[UEFI Practice] OpenSSL in BIOS

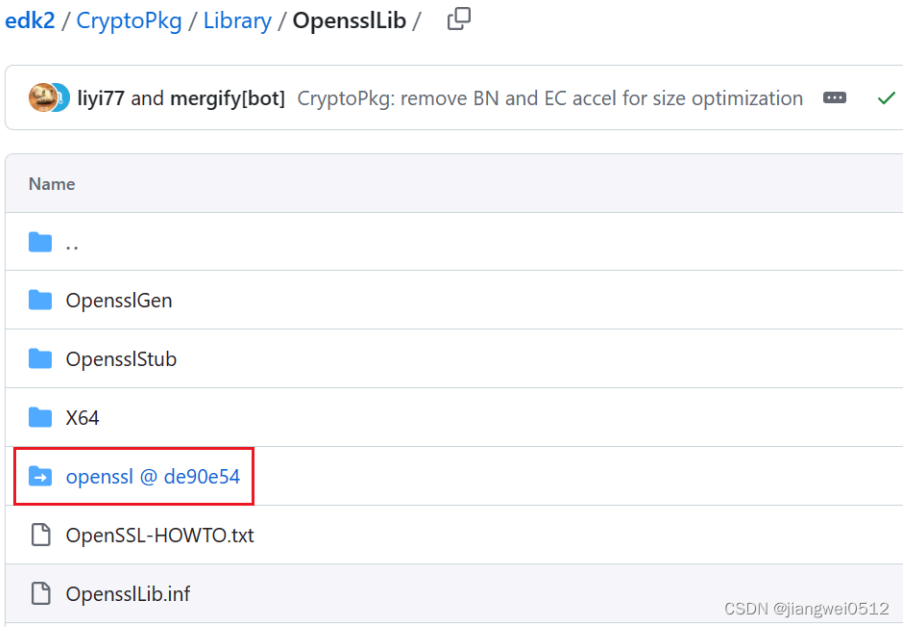
原创 jiangwei0512 Modified on 2024-03-02 16:42:37 Read 1.3k Collection 17 Likes 11

Category Column: UEFI Development Basics Article Tags: uefi openssl

openssl in BIOS

OpenSSL is a cryptographic library or cryptographic tool. The blog Cryptography Basics_hex string is too short, padding with zero bytes t-CSDN introduced the basic cryptographic concepts and the use of OpenSSL tools. Here we will introduce how to use OpenSSL under BIOS.

The open source BIOS code library EDK contains a CryptoPkg, which contains the cryptographic library interface required by BIOS. By including the openssl cryptographic library code, various cryptographic algorithms and tools can be used under BIOS. It should be noted that the EDK code does not directly contain the openssl code, but is implemented through external links, as shown in the following figure:



Clicking the red box will jump to the corresponding code base, where the number after @ is the corresponding version. This is because both EDK and openssl codes are constantly being updated, so there are compatibility issues. By specifying version information, it can be ensured that EDK can compile and use openssl normally.

In this way, when we download the EDK code directly, it will not contain the openssl code, and an additional download is required, which can be downloaded through the submodule subcommand of git. The test code https://gitee.com/jiangwei0512/edk2-beni.git used later has provided a one-click compilation method, and the corresponding openssl version will be downloaded during the first compilation.

Code Processing

After downloading the openssl code, EDK uses existing libraries `OpensslLib` to include the code and compile it. There are multiple such libraries, and the difference lies in the openssl functions included. If you need to support the TLS function in HTTPS, include it if necessary `OpensslLib.inf` , otherwise `OpensslLibCrypto.inf` just include it:


```
!if $(NETWORK_TLS_ENABLE) == TRUE
  OpensslLib|CryptoPkg/Library/OpensslLib/OpensslLib.inf
!else
  OpensslLib|CryptoPkg/Library/OpensslLib/OpensslLibCrypto.inf
!endif
```

Others include `OpensslLibAccel.inf` , `OpensslLibFull.inf` , `OpensslLibFullAccel.inf` etc., which have no essential differences, except for the number of functions they include.

It should be noted that this `OpensslLib` library is not directly used by other EDK codes. There is also a layer of EDK general library wrapped in the middle. These libraries correspond to different stages or functions of BIOS:


```
# SEC
[Components]
  #
  # SEC Phase modules
  #
  OvmfPkg/Sec/SecMain.inf {
    <LibraryClasses>
      NULL|MdeModulePkg/Library/LzmaCustomDecompressLib/LzmaCustomDecompressLib.inf
      NULL|OvmfPkg/IntelTdx/TdxHelperLib/SecTdxHelperLib.inf
      BaseCryptLib|CryptoPkg/Library/BaseCryptLib/SecCryptLib.inf
  }

# 通用
[LibraryClasses.common]
  BaseCryptLib|CryptoPkg/Library/BaseCryptLib/BaseCryptLib.inf

# Runtime
[LibraryClasses.common.DXE_RUNTIME_DRIVER]

```

```
  BaseCryptLib|CryptoPkg/Library/BaseCryptLib/RuntimeCryptLib.inf

# SMM
[LibraryClasses.common.DXE_SMM_DRIVER]
  BaseCryptLib|CryptoPkg/Library/BaseCryptLib/SmmCryptLib.inf

# TLS
!if $(NETWORK_TLS_ENABLE) == TRUE
  TlsLib|CryptoPkg/Library/TlsLib/TlsLib.inf
!endif
```

收起 ^

Therefore, when you need to use openssl under BIOS, you need to include libraries such as BaseCryptLib and introduce `BaseCryptLib.h` header files in the code.

In addition, there are several points to note.

First, in order for the open source openssl to be compiled normally in the EDK source code, some basic functions of its underlying layer need to be included, such as the very commonly used `memset()` functions in the openssl code. If it is compiled under Linux, the C standard library can be directly included, but there is no C standard library in the EDK source code. Therefore, in order to support this command, these basic functions need to be packaged, which introduces another C RunTime Library (CrtLib) under EDK, which can be seen from edk2\CryptoPkg\Library\Include\CrtLibSupport.h:

c AI generated projects 登录复制 run

```c
#ifndef __CRT_LIB_SUPPORT_H__
#define __CRT_LIB_SUPPORT_H__

#include <Library/BaseLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/DebugLib.h>
#include <Library/PrintLib.h>

#define OPENSSLDIR  ""
#define ENGINESDIR  ""
#define MODULESDIR  ""

#define MAX_STRING_SIZE  0x1000

// 中间略

//
// Macros that directly map functions to BaseLib, BaseMemoryLib, and DebugLib functions
//
#define memcpy(dest, source, count)         CopyMem(dest,source,(UINTN)(count))
#define memset(dest, ch, count)             SetMem(dest,(UINTN)(count),(UINT8)(ch))
#define memchr(buf, ch, count)              ScanMem8(buf,(UINTN)(count),(UINT8)ch)
#define memcmp(buf1, buf2, count)           (int)(CompareMem(buf1,buf2,(UINTN)(count)))
#define memmove(dest, source, count)        CopyMem(dest,source,(UINTN)(count))
#define strlen(str)                         (size_t)(AsciiStrnLenS(str,MAX_STRING_SIZE))
#define strncpy(strDest, strSource, count)  AsciiStrnCpyS(strDest,MAX_STRING_SIZE,strSource,(UINTN)count)
#define strcat(strDest, strSource)          AsciiStrCatS(strDest,MAX_STRING_SIZE,strSource)
#define strncmp(string1, string2, count)    (int)(AsciiStrnCmp(string1,string2,(UINTN)(count)))
#define strcasecmp(str1, str2)              (int)AsciiStriCmp(str1,str2)
#define strstr(s1, s2)                      AsciiStrStr(s1,s2)
#define sprintf(buf, ...)                   AsciiSPrint(buf,MAX_STRING_SIZE,__VA_ARGS__)
#define localtime(timer)                    NULL
#define assert(expression)
#define offsetof(type, member)  OFFSET_OF(type,member)
#define atoi(nptr)              AsciiStrDecimalToUintn(nptr)
#define gettimeofday(tvp, tz)   do { (tvp)->tv_sec = time(NULL); (tvp)->tv_usec = 0; } while (0)

#endif
```

收起 ^

The functions in the C standard library can be implemented through other libraries under EDK (such as the IntrinsicLib library). The internal implementation of the C standard library header functions such as stdlib.h included in openssl is to include the header file CrtLibSupport.h:

c AI generated projects 登录复制 run

```c
/** @file
  Include file to support building the third-party cryptographic library.

Copyright (c) 2010 - 2017, Intel Corporation. All rights reserved.<BR>
SPDX-License-Identifier: BSD-2-Clause-Patent

**/

#include <CrtLibSupport.h>
```

This creates the following pattern:

bash AI generated projects 登录复制

```bash
+================================+
| EDK II Firmware Module/Library |
+================================+
    ^                ^
    |                |
    v                v
+========+  +====================+
| TlsLib |  |    BaseCryptLib    |
+========+  +====================+
    ^                ^
    |                |
    v                v
+================================+
|     OpensslLib (Private)       |
+================================+
               ^
```

```
                |
                v
+================================+
|     IntrinsicLib (Private)     |
+================================+
```

收起 ^

Secondly, EDK includes OpenSSL in the form of a library, but there are different ways to implement this library, which can be a real library or include Protocol or PPI in the library. The former should be a little faster, while the latter can reduce the space occupied by OpenSSL included in EDK.

bash AI generated projects 登录复制

```
+==================+    +==================+    +==================+
|    EDK II PEI    |    |  EDK II DXE/UEFI |    |    EDK II SMM    |
|   Module/Library |    |   Module/Library |    |   Module/Library |
+==================+    +==================+    +==================+
  ^            ^          ^            ^          ^            ^
  |            |          |            |          |            |
  v            v          v            v          v            v
+==================+    +==================+    +==================+
|TlsLib|BaseCryptLib|    |TlsLib|BaseCryptLib|    |TlsLib|BaseCryptLib|
+------------------+    +------------------+    +------------------+
|   BaseCryptLib   |    |   BaseCryptLib   |    |   BaseCryptLib   |
|   OnPpiProtocol/ |    |   OnPpiProtocol/ |    |   OnPpiProtocol/ |
|  PeiCryptLib.inf |    |   DxeCryptLib.inf |    |  SmmCryptLib.inf |
+==================+    +==================+    +==================+
          ^                    ^                    ^
         ||| (Dynamic)         ||| (Dynamic)         ||| (Dynamic)
          v                    v                    v
+==================+    +==================+    +====================+
|    Crypto PPI    |    |  Crypto Protocol |    | Crypto SMM Protocol |
+------------------|    |------------------|    |--------------------|
|    CryptoPei     |    |    CryptoDxe     |    |     CryptoSmm      |
+==================+    +==================+    +====================+
     ^      ^              ^      ^              ^      ^
     |      |              |      |              |      |
     v      |              v      |              v      |
+========+  |          +========+  |          +========+  |
| TlsLib |  |          | TlsLib |  |          | TlsLib |  |
+========+  v          +========+  v          +========+  v
  ^  +==============+    ^  +==============+    ^  +==============+
  |  | BaseCryptLib |    |  | BaseCryptLib |    |  | BaseCryptLib |
  |  +==============+    |  +==============+    |  +==============+
  |          ^           |          ^           |          ^
  |          |           |          |           |          |
  v          v           v          v           v          v
+==================+    +==================+    +==================+
|    OpensslLib    |    |    OpensslLib    |    |    OpensslLib    |
+==================+    +==================+    +==================+
          ^                    ^                    ^
          |                    |                    |
          v                    v                    v
+==================+    +==================+    +==================+
|    IntrinsicLib  |    |    IntrinsicLib  |    |    IntrinsicLib  |
+==================+    +==================+    +==================+
```

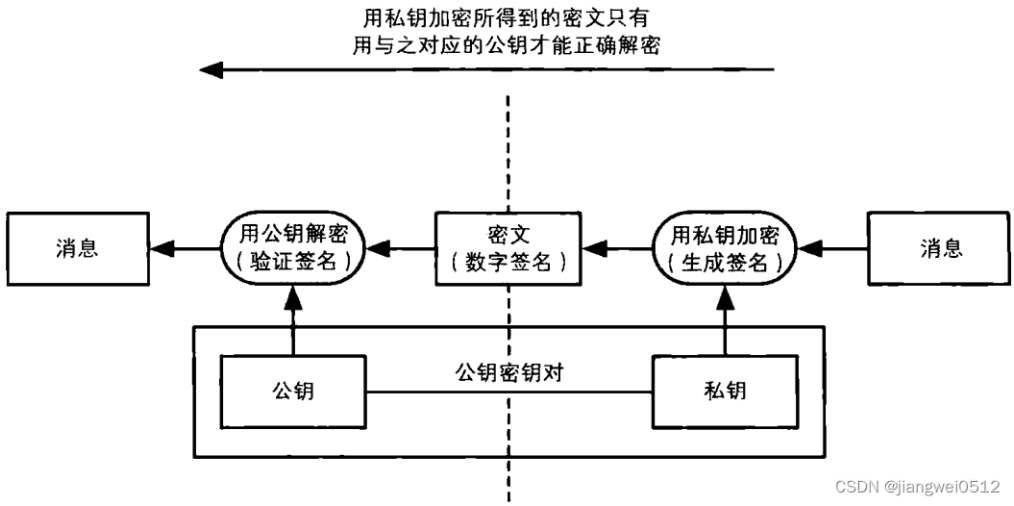
收起 ^

Depending on the size of the BIOS binary being used, different methods can be chosen.

Code Sample

During the BIOS startup process, the BootLoader will be executed to load the system. The most commonly used one is GRUB, which may be a UEFI application named bootx64.efi. At the end of the BIOS startup, the control will be handed over to this bootx64.efi, and the latter will start the system. But there is a problem here, how to ensure that this application is really what we need? If the application is modified or even replaced, resulting in the execution of some code that we do not want to execute, it is a very serious security vulnerability.

After reading the blog Cryptography Basics_hex string is too short, padding with zero bytes t-CSDN, you can know that you can use digital signatures to solve the problem of bootx64.efi being modified. The principle is shown in the figure below:



The actual process used is as follows:

1. Use the private key to encrypt bootx64.efi. The private key is retained by the provider of bootx64.efi and cannot be disclosed.
2. Put the public key in the BIOS code, and use this public key to verify bootx64.efi during startup. If successful, load this program, otherwise do not load it.

Through this operation, if bootx64.efi is modified, this program will not be executed, thus preventing the execution of abnormal code. The following will introduce the contents of these two parts.

Private key encryption

The first step is to create a private key, which can be done through the openssl tool.

1. Create a private key:

```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>openssl.exe genrsa -out private.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................................................................+++++
.....+++++
e is 65537 (0x010001)
```

2. Extract the public key from the private key:

```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>openssl.exe rsa -in private.pem -pubout -out public.pem
writing RSA key
```

3. Encrypt the SHA256 hash value of bootx64.efi (this file is not available at hand, use helloworld.efi instead):

```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>openssl.exe dgst -sign private.pem -sha256 -out sign.bin helloworld.efi
```

private.pem is the private key created earlier, helloworld.efi is the signed file (used to replace bootx64.efi for testing), and sign.bin is the output file, which represents the digital signature of helloworld.efi.

4. If you want to verify the signature, you can use the following command:

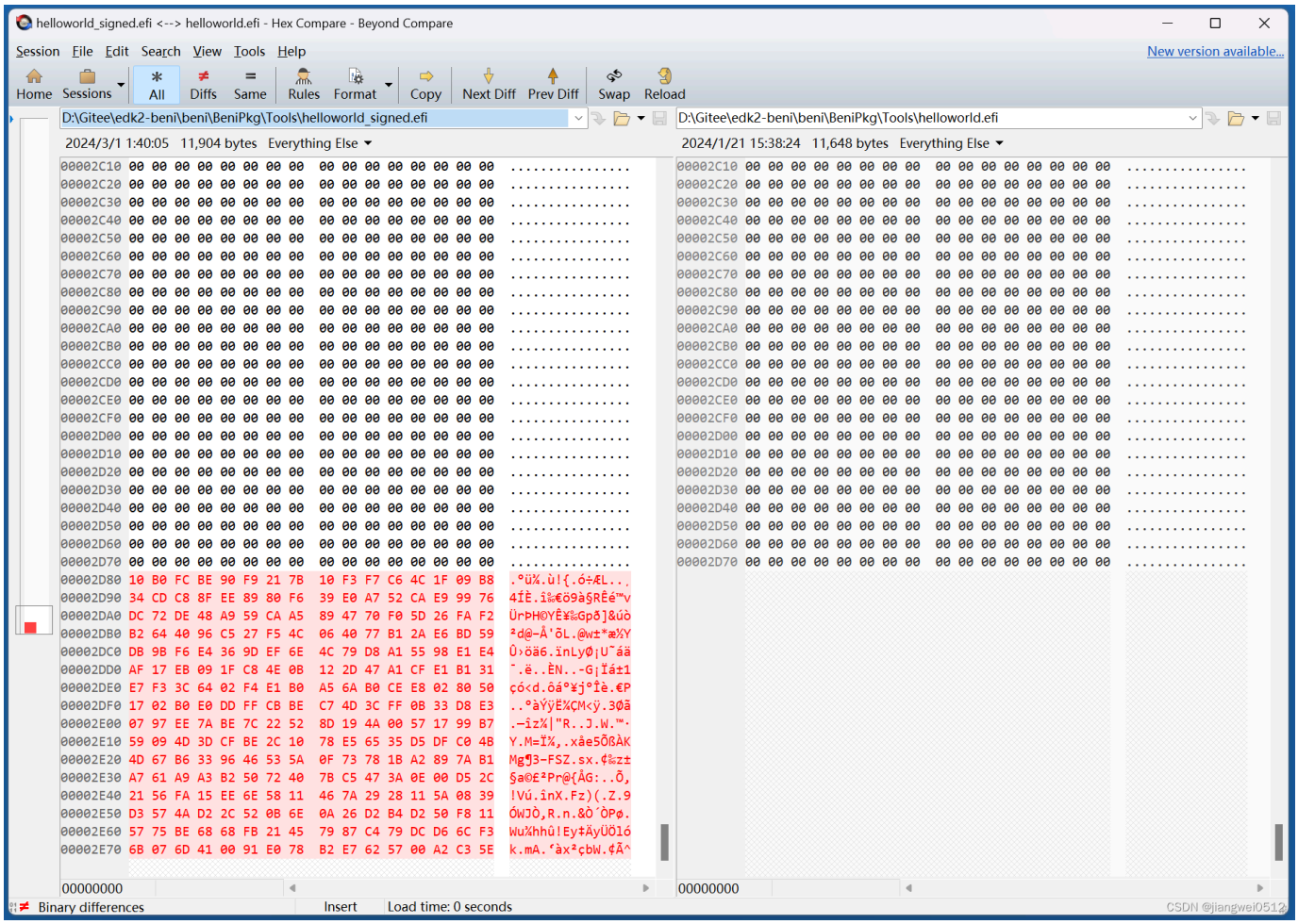
```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>openssl.exe dgst -verify public.pem -sha256 -signature sign.bin helloworld.efi
Verified OK
```

You can see "`Verified OK`", indicating that the verification is successful. Here public.pem is the corresponding public key. Our subsequent code is to implement this step.

5. Finally, put the original file (here is helloworld.efi) and the digital signature together to get the final efi file:

```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>copy /b helloworld.efi+sign.bin helloworld_signed.bin
helloworld.efi
sign.bin
已复制         1 个文件。
```

Finally, we get a helloworld_signed.efi, which is compared with the original version. The differences are as follows:



Public key decryption

This part requires BIOS code to implement, and a command called exec is used here for testing.

First, let's look at the original code, which is located in beni/BeniPkg/DynamicCommand/ExecuteShellAppCommand/Exec.c · jiangwei/edk2-beni - Code Cloud - Open Source China (gitee.com) , and its main implementation is:

c — AI generated projects — 登录复制 — run

```c
VOID
Exec (
  IN  CONST CHAR16                  *AppName
  )
{
  EFI_STATUS                Status = EFI_ABORTED;
  EFI_DEVICE_PATH_PROTOCOL  *DevPath = NULL;
  CHAR16                    *Str = NULL;

  DevPath = gEfiShellProtocol->GetDevicePathFromFilePath (AppName);
  if (NULL == DevPath) {
    DEBUG ((EFI_D_ERROR, "Device path not found!\n"));
    return;
  } else {
    Str = ConvertDevicePathToText (DevPath, TRUE, FALSE);
    if (Str) {
      DEBUG ((EFI_D_ERROR, "DevPath: %s\n", Str));
      FreePool (Str);
    }
  }
twen
twen  Status = gEfiShellProtocol->Execute (&gImageHandle, (CHAR16 *)AppName, NULL, NULL);
twen  if (EFI_ERROR (Status)) {
twen    DEBUG ((EFI_D_ERROR, "Execute failed. - %r\n", Status));
  }

  return;
}
```

收起 ^

This is a command under Shell, which is executed by the following command:

bash — AI generated projects — 登录复制

```bash
exec helloworld.efi
```

helloworld.efi is a shell application (unencrypted), and this operation will execute this command.

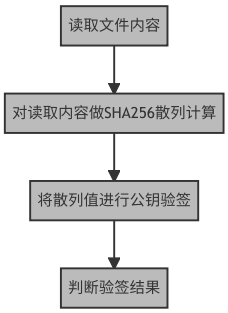
But when the security element is added, there is helloworld_signed.efi, so the code also needs to add relevant processing and `Exec()` add judgment in the function:

```c
VOID
Exec (
  IN  CONST CHAR16                  *AppName
  )
{
  EFI_STATUS                Status = EFI_ABORTED;
  EFI_DEVICE_PATH_PROTOCOL  *DevPath = NULL;
  CHAR16                    *Str = NULL;

  if (!(SecureCheck (AppName))) {
    ShellPrintHiiEx (-1, -1, NULL, STRING_TOKEN (STR_SECURE_ERROR), mExecHiiHandle);
    return;
  } else {
    ShellPrintHiiEx (-1, -1, NULL, STRING_TOKEN (STR_SECURE_SUCCESS), mExecHiiHandle);
  }
```

收起 ^

`SecureCheck()` OpenSSL is needed, and its operation process is as follows:



The following are the steps described above:

1. There is no need to explain how to read files, it can be done by calling the general interface.
2. SHA256 is used to hash the read content, because SHA256 is used in private key encryption , and the two must correspond. The code is as follows (for easy viewing, only the main code is retained):

```c
  HashContext = AllocateZeroPool (Sha256GetContextSize ());

  CryptoStatus = Sha256Init (HashContext);

  CryptoStatus = Sha256Update (HashContext, FileBuffer, (FileSize - RSA_LEN));

  CryptoStatus = Sha256Final (HashContext, Digest);
```

It should be noted here that the current application contains **two parts: original content** and **digital signature** , while hashing is only for the former part.

3. Next is signature verification. Here you need to use the public key, which has been generated in the introduction of private key encryption , but needs to be converted before it can be used in the code:

```bash
D:\Gitee\edk2-beni\beni\BeniPkg\Tools>openssl.exe rsa -in private.pem -text
RSA Private-Key: (2048 bit, 2 primes)
modulus:
    00:91:ad:ec:3f:4d:85:5f:c0:a7:95:14:92:6c:2f:
    0d:37:6e:58:2d:9a:06:0b:07:c0:15:90:1e:d9:70:
    25:a5:fe:87:68:c3:cd:a2:e5:d4:d7:3c:06:1f:30:
    a3:81:a7:6a:f0:27:aa:26:0c:cb:7d:cb:c2:2c:c6:
    67:b5:76:ef:30:4d:8d:12:6b:4d:20:11:2c:c4:69:
    a6:9b:db:0e:c8:ae:3e:cc:a8:e3:83:b9:80:5b:d2:
    97:3c:e2:e7:85:5a:db:53:23:8a:b4:a0:f8:02:f3:
    03:ec:41:37:97:d0:b5:35:f5:01:d9:3b:e8:24:24:
    ef:39:80:40:5e:c0:c6:b5:3d:32:3b:f1:4b:80:a9:
    2d:93:06:d4:8e:06:b6:b0:3e:ce:6a:17:75:28:32:
    50:a4:c1:86:4c:c0:46:bb:8d:83:6c:8e:53:96:72:
    7d:99:85:6f:19:b5:0c:33:1e:00:57:19:15:59:6b:
    58:30:dc:c5:00:0d:7c:cc:37:05:00:4f:17:a7:41:
    05:e0:d2:f7:67:67:f8:ce:77:a3:1b:9a:45:cf:04:
    14:04:9a:df:58:9d:2a:99:00:f7:16:94:ad:90:77:
    86:ff:6e:6b:03:d3:80:f3:f6:de:d9:cc:89:cc:bc:
    3b:f9:42:06:5d:ba:9b:93:96:b6:f3:e0:fd:98:a1:
twen    ff:9b
twen publicExponent: 65537 (0x10001)
```

收起 ^

Here, **modulus** and **publicExponent** are the values that need to be used in the code, and finally in the code:

```c
///
/// Public modulus of RSA Key.
///
CONST UINT8 mPublicKey[] = {
  0x91, 0xad, 0xec, 0x3f, 0x4d, 0x85, 0x5f, 0xc0, 0xa7, 0x95, 0x14, 0x92, 0x6c, 0x2f, 0x0d, 0x37,
  0x6e, 0x58, 0x2d, 0x9a, 0x06, 0x0b, 0x07, 0xc0, 0x15, 0x90, 0x1e, 0xd9, 0x70, 0x25, 0xa5, 0xfe,
  0x87, 0x68, 0xc3, 0xcd, 0xa2, 0xe5, 0xd4, 0xd7, 0x3c, 0x06, 0x1f, 0x30, 0xa3, 0x81, 0xa7, 0x6a,
  0xf0, 0x27, 0xaa, 0x26, 0x0c, 0xcb, 0x7d, 0xcb, 0xc2, 0x2c, 0xc6, 0x67, 0xb5, 0x76, 0xef, 0x30,
  0x4d, 0x8d, 0x12, 0x6b, 0x4d, 0x20, 0x11, 0x2c, 0xc4, 0x69, 0xa6, 0x9b, 0xdb, 0x0e, 0xc8, 0xae,
  0x3e, 0xcc, 0xa8, 0xe3, 0x83, 0xb9, 0x80, 0x5b, 0xd2, 0x97, 0x3c, 0xe2, 0xe7, 0x85, 0x5a, 0xdb,
```

```
  0x53, 0x23, 0x8a, 0xb4, 0xa0, 0xf8, 0x02, 0xf3, 0x03, 0xec, 0x41, 0x37, 0x97, 0xd0, 0xb5, 0x35,
  0xf5, 0x01, 0xd9, 0x3b, 0xe8, 0x24, 0x24, 0xef, 0x39, 0x80, 0x40, 0x5e, 0xc0, 0xc6, 0xb5, 0x3d,
  0x32, 0x3b, 0xf1, 0x4b, 0x80, 0xa9, 0x2d, 0x93, 0x06, 0xd4, 0x8e, 0x06, 0xb6, 0xb0, 0x3e, 0xce,
  0x6a, 0x17, 0x75, 0x28, 0x32, 0x50, 0xa4, 0xc1, 0x86, 0x4c, 0xc0, 0x46, 0xbb, 0x8d, 0x83, 0x6c,
  0x8e, 0x53, 0x96, 0x72, 0x7d, 0x99, 0x85, 0x6f, 0x19, 0xb5, 0x0c, 0x33, 0x1e, 0x00, 0x57, 0x19,
  0x15, 0x59, 0x6b, 0x58, 0x30, 0xdc, 0xc5, 0x00, 0x0d, 0x7c, 0xcc, 0x37, 0x05, 0x00, 0x4f, 0x17,
  0xa7, 0x41, 0x05, 0xe0, 0xd2, 0xf7, 0x67, 0x67, 0xf8, 0xce, 0x77, 0xa3, 0x1b, 0x9a, 0x45, 0xcf,
  0x04, 0x14, 0x04, 0x9a, 0xdf, 0x58, 0x9d, 0x2a, 0x99, 0x00, 0xf7, 0x16, 0x94, 0xad, 0x90, 0x77,
  0x86, 0xff, 0x6e, 0x6b, 0x03, 0xd3, 0x80, 0xf3, 0xf6, 0xde, 0xd9, 0xcc, 0x89, 0xcc, 0xbc, 0x3b,
  0xf9, 0x42, 0x06, 0x5d, 0xba, 0x9b, 0x93, 0x96, 0xb6, 0xf3, 0xe0, 0xfd, 0x98, 0xa1, 0xff, 0x9b,
};

///
/// Public exponent of RSA Key.
///
CONST UINT8 mRsaE[] = {
  0x01, 0x00, 0x01
  };
```

收起 ^

Once you have the public key, you can verify the signature, and the code processing is relatively simple:

c — AI generated projects — 登录复制 — run

```
![exec_helloworld_signed](BIOS.assets/exec_helloworld_signed.png)  Rsa = RsaNew ();

  CryptoStatus = RsaSetKey (Rsa, RsaKeyN, mPublicKey, sizeof (mPublicKey));

  CryptoStatus = RsaSetKey (Rsa, RsaKeyE, mRsaE, sizeof (mRsaE));

  CryptoStatus = RsaPkcs1Verify (
                  Rsa,
                  Digest,
                  SHA256_DIGEST_SIZE,
                  FileBuffer + (FileSize - RSA_LEN),
                  RSA_LEN
                  );
```

收起 ^

`RsaPkcs1Verify()` Accepted parameters:

- `Rsa` : RSA context.
- `Digest` , `SHA256_DIGEST_SIZE` : SHA256 hash value and its size.
- `FileBuffer + (FileSize - RSA_LEN)` , `RSA_LEN` : Digital signature and its size. Since a 2046-bit signature is used, this value is 256.

`FileBuffer + (FileSize - RSA_LEN)` This corresponds to the last 256 bytes of the application.

Test Results

The above binaries and codes are already included in edk2-beni: for learning and verifying UEFI BIOS. (gitee.com) , helloworld_signed.efi is included in the code, and it will be placed in fs0: after entering the Shell. The program can be executed successfully, but when any byte in helloworld_signed.efi is modified, an error will be reported: