About UEFI and Secure Boot

Written 29th December 2017

Sébastien Millet

milletseb@laposte.net

Part 1: UEFI

1. UEFI documentation

- You can find UEFI specifications here:

http://www.uefi.org/specifications

As of this writing, the current version is 2.7.

- The following page gives a thorough description of UEFI, in particular, its history, limits and conflicts that UEFI adoption triggered:

http://www.popflock.com/learn?s=UEFI

- Ubuntu page about UEFI:

https://help.ubuntu.com/community/UEFIBooting

2. Displaying UEFI boot configuration

UEFI comes with a NVRAM (Non Volatile RAM) to store its configuration.

This boot configuration can be tuned from two places:

1. Entering UEFI setup.

UEFI setup was formerly called BIOS configuration or BIOS setup, but note UEFI is totally different from the BIOS.

As this UEFI setup depends on the hardware, there is not much common information applicable to all cases that can be written here.

2. Using OS-level administration tools.

    - In Linux, running the command

      `efibootmgr -v`

      outputs detailed boot configuration, as found in the computer's NVRAM.

    - In Windows, running the command

`bcdedit /enum firmware`

outputs detailed boot configuration, as found in the computer's NVRAM.

3. How UEFI boots the computer

The way UEFI boots a computer is shown in depth here:

https://blog.uncooperative.org/blog/2014/02/06/the-efi-system-partition/

Quick abstract of the page above. Below, the term 'variable' refers to NVRAM-stored variables.

- The system examines the *BootOrder* variable. This variable contains a comma-separated list of 4-digit hexadecimal numbers. For each of these numbers, the system reads the variable *Boot####*, where #### is the 4-digit number of the BootOrder variable. This variable contains three things, a label (human-friendly readable), a path to a EFI-executable format file and optional data to pass to the file.

The "path to a file" contains data to locate the file not only on a file system but also across controllers, devices and partitions, therefore the whole string can be far from eye-friendly.

If the file is found and is found to be executable in the context (must be in EFI-executable format and possibly needs to pass SecureBoot checks, this point is discussed later), it is launched. Otherwise, the next variable (referred to in the *BootOrder* variable) is examined, and so on.

Example on my Acer swift 3 laptop with dual-boot between Kubuntu and Windows 10. The three-dot sequences (...) in the beginning or end of lines were added to see lines up to the end.

```
sebastien@maison-nblin:~/SecureBoot$ efibootmgr -v
BootCurrent: 0001
Timeout: 0 seconds
BootOrder: 0001,0005,0002,2001,2002,2003
Boot0000* Unknown Device:       HD(1,GPT,3f0f2672-851a-4ef7-82f2-61de04d2...
        ...3794,0x800,0x32000)/File(\EFI\ubuntu\shimx64.efi)RC
Boot0001* rEFInd        PciRoot(0x0)/Pci(0x1c,0x4)/Pci(0x0,0x0)/NVMe(0x1,...
        ...00-00-00-00-00-00-00-00)/HD(1,GPT,3f0f2672-851a-4ef7-82f2-61de...
        ...04d23794,0x800,0x32000)/File(\EFI\refind\refind_x64.efi)A01 ..
Boot0002* fwupx64       PciRoot(0x0)/Pci(0x1c,0x4)/Pci(0x0,0x0)/NVMe(0x1,...
        ...00-00-00-00-00-00-00-00)/HD(1,GPT,3f0f2672-851a-4ef7-82f2-61de...
        ...04d23794,0x800,0x32000)/File(\EFI\ubuntu\fwupx64.efi)A01 ..
Boot0003* Unknown Device:       HD(1,GPT,3f0f2672-851a-4ef7-82f2-61de04d2...
        ...3794,0x800,0x32000)/File(\EFI\ubuntu\shimx64.efi)RC
Boot0005* Windows Boot Manager  HD(1,GPT,3f0f2672-851a-4ef7-82f2-61de04d2...
```

```
        ...3794,0x800,0x32000)/File(\EFI\Microsoft\Boot\bootmgfw.efi)WIND...
        ...OWS.........x...B.C.D.O.B.J.E.C.T.=.{.9.d.e.a.8.6.2.c.-.5.c.d....
        ...d.-.4.e.7.0.-.a.c.c.1.-.f.3.2.b.3.4.4.d.4.7.9.5.}...............
        ..........
Boot2001* EFI USB Device        RC
Boot2002* EFI DVD/CDROM RC
Boot2003* EFI Network   RC
sebastien@maison-nblin:~/SecureBoot$
```

It is worth noting that not all variables *Boot####* are listed in *BootOrder*, meaning, some *Boot####* variables are ignored at start-up.

- The Default Boot Behavior

If the above fails (*Boot...* variables don't allow UEFI to find a good candidate to start the system), UEFI is said to look for an ESP (more to come on that) partition and if it finds one, it looks for a file *\EFI\BOOT\BOOT{arch}.EFI*, where *{arch}* depends on the machine architecture.

Examples for a few architectures:

```
x86-64: \EFI\BOOT\BOOTX64.EFI
x86-32: \EFI\BOOT\BOOTIA32.EFI
Itanium:\EFI\BOOT\BOOTIA64.EFI
ARM 32-bit: \EFI\BOOT\BOOTARM.EFI
ARM 64-bit: \EFI\BOOT\BOOTAA64.EFI
```

Source:

https://www.happyassassin.net/2014/01/25/uefi-boot-how-does-that-actually-work-then/

- On what volume(s) is the ESP being searched for?

It should be searched first on removable medias, then on fixed medias, but I got to check further if it is a recommendation or a rule. I presume it depends anyway on the configuration (whether or not you asked UEFI to start from a removable media).

- The ESP

ESP stands for *EFI System Partition*.

A partition is identified as being ESP by its *type* in the partition table. Although a disk can hold multiple partitions that are of type "ESP", it is strongly recommended a disk holds not more than one.

UEFI specifications call for a GPT partitioning system (UEFI uses the term *layout*). However, the specs document also says the following (extract of version 2.7 page 125):

MBR is explained later in this document, in paragraph 4.1.1.

> A legacy MBR may be located at LBA 0 (i.e., the first logical block) of the disk if it is not using the GPT disk layout (i.e., if it is using the MBR disk layout). The boot code on the MBR is not executed by UEFI firmware.

Most documentation ties UEFI with GPT, and it is true it is the natural layout of UEFI, but it ain't a rule. Indeed, page 126 of the UEFI specs document mentions the code 0xEF that must be taken into account when encountered on a MBR layout partition. This code corresponds to the type 'EFI (FAT-12/16/32)' as reported by fdisk on a MBR layout (note: fdisk does not speak about a *MBR layout*, instead, its terminology is *dos disklabel* versus *gpt disklabel*).

The UEFI standard requests the ESP to be formatted with FAT. In some specific cases the FAT size matters but it seems rather rare. The UEFI specs say a fixed disk must have FAT32 while FAT12 and FAT16 are also possible (in addition to FAT32) for removable disks.

The following site gives numerous useful details about UEFI and I recommend reading it entirely. About the FAT size topic, this page in particular gives some details:

https://www.rodsbooks.com/efi-bootloaders/principles.html

If you don't know what to do: go for FAT32.

The partition size is subject to several rules and recommendations, so there is no short instruction "choose size NNN and you'll be happy".

You want one simple instruction? OK, here we go: choose 200 MB. I said nothing.

4. BIOS versus UEFI

UEFI is 100% different from BIOS and you're going to have a hard time if thinking of UEFI as just a more recent BIOS version.

I wrote below some remarks of what to keep in mind when learning UEFI with BIOS habits.

4.1. Booting on a media

4.1.1. Reminder about BIOS

BIOS can start a PC from a disk only [1] by executing the code found on the MBR (*Master Boot Record*). The MBR is the first sector of the disk, it is 512 bytes in length.

[1] PXE (booting from network) actually will not execute code found on the MBR, but we are speaking here about starting from a storage media.

Given a media, either the BIOS can boot it (the media is said "bootable"), or, it cannot. That is to say, you can't ask the BIOS to start one *partition* on a media versus another, nor can you ask to start a given file.

For disks, the mark "is bootable" is a specific two-byte sequence written at the end of the MBR, so that the BIOS knows it is possible to load and execute it.

This way of working has some black magic inside: the Operating System to load (or the Boot Manager to load) is found on files inside a file system. The MBR is too small to contain all needed code to manage a file system. To resolve this chicken-and-egg situation, some *early* loaded code is made available out of any file system, that is, out of disk partitions, right after the MBR.

If you want more about this topic, in the case of GRUB (GRUB is a popular Boot Manager), this in-between code is called stage1.5 code. Stage1 code is MBR's while stage2 is regular GRUB software, loaded from files. See this URL that has a very interesting diagram about GRUB stages:

https://unix.stackexchange.com/questions/259143/how-does-grub-stage1-exactly-access-load-stage-2

This is fairly "hack-ish", and UEFI approach of a dedicated ESP (more to come about ESP) using a file system guaranteed to be understood by UEFI, is conceptually much cleaner. As explained below, UEFI implementations found in the real world seem to ignore the fact that a given partition is marked of type 'ESP', but that is another story.

- The troll note

This document aims to give details about UEFI. It is not in favor of UEFI, nor is it against UEFI. UEFI may have numerous pluses and minuses in its conception or in its implementations. Nevertheless for the author of this document, it is beyond doubt that UEFI is *conceptually* far better than BIOS.

4.1.2. UEFI

UEFI, via the *Boot...* NVRAM variables, can be instructed to boot from different disks, different volumes [2] on a disk, and on different files on a volume. UEFI can even pass parameters to the loaded file, giving a lot of flexibility to the boot process.

Still, UEFI needs a way to start "from scratch", in the case NVRAM is not yet configured or got wrongly updated, if something goes wrong, or most often, if you just want to boot from a removable media that is not configured whatsoever in your UEFI. Either way, this is what we called above the *Default Boot Behavior*.

[2] In this document, *volume* and *partition* are used interchangeably.

4.1.3. Table of BIOS versus UEFI criteria to see a media as bootable

	BIOS	UEFI
Criteria for being a bootable disk	Has a MBR marked as bootable	Has a partition readable by UEFI [3] that contains a file named *\EFI\BOOT\bootx64.efi* [4]

4.1.4. Tests done

Tests done on an Acer Swift 3.

- Ask the PC to display the boot device selection menu (F12 on my Acer, generally different on other manufacturers).
- If a plugged USB would contain a partition that matches both of the following criteria ...
  - Formatted with FAT (any size is fine, from 12 to 32)
  - Contains a file \EFI\BOOT\BOOTX64.EFI (my PC is an x64 architecture)
- ... then, UEFI would include "HDD on USB" in the boot device list. Let me insist: if *any* of the criteria above was not met, the "HDD on USB" entry would *not* be displayed. It is to be noted that during these tests, the partition type was never taken into account - I could mark a partition as being "Microsoft basic data" or even "Linux filesystem" (!), it would be fine. Tests done with a GPT partition layout and with a MBR layout.
- Conclusion: under no circumstance did I find a case where marking a partition as 'ESP' would influence UEFI behavior. Obviously, this result applies to the particular UEFI implementation tested, but it matches numerous feedbacks and documents on Internet reporting similar results. Keep in mind marking a partition as *ESP* is likely to have no influence on UEFI behavior. It is important, still, to use the *ESP* type. It makes things clearer, and I presume (but I haven't tested it), it'll also help OS installers.

[3] It is typically a FAT partition of type "EFI System", making the disk's ESP, the *EFI System Partition*. As highlighted by https://blog.uncooperative.org/blog/2014/02/06/the-efi-system-partition/ marking a partition as *ESP* is not mandatory. Having a partition marked ESP on a disk brings a lot of robustness and clarity, in particular when compared to BIOS craziness. But given the flexibility of UEFI in practice, the *ESP* type should be regarded as a mark to help humans more than machines. Well, humans and OS installers (no, don't ask me. I didn't check the behavior of Kubuntu when installing in UEFI mode on a disk that has no ESP).

[4] bootx64.efi is for the case of an x64 architecture. See list earlier in this document, for other architectures.

4.2. BIOS and UEFI together

Most UEFIs can manage old boot system, as if being BIOS. Note it is a generality, not a rule. The popflock.com URL given above mentions Intel announcement that by 2020, BIOS support would be removed from their products.

Conversely, being able to deal with an MBR layout is mandatory in UEFI specifications.

Switching UEFI to behaving as if being BIOS is called CSM, *Compatibility Support Module*, or *Legacy* mode, also sometimes simply *BIOS mode*. You normally change it manually in UEFI configuration. popflock.com reports that certain UEFI implementations turn on CSM automatically if they encounter an MBR layout.

Because MBR and GPT do not reside in the same areas of the disk, they can coexist peacefully. Before I study UEFI, my dual-boot computer would need *UEFI mode* to start Windows 10, and the *BIOS mode* (called *Legacy mode* on my laptop) to start Linux. But, switching from one OS to the other would need, to enter UEFI settings (did I say BIOS screen?) and switch mode between "UEFI" and "Legacy". Yes, believe me, life is sometimes difficult.

Because of this duality of the partition table, there are tools to synchronize between MBR layout and GPT layout partition tables. Under Ubuntu, one tool to do it is *gptsync*.

4.3 Turning a BIOS installed Linux system into UEFI, and vice versa

This URL gives a lot of useful instructions to switch, and more:

https://help.ubuntu.com/community/UEFI

Note

During all the tests I did, the original start mode was kept during any further action. That is, when starting Ubuntu installer in BIOS mode, target installation could start only in BIOS mode, and the same for UEFI. Once you start and install in a given mode, you keep it forever.

This is an important point and it also applies to *Boot-Repair*. Boot-Repair is recommended by the site above.

But, as it turns out, just copying Boot-Repair on a USB (for example with *dd*) won't make it UEFI-bootable, as it'll lack a FAT partition containing a proper .efi file. Read carefully Boot-Repair site instructions to manage a PC that starts in UEFI.

Conversely, Ubuntu installer ISO (checked on Ubuntu 17.10 desktop x64) has an included minimalistic FAT partition with proper .efi file, making it compatible with both legacy and UEFI modes. It is to be noted the partition layout on the

ISO is *MBR*. I was expecting *GPT* but *MBR* has a logic of its own: it must work on a recent machine (as per UEFI standard) but will also make older machines happy.

5. More reading

Links given below are just partial - they correspond to the topics I am personally interested in...

5.1. rEFInd

rEFInd is a *Boot Manager*, as opposed to a *Boot Loader*, although it can start an OS directly. This document contains other places that point to rEFInd website pages. rEFInd home page:

http://www.rodsbooks.com/refind

rEFInd has great qualities:

1. It can manage multiple OSes gracefully, with a nice interface.
2. It is easy to launch certain EFI utilities from rEFInd, like *KeyTool*. *KeyTool* allows to control *PK*, *KEK*, *db*, *dbx* and *MokList* SecureBoot UEFI variables.
3. When you learn how to install and use rEFInd, you happen to learn what UEFI is. Same with Secure Boot.

rEFInd is really *the site to read* when learning *UEFI* and when learning *UEFI Secure Boot*. It is very well documented.

5.2. GRUB

The following URL describes how to boot different ISO images with proper GRUB configuration. It is a way to increase one's skills in GRUB rather than UEFI itself, but still, it is interesting to see how you can play with GRUB in UEFI mode:

https://ubuntuforums.org/showthread.php?t=2276498

5.3. tianocore

tianocore is an open source implementation of UEFI.

http://www.tianocore.org/

5.4. EFI shell

One page of Rod Smith's site gives the place where to download EFI shell from.

The page is:

https://www.rodsbooks.com/efi-programming/hello.html

The links referred to by this page are:

- x86-64

https://edk2.svn.sourceforge.net/svnroot/edk2/trunk/edk2/EdkShellBinPkg/FullShell/X64/Shell_Full.efi

- x86

https://edk2.svn.sourceforge.net/svnroot/edk2/trunk/edk2/EdkShellBinPkg/FullShell/Ia32/Shell_Full.efi

Part 2: Secure Boot

Secure Boot is an UEFI *feature*. You can have UEFI without Secure Boot, but not Secure Boot without UEFI.

1. Secure Boot modes

A machine that has the Secure Boot feature can be in three modes:

1. Setup mode (it implies, Secure Boot is off)
2. User mode, Secure Boot off
3. User mode, Secure Boot on

- Note

It seems the so-called *Setup Mode* is sometimes referred to as *Custom Mode*, on some documents found on Internet, and on some UEFI firmwares. The *User Mode* versus *Setup Mode* is in itself, very poorly documented. So is the *Custom Mode*. The author of this document *does think Custom Mode* and *Setup Mode* are one unique thing, but even this point is not so clear.

Numerous documentations mention these two different states without a clear definition of what it means and how you move from one to another.

2. Secure Boot checks

Secure Boot will check if a binary is allowed before executing it. The check will be done only in mode 3, that is, *User mode, Secure Boot on*.

The check is one of:

- Verify if the file is authorized. It is managed with SHA256 hash lists.

or

- If the file is signed, verify if the signature allows to execute it. It is mostly managed with RSA keys of 2048 bits found in X509 certificates. The signature is part of the EFI file itself. That means, either an EFI file is signed, or, it is not (I haven't found mention of detached signature files in this standard).

3. How the checks are done

The machine' NVRAM contains Secure Boot related variables. They are:

- *db*: contains SHA256 hash of authorized files and certificates that, when signing a file [5], will allow its execution. That means, *db* contains two different types of entries, *authorized certificates* and *authorized hashes*.
- *dbx*: contains SHA256 hash of disallowed files and certificates that, when signing a file, will disallow its execution. In case a file is matched by both db and dbx, dbx wins and the file is not allowed to be executed. As for *db*, *dbx* contains two different types of entries, *forbidden certificates* and *forbidden hashes*.
- *KEK*: Key-Exchange Keys. When a change is to be made on *db* or *dbx*, it must be signed with a certificate listed in *KEK* variable. Also, signed code can be signed by a *KEK* certificate directly, instead of a *db* one.
- *PK*: Platform Key. As opposed to other variables, it is one unique entry, made of a certificate. When a change is to be made on *KEK*, it must be signed by this certificate.
- *MokList*: Machine-Owner Key. Actually this is not part of UEFI specifications and is not managed by UEFI. MOKs are managed by Linux standard loaders like *shim*. According to rodsbooks.com site (see URL below), MOKs are easier to install. Rod Smith mentions the Linux utility *efi-readvar* would not show MOKs of his machine, and it is the same for me.

4. About *Setup Mode* and *User Mode*

When in *Setup Mode*, you can update the *PK*, *KEK*, *db* and *dbx* variables to your convenience.

[5] I write that *certificates sign a file* to keep it simple. Strictly speaking, the certificate is associated to a private and public key pair (the certificate file contains the public part of the key pair), and a signature is done with the private key.

It is worth noting that entering UEFI *Setup Mode* is "poorly documented" to say the least. Very little Internet documentation is available about it.

On my Acer Swift 3, I have to enter UEFI configuration screen and reset Secure Boot. It sets variables to their factory initial values. There is no mention of the fact that it switches the machine to *Setup Mode*.

After doing it, the machine is in *Setup Mode*, until the *PK* variable is set. When *PK* is set, it immediately switches the mode from *Setup* to *User*, and no further update can be done. As said earlier, *db* updates (signed with *KEK* certificate) or *KEK* updates (signed with *PK*) could never be done.

***efi-updatevar* (*efitools* debian/ubuntu package) would always produce an error. Same result with *KeyTool*.**

In the end, all this system of *PK -> KEK -> (db, dbx)* update hierarchy is just useless. I lastly left *PK* and *KEK* to their default values, and added my own certificate to *db*, so that I can sign whichever binary I want to sign to make it executable while Secure Boot is on. While putting my own keys in *PK* and *KEK* would look like a satisfying situation for the ego, it would serve no purpose at all.

The below link is one of the few that gives details about *Setup Mode* and *User Mode*:

https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance

5. Updating Secure Boot variables

5.1. rosdbook site

The URL below explains how to customize your Secure Boot, that is, how to modify *PK*, *KEK*, *db* and *dbx* to your convenience. It gives the steps to follow (and it even provides scripts) to create proper certificates and shows how to create the intermediate files to update UEFI Secure Boot variables, the *.esl* and *.auth* files.

It also explains the update hierarchy: *PK* signature to update *KEK*, *KEK* signature to update *db* or *dbx*.

https://www.rodsbooks.com/efi-bootloaders/controlling-sb.html

In my case though, I could never update *db* or *dbx* using a *KEK*-signed update file (.auth), nor could I update *KEK* list with a *PK*-signed update file. In the end, I don't see the point of all this mechanism: whenever I would need to update anything in *db* or *dbx*, I got to enter UEFI *Setup Mode* and to rewrite completely the variables *PK*, *KEK* and *db*.

5.2. gentoo site

To customize Secure Boot keys, in addition to Rod Smith link, you may have a look at the page below. Commands are not 100% the same as on Rod Smith's site (a *-a* option shows up), besides, it is another point of view of the same actions.

https://wiki.gentoo.org/wiki/Sakaki%27s_EFI_Install_Guide/Configuring_Secure_Boot

6. Hashes

The *db* and *dbx* variables can contain both certificates and hashes. This allows to whitelist (*db*) or blacklist (*dbx*) directly a given file, without considering its signature.

Here comes the tricky point: the *EFI executable* hash does not correspond to the file hash, as one would expect. Maybe it is related to some code alignment matters? Anyway, if one wishes to confirm what a given file hash corresponds to, it is not possible "as is".

The below shows how to recompile *sbsign* executable, adding *–hash* option. *–hash* will just print out the EFI file hash (SHA256) and quit.

6.1. Get the source of *sbsigntool*

The source is available here: git://kernel.ubuntu.com/jk/sbsigntool.git

If using a packaged Linux distribution, you had better rely on the package manager. On Ubuntu, you got to execute the command below:

```
apt-get source sbsigntool
```

6.2. Apply the patch

Apply the patch below to sbsigntool' source.

File *sbsigntool-0.6-patch.txt*:

```
diff -Naur sbsigntool-0.6/src/sbsign.c sbsigntool-0.6-hash/src/sbsign.c
--- sbsigntool-0.6/src/sbsign.c 2017-12-16 10:18:36.000000000 +0100
+++ sbsigntool-0.6-hash/src/sbsign.c    2017-12-16 10:31:34.781884149 +0100
@@ -57,6 +57,8 @@

 static const char *toolname = "sbsign";
```

```
+static int only_print_hash = 0;
+
 struct sign_context {
    struct image *image;
    const char *infilename;
@@ -70,6 +72,7 @@
    { "cert", required_argument, NULL, 'c' },
    { "key", required_argument, NULL, 'k' },
    { "detached", no_argument, NULL, 'd' },
+   { "hash", no_argument, NULL, 'H' },
    { "verbose", no_argument, NULL, 'v' },
    { "help", no_argument, NULL, 'h' },
    { "version", no_argument, NULL, 'V' },
@@ -90,7 +93,9 @@
        "\t--output <file>    write signed data to <file>\n"
        "\t                    (default <efi-boot-image>.signed,\n"
        "\t                    or <efi-boot-image>.pk7 for detached\n"
-       "\t                    signatures)\n",
+       "\t                    signatures)\n"
+       "\t--hash             output the SHA256 sum that would be signed\n"
+        "\t                    and exit (don't sign)\n",
        toolname);
 }

@@ -123,7 +128,7 @@

    for (;;) {
        int idx;
-       c = getopt_long(argc, argv, "o:c:k:dvVh", options, &idx);
+       c = getopt_long(argc, argv, "o:c:k:dvVhH", options, &idx);
        if (c == -1)
            break;

@@ -140,6 +145,9 @@
        case 'd':
            ctx->detached = 1;
            break;
+       case 'H':
+           only_print_hash = 1;
+           break;
        case 'v':
            ctx->verbose = 1;
            break;
@@ -161,13 +169,13 @@
    if (!ctx->outfilename)
```

```
        set_default_outfilename(ctx);

-   if (!certfilename) {
+   if (!only_print_hash && !certfilename) {
        fprintf(stderr,
            "error: No certificate specified (with --cert)\n");
        usage();
        return EXIT_FAILURE;
    }
-   if (!keyfilename) {
+   if (!only_print_hash && !keyfilename) {
        fprintf(stderr,
            "error: No key specified (with --key)\n");
        usage();
@@ -184,6 +192,16 @@
    OpenSSL_add_all_digests();
    OpenSSL_add_all_ciphers();

+    if (only_print_hash) {
+        uint8_t sha[SHA256_DIGEST_LENGTH];
+        image_hash_sha256(ctx->image, sha);
+        for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
+            printf("%02x", sha[i]);
+        }
+        printf("\n");
+        return EXIT_SUCCESS;
+    }
+
    EVP_PKEY *pkey = fileio_read_pkey(keyfilename);
    if (!pkey)
        return EXIT_FAILURE;
```

---

cd to the source directory and apply the patch like this:

```
> patch -p 1 < sbsigntool-0.6-patch.txt
patching file src/sbsign.c
>
```

Then, generate binaries as usual. Only *sbsign* binary is different from original source.

Using the modified *sbsign* allows to match a given hash with a file, example:

```
sebastien@maison-nblin:~/sbsigntool$ efi-readvar -v db
Variable db, length 6924
db: List 0, type X509
    Signature 0, size 1515, owner 77fa9abd-0359-4d32-bd60-28f4e78f784b
```

```
        Subject:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Windows Pr
        Issuer:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Root Certi
db: List 1, type X509
    Signature 0, size 1572, owner 77fa9abd-0359-4d32-bd60-28f4e78f784b
        Subject:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Corporatio
        Issuer:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Corporatio
db: List 2, type X509
    Signature 0, size 923, owner 92fcafcd-c861-4b8b-aff2-a3d5a3e093f8
        Subject:
            C=Taiwan, ST=TW, L=Taipei, O=Acer, CN=Acer Database
        Issuer:
            C=Taiwan, ST=TW, L=Taipei, O=Acer, CN=Acer Root CA
db: List 3, type X509
    Signature 0, size 797, owner 92fcafcd-c861-4b8b-aff2-a3d5a3e093f8
        Subject:
            CN=PEGA-SW2
        Issuer:
            CN=PEGA-SW2
db: List 4, type X509
    Signature 0, size 1096, owner 685984e3-5d0f-4682-94c1-0f85ecb55d34
        Subject:
            C=GB, ST=Isle of Man, L=Douglas, O=Canonical Ltd., CN=Canonical Ltd. Master Cert
        Issuer:
            C=GB, ST=Isle of Man, L=Douglas, O=Canonical Ltd., CN=Canonical Ltd. Master Cert
db: List 5, type X509
    Signature 0, size 777, owner eee29bf3-fbc6-4c38-96df-bfe181770f39
        Subject:
            CN=SMT db
        Issuer:
            CN=SMT db
db: List 6, type SHA256
    Signature 0, size 48, owner 00000000-0000-0000-0000-000000000000
        Hash:a220f1b4305d04f59565e5c08ad35fcdda3540e4841d412e31d97ae780b6f66d
sebastien@maison-nblin:~/sbsigntool$ ./sbsign --hash /usr/lib/fwupdate/fwupx64.efi
warning: data remaining[57344 vs 65261]: gaps between PE/COFF sections?
warning: data remaining[57344 vs 65264]: gaps between PE/COFF sections?
a220f1b4305d04f59565e5c08ad35fcdda3540e4841d412e31d97ae780b6f66d
sebastien@maison-nblin:~/sbsigntool$
```

7. More reading

7.1. Microsoft

The following site (already mentioned above) gives interesting schemas about UEFI, and is rather precise around Secure Boot.

https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance

7.2. OS DEV

http://wiki.osdev.org/UEFI