

BIOS practice: implementation of secure boot function

原创

Anthony

Modified on 2022-03-02 20:08:18

Read 1.2w

Collection 35

Likes 6

Copyright CC 4.0 BY-SA

Category Column: BIOS learning practice

Article Tags: UEFI

2048 AI社区

文章已被社区收录

加入社区

BIOS learning practice

This column includes this content

21 articles








Subscribe to our column

What is Secure Boot?

Secure boot, as the name implies, is a secure boot. Secure boot is a security standard developed by members of the PC industry to help ensure that devices only boot with software trusted by the original equipment manufacturer (OEM). When the PC starts, the firmware checks the signature of each startup software, including UEFI firmware drivers (also known as Option ROMs), EFI applications, and operating systems. If the signature is valid, the PC will start and the firmware will hand over control to the operating system.

OEMs can use the firmware manufacturer's instructions to create Secure Boot keys and store them in the PC firmware. When adding UEFI drivers, you also need to ensure that they are signed and included in the Secure Boot database.

First, let's look at the certificates:

 MicCorKEKCA2011_2011-06-24.crt	2020/11/18 13:40	安全证书	2 KB
 MicCorUEFCA2011_2011-06-27.crt	2020/11/18 13:40	安全证书	2 KB
 MicWinProPCA2011_2011-10-19.crt	2020/11/18 13:40	安全证书	2 KB
 pk.der	2020/11/18 13:40	安全证书	1 KB
 pk.pem	2020/11/18 13:40	PEM 文件	2 KB
 pk.pfx	2020/11/18 13:40	Personal Inform...	3 KB
 UOS-UEFI-RSA.der	2020/11/18 13:40	安全证书	2 KB

CSDN @潇洒Anthony

And where they are placed, these are placed in the fdf file, indexed by guid:

```
!if $(SECURE_BOOT_ENABLE) == TRUE
FILE FREEFORM = PCD(gPhytiumPlatformTokenSpaceGuid.PcdSecureKeyPKFile) {
    SECTION RAW = $(SECURE_KEY_PATH)/pk.der
    SECTION UI = "PKpub.der"
}
FILE FREEFORM = PCD(gPhytiumPlatformTokenSpaceGuid.PcdSecureKeyMSKEKFile) {
    SECTION RAW = $(SECURE_KEY_PATH)/MicCorKEKCA2011_2011-06-24.crt
    SECTION UI = "MicCorKEKCA2011_2011-06-24.crt"
}
FILE FREEFORM = PCD(gPhytiumPlatformTokenSpaceGuid.PcdSecureKeyMSProFile) {
    SECTION RAW = $(SECURE_KEY_PATH)/MicWinProPCA2011_2011-10-19.crt
    SECTION UI = "MicWinProPCA2011_2011-10-19.crt"
}
FILE FREEFORM = PCD(gPhytiumPlatformTokenSpaceGuid.PcdSecureKeyMSUEFFile) {
    SECTION RAW = $(SECURE_KEY_PATH)/MicCorUEFCA2011_2011-06-27.crt
    SECTION UI = "MicCorUEFCA2011_2011-06-27.crt"
}
FILE FREEFORM = PCD(gPhytiumPlatformTokenSpaceGuid.PcdSecureKeyMSDBXFile) {
    SECTION RAW = $(SECURE_KEY_PATH)/DBXUpdate.bin
    SECTION UI = "DBXUpdate"
}
FILE FREEFORM = B4E606D1-4D39-41cd-8D7C-46F61DCD7A7C {
    SECTION RAW = $(SECURE_KEY_PATH)/UOS-UEFI-RSA.der
    SECTION UI = "UosDB"
}
!endif
```

CSDN @潇洒Anthony

Keys are in pairs, namely public keys and private keys. The public key is placed in the BIOS, and the private key is used to sign the bootloader, driver, and program that need to run on the motherboard.

How to verify that secure boot is effective

Use a USB flash drive to create a shell disk. Under the premise of enabling the installation key in secure boot, you cannot enter the shell, but you can enter the signed shell program normally.

Type for Secure Boot

↓

Database Key (db) - it is used to sign or verify running binaries (boot loaders, boot managers, shells, drivers, etc.). db can hold multiple KEY values - this is an important fact for some purposes. Note that db can contain both public keys (matching private keys that can be used to sign multiple binaries) and hash values (to describe a single binary).

Database Blacklist (dbx) - dbx is an anti-database; it contains keys and hashes corresponding to known malware or other undesirable software. The keys or hashes can be installed just like db. If a binary matches a key or hash that exists in both db and dbx, dbx should take precedence.

Key Exchange Key (KEK) - KEK is used to sign keys (public keys) so that the firmware will consider them valid when entering them into the database (db or dbx). Without KEK, the firmware will have no way of knowing if the new key is valid or provided by malware. Therefore, secure boot will be a joke without KEK or require the database to remain static (validated when updating the db database). Since the critical point of secure boot is dbx, a static database will not be available. Computers usually have two keks, one from the system manufacturer and one from the motherboard manufacturer. This allows either party to release updates.

Platform Key (PK) - PK is the top-level key in secure boot, and its function in relation to KEK is similar to that of KEK with db and dbx. UEFI secure boot supports a single PK, which is usually provided by the motherboard manufacturer. Therefore, only the motherboard manufacturer can fully control the computer. An important part of controlling the secure boot process yourself is to use the PK generated by yourself and place it in the motherboard flash, and then verify it step by step to ensure the reliability of secure boot.

Code Comb

Preparation before safe start:

The main preparation conditions are in the dsc and fdf files, that is, what files we need to support

1. Libraries that need support

		AI generated projects	登录复制
1	IntrinsicLib CryptoPkg/Library/IntrinsicLib/IntrinsicLib.inf		
2	AuthVariableLib SecurityPkg/Security/Library/AuthVariableLib26/AuthVariableLib.inf		
3	BaseCryptLib CryptoPkg/Library/BaseCryptLib/BaseCryptLib.inf		
4	PlatformSecureLib SecurityPkg/Library/PlatformSecureLib/PlatformSecureLib.inf		

2. Policy definition selection, that is, which files require what kind of verification method

		AI generated projects	登录复制
1	gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy 0x04		
2	gEfiSecurityPkgTokenSpaceGuid.PcdFixedMediaImageVerificationPolicy 0x04		
3	gEfiSecurityPkgTokenSpaceGuid.PcdRemovableMediaImageVerificationPolicy 0x04		

2. The configuration file of secure boot, which includes the display of the interface, the loading of files in Setup mode (factory mode) and User mode (user mode) under secure boot, the establishment of the database and the generation of variables. This involves how to import the previously mentioned certificates into the firmware through indexes to establish the database and store it in the nvrom area.

		AI generated projects	登录复制
Security/SecureBootConfigDxe/SecureBootConfigDxe.inf			

3. Verify the signature of the file, that is, verify whether the executed file has been signed:

		AI generated projects	登录复制
1	MdeModulePkg/Universal/SecurityStubDxe/SecurityStubDxe.inf {		
2	<LibraryClasses>		
3	NULL SecurityPkg/Library/DxeImageVerificationLib/DxeImageVerificationLib.inf		
4	!if \$(TPM_SUPPORT) == TRUE		
5	NULL SecurityPkg/Library/DxeTpm2MeasureBootLib/DxeTpm2MeasureBootLib.inf		
6	!endif		
7	}		

4. In addition to the normal botloader and driver, the machine can also be started through the network. This part also needs to be signed before it can run, which requires the following files.

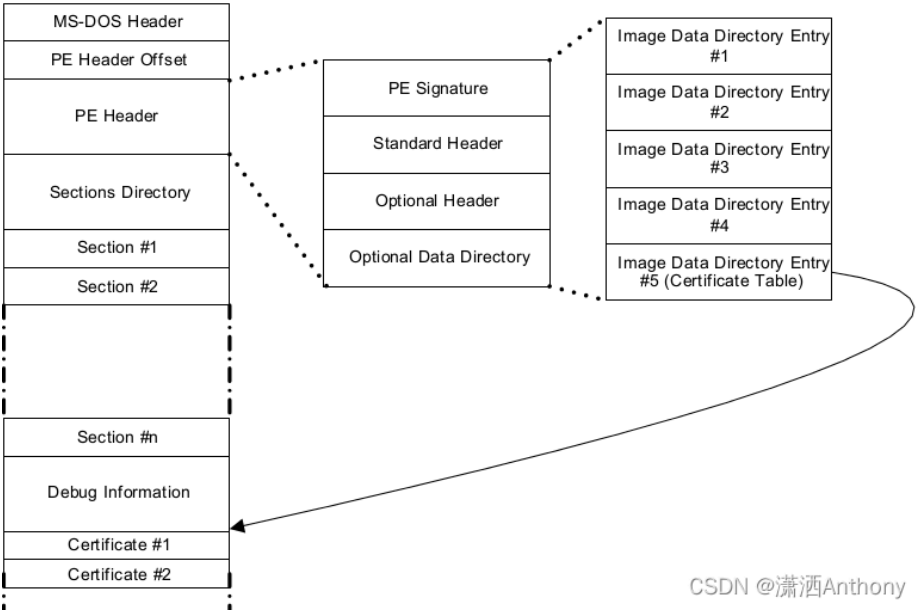
		AI generated projects	登录复制
1	NetworkPkg/TlsDxe/TlsDxe.inf		
2	NetworkPkg/TlsAuthConfigDxe/TlsAuthConfigDxe.inf		

5. nvrom opens up an area for secure boot

		AI generated projects	登录复制
1	gEfiMdeModulePkgTokenSpaceGuid.PcdMaxVariableSize 0x10000		
2			
3	NorFlashDxe/NorFlashAuthenticatedDxe.inf		

How to verify the signature:

Before loading and generating the DB database, you need to understand how the file is signed. First look at a picture, and then understand the content in the picture according to the code:



CSDN @潇洒Anthony

Let's take option ROM file verification as an example. In the above text, we saw PcdOptionRomImageVerificationPolicy. We searched in the code and found the location: DxelImageVerificationHandler function under DxelImageVerificationLib.C:

```
1 //
2 // Check the image type and get policy setting.
3 //
4 switch (GetImageType (File)) {
5
6 case IMAGE_FROM_FV:
7     Policy = ALWAYS_EXECUTE;
8     break;
9
10 case IMAGE_FROM_OPTION_ROM:
11     Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);
12     break;
13
14 case IMAGE_FROM_REMOVABLE_MEDIA:
15     Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);
16     break;
17
18 case IMAGE_FROM_FIXED_MEDIA:
19     Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);
20     break;
21
22 default:
23     Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
24     break;
25 }
```

收起 ^

Look at the definition of Policy value:

```
1 //
2 // Image type definitions.
3 //
4 #define IMAGE_UNKNOWN 0x00000001
5 #define IMAGE_FROM_FV 0x00000002
6 #define IMAGE_FROM_OPTION_ROM 0x00000004
7 #define IMAGE_FROM_REMOVABLE_MEDIA 0x00000008
8 #define IMAGE_FROM_FIXED_MEDIA 0x00000010
9
10 //
11 // Authorization policy bit definition
12 //
```

```

13 | #define ALWAYS_EXECUTE 0x00000000 14 | #define NEVER_EXECUTE 0x00000001
15 | #define ALLOW_EXECUTE_ON_SECURITY_VIOLATION 0x00000002
16 | #define DEFER_EXECUTE_ON_SECURITY_VIOLATION 0x00000003
17 | #define DENY_EXECUTE_ON_SECURITY_VIOLATION 0x00000004
18 | #define QUERY_USER_ON_SECURITY_VIOLATION 0x00000005

```

收起 ^

By judging, if it is always executed or never executed, it returns directly, and then we look for the secureboot variable to determine whether it is on or off. If it is off, then no verification is performed directly, that is, normal startup. If it is on, we continue to the next step and obtain the DOS header

AI generated projects

登录复制

```

1 | //
2 | // Read the Dos header.
3 | //
4 | if (FileBuffer == NULL) {
5 |     return EFI_INVALID_PARAMETER;
6 | }
7 |
8 | mImageBase = (UINT8 *) FileBuffer;
9 | mImageSize = FileSize;
10 |
11 | ZeroMem (&ImageContext, sizeof (ImageContext));
12 | ImageContext.Handle = (VOID *) FileBuffer;
13 | ImageContext.ImageRead = (PE_COFF_LOADER_READ_FILE) DxeImageVerificationLibImageRead;

```

收起 ^

Determine whether it is a valid PE image:

AI generated projects

登录复制

```

1 | //
2 | // Get information about the image being loaded
3 | //
4 | Status = PeCoffLoaderGetImageInfo (&ImageContext);
5 | if (EFI_ERROR (Status)) {
6 |     //
7 |     // The information can't be got from the invalid PeImage
8 |     //
9 |     DEBUG ((DEBUG_INFO, "DxeImageVerificationLib: PeImage invalid. Cannot retrieve image information.\n"));
10 |     goto Done;
11 | }

```

收起 ^

If it is valid, continue to get the PE header, and then check the PE/COFF image---> Get the magic value from the PE/COFF Optional Header-->Use PE32 offset-->Use PE32+ offset-->Start Image Validation.

The previous part is skipped directly. The most important image verification starts from here. First, match the hash value of the image to see if it is verified. If it is not verified, check whether the hash value data exists in the DB database and does not exist in the DBX database. If it is verified, the verification passes. If it is verified and contains the signature verification information, traverse the certificates in the DB database to see if there are any that match. If there are any that match and are no longer in the DBX database, the verification passes. If none of them meet the requirements, it means that the image with the signature verification does not match. I will not post the specific code because there are a lot of codes and it is a bit obscure to analyze them sentence by sentence, but this is roughly what it means. Let's talk about the SecureBootConfigDxe configuration file.

SecureBootConfigDxe

There will be UNI, VFR files, etc., that is, an option will be generated in the BIOS setup interface for users to choose. When the safe mode is started, there will be two modes, factory mode and user mode. I will not introduce the endpoint and other things. There are also some variable-related settings. I will mainly talk about how to import the certificate to generate the DB database and DBX database.

For subsequent understanding:

```

1 | typedef struct{
2 |     EFI_GUID *VarGuid;
3 |     CHAR16 *VarName;
4 | } AUTHVAR_KEY_NAME;
5 |
6 | STATIC AUTHVAR_KEY_NAME gSecureKeyVariableList[] = {
7 |     {&gEfiGlobalVariableGuid, EFI_PLATFORM_KEY_NAME}, // PK 0
8 |     {&gEfiGlobalVariableGuid, EFI_KEY_EXCHANGE_KEY_NAME}, // KEK 1
9 |     {&gEfiImageSecurityDatabaseGuid, EFI_IMAGE_SECURITY_DATABASE}, // db 2
10 |    {&gEfiImageSecurityDatabaseGuid, EFI_IMAGE_SECURITY_DATABASE1}, // dbx 3

```

```
11 |};
```

收起 ^

AI generated projects

登录复制

```
1  STATIC
2  struct{
3      EFI_GUID      *File;
4      CHAR16        *UiName;
5      AUTHVAR_KEY_NAME *KeyName;
6      UINT8         *Data;
7      UINTN         DataSize;
8      EFI_GUID      *SignatureOwner;
9  }gSecureKeyTable[] = {
10     {(EFI_GUID*)PcdGetPtr(PcdSecureKeyDBFile),  L"DB",      &gSecureKeyVariableList[2], NULL, 0, &gMySignatureOwnerGuid},
11     {(EFI_GUID*)PcdGetPtr(PcdSecureKeyMSKEKFile), L"MSKEK", &gSecureKeyVariableList[1], NULL, 0, &gMicrosoftSignatureOwnerGuid},
12     {(EFI_GUID*)PcdGetPtr(PcdSecureKeyMSProFile), L"MSPro", &gSecureKeyVariableList[2], NULL, 0, &gMicrosoftSignatureOwnerGuid},
13     {(EFI_GUID*)PcdGetPtr(PcdSecureKeyMSUEFFile), L"MSUEF", &gSecureKeyVariableList[2], NULL, 0, &gMicrosoftSignatureOwnerGuid},
14     {(EFI_GUID*)PcdGetPtr(PcdSecureKeyPKFile),   L"PK",     &gSecureKeyVariableList[0], NULL, 0, &gMySignatureOwnerGuid},
15  };
```

收起 ^

Take adding DBXkey as an example:

AI generated projects

登录复制

```
1  STATIC
2  EFI_STATUS
3  AddDbxInitKey (
4      VOID
5  )
6  {
7      EFI_STATUS Status;
8
9      Status = AppendX509FromFV(
10         (EFI_GUID*)PcdGetPtr(PcdSecureKeyMSDBXFile),
11         gSecureKeyVariableList[3].VarName,
12         gSecureKeyVariableList[3].VarGuid,
13         &gMicrosoftSignatureOwnerGuid
14     );
15     DEBUG((EFI_D_INFO, "%a() %r\n", __FUNCTION__, Status));
16     return Status;
17 }
```

收起 ^

```
1  STATIC
2  EFI_STATUS
3  AppendX509FromFV(
4      IN EFI_GUID      *CertificateGuid,
5      IN CHAR16        *VariableName,
6      IN EFI_GUID      *VendorGuid,
7      IN EFI_GUID      *SignatureOwner
8  )
9  {
10     EFI_STATUS      Status;
11     VOID            *Data;
12     UINTN           DataSize;
13     UINTN           SigDBSize;
14     UINT32          Attr;
15     UINTN           X509DataSize;
16     VOID            *X509Data;
17
18     X509DataSize = 0;
19     X509Data     = NULL;
20     SigDBSize    = 0;
21     DataSize     = 0;
22     Data         = NULL;
23
24     Status = GetX509Cert( CertificateGuid, &X509Data,&X509DataSize);
25     if (EFI_ERROR (Status)) {
26         goto ON_EXIT;
27     }
```

```

28 |
29 |     SigDBSize = X509DataSize;
30 |
31 |     Data = AllocateZeroPool (SigDBSize);
32 |     if (Data == NULL) {
33 |         Status = EFI_OUT_OF_RESOURCES;
34 |         goto ON_EXIT;
35 |     }
36 |
37 |     CopyMem ((UINT8* )Data, X509Data, X509DataSize);
38 |
39 |     Attr = EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_RUNTIME_ACCESS
40 |           | EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS;
41 |
42 |     //
43 |     // Check if signature database entry has been already existed.
44 |     // If true, use EFI_VARIABLE_APPEND_WRITE attribute to append the
45 |     // new signature data to original variable
46 |     //
47 |
48 |     Status = gRT->GetVariable(
49 |         VariableName,
50 |         VendorGuid,
51 |         NULL,
52 |         &DataSize,
53 |         NULL
54 |     );
55 |
56 |     if (Status == EFI_BUFFER_TOO_SMALL) {
57 |         Attr |= EFI_VARIABLE_APPEND_WRITE;
58 |     } else if (Status != EFI_NOT_FOUND) {
59 |         goto ON_EXIT;
60 |     }
61 |
62 |     Status = gRT->SetVariable(
63 |         VariableName,
64 |         VendorGuid,
65 |         Attr,
66 |         SigDBSize,
67 |         Data
68 |     );

```

收起 ^

```

1 | STATIC
2 | EFI_STATUS
3 | GetX509Cert (
4 |     IN     EFI_GUID      *ImageGuid,
5 |     OUT    VOID          **DefaultsBuffer,
6 |     OUT    UINTN         *DefaultsBufferSize
7 | )
8 | {
9 |     EFI_STATUS      Status;
10 |    EFI_FIRMWARE_VOLUME2_PROTOCOL *Fv;
11 |    UINTN            FvProtocolCount;
12 |    EFI_HANDLE       *FvHandles;
13 |    UINTN            Index1;
14 |    UINT32           AuthenticationStatus;
15 |
16 |    *DefaultsBuffer = NULL;
17 |    *DefaultsBufferSize = 0;
18 |
19 |    FvHandles = NULL;
20 |    Status = gBS->LocateHandleBuffer (
21 |        ByProtocol,
22 |        &gEfiFirmwareVolume2ProtocolGuid,
23 |        NULL,
24 |        &FvProtocolCount,
25 |        &FvHandles
26 |    );
27 |
28 |    if (!EFI_ERROR (Status)) {
29 |        for (Index1 = 0; Index1 < FvProtocolCount; Index1++) {
30 |            Status = gBS->HandleProtocol (
31 |                FvHandles[Index1],
32 |                &gEfiFirmwareVolume2ProtocolGuid,
33 |                (VOID **) &Fv

```

```

34         ); 35         *DefaultsBufferSize= 0;
36
37         Status = Fv->ReadSection (
38             Fv,
39             ImageGuid,
40             EFI_SECTION_RAW,
41             0,
42             DefaultsBuffer,
43             DefaultsBufferSize,
44             &AuthenticationStatus
45         );

```

收起 ^

Finally, the acquired data is imported and processed;

```

1  STATIC
2  EFI_STATUS
3  AddSecureBootVarNoAuth(
4      CHAR16      *VarName,
5      EFI_GUID     *VarGuid,
6      CONST UINT8  *CertData,
7      UINTN        CertDataSize,
8      EFI_GUID     *SignatureOwner
9  )
10 {
11     EFI_STATUS      Status;
12     EFI_SIGNATURE_LIST *SignList;
13     UINTN           SignListSize;
14     UINT8           *VarData;
15     UINTN           VarDataSize;
16     UINT32          Attribute;
17     UINTN           DataSize;
18     EFI_SIGNATURE_DATA *SignData;
19
20
21     SignList = NULL;
22     VarData = NULL;
23
24     SignListSize = sizeof(EFI_SIGNATURE_LIST) + OFFSET_OF(EFI_SIGNATURE_DATA, SignatureData) + CertDataSize;
25     SignList = (EFI_SIGNATURE_LIST*)AllocatePool(SignListSize);
26     if(SignList == NULL){
27         Status = EFI_OUT_OF_RESOURCES;
28         goto ProcExit;
29     }
30
31     CopyMem(&SignList->SignatureType, &gEfiCertX509Guid, sizeof(EFI_GUID));
32     SignList->SignatureListSize = (UINT32)SignListSize;
33     SignList->SignatureHeaderSize = 0;
34     SignList->SignatureSize = (UINT32)(OFFSET_OF(EFI_SIGNATURE_DATA,SignatureData) + CertDataSize);
35     SignData = (EFI_SIGNATURE_DATA*)((UINT8*)SignList + sizeof(EFI_SIGNATURE_LIST));
36     CopyMem(&SignData->SignatureOwner, SignatureOwner, sizeof(EFI_GUID));
37     CopyMem(&SignData->SignatureData[0], CertData, CertDataSize);
38
39     VarData = (UINT8*)SignList;
40     VarDataSize = SignListSize;
41     Status = CreateDummyTimeBasedPayload(&VarDataSize, &VarData);
42     if(EFI_ERROR(Status)){
43         if((UINTN)VarData == (UINTN)SignList){
44             VarData = NULL;
45         }
46         goto ProcExit;
47     }
48
49     Attribute = EFI_VARIABLE_NON_VOLATILE |
50                 EFI_VARIABLE_BOOTSERVICE_ACCESS |
51                 EFI_VARIABLE_RUNTIME_ACCESS |
52                 EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS;
53
54     if(StrCmp(VarName, EFI_PLATFORM_KEY_NAME)!=0){
55         DataSize = 0;
56         Status = gRT->GetVariable(
57             VarName,
58             VarGuid,
59             NULL,
60             &DataSize,

```

```

61         NULL62 |
62     );
63     if(Status == EFI_BUFFER_TOO_SMALL){
64         Attribute |= EFI_VARIABLE_APPEND_WRITE;
65     }
66 }
67
68 Status = gRT->SetVariable (
69     VarName,
70     VarGuid,
71     Attribute,
72     VarDataSize,
73     VarData
74 );

```

收起 ^

The options in the configuration are the same as normal options. The selected function determines the operation. There are also some contents that need to be carefully considered, but the general principle is as mentioned above. Of course, the description is not so comprehensive, but it is just a general understanding of safe boot.