea5706
fb8402a85e991204b0cd229a20136961cbbd5ef956843fb7742881aae6014fe0b0dd328049815713ebab380656d2b7f30caf26ca647d33c57500db3bbed6d75f20f2629f7c6e4205eaf87f18b
8e579900f384e52510052ceb77a3dbbd7ed4b560b66aa0992592f1069f462e18c2b'),'big')


PUBLIC_KEY = RSA.construct((MODULUS, PUBLIC_EXPONENT))


def parse(fp):
    tag = fp.read(1)[0]
    size = struct.unpack('<I', fp.read(3) + b'\0')[0]
    return tag, fp.read(size)


def main(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument('Database', type=argparse.FileType('rb'))

    args = parser.parse_args(args=argv)
    fp = args.Database
```

```
    while True:
        try:
            tag, data = parse(fp)

            if tag == 0xA9:
                assert len(data) >= 4
                offset = struct.unpack_from('<I', data)[0] + 4
                assert offset < len(data)
                some_type = struct.unpack_from('<B', data[offset:])[0]
                if some_type == 9:
                    data_type, trust_code = struct.unpack_from('<BB', data[4:])
                    entry_data = data[6:offset]
                    item = Item(EntryType(data_type), TrustLevel(trust_code),
                                entry_data, data[offset + 2:])
                    print(str(item))

            elif tag == 0xAC:
                print("Encrypted signature:")
                print_hexdump(data, colored=True, header=True)
                signature = PUBLIC_KEY.encrypt(bytes(reversed(data)), None)[0]
                print("\n");

                print("Decrypted signature (DER):")
                print_hexdump(signature, colored=True, header=True, folded=True)
                print("\n");

                signature = signature.split(b'\x00', 1)[1]
                try:
                    assert signature[0] == 0x30
                    l = signature[1]
                    signature = signature[2:2+l]
                    assert signature[0] == 0x30

                    l = signature[1]
                    algo = signature[2:2+l]
                    hashsum = signature[2+l:]

                    assert hashsum[0] == 0x4
                    l = hashsum[1]
                    hashsum = hashsum[2:l+2]
                    print("Hash:", hashsum.hex())
                    print("\n");

                except:
                    pass

            elif tag == 0x5C:
                continue;

            elif tag == 0x5D:
                continue;

            else:
                raise ValueError("Parsing error. Unknown tag {:02x}".format(tag))
        except IndexError:
            break

if __name__ == '__main__':
    main()
```