UEFI boot: how does that actually work, then?

By adamw on January 25, 2014

It's AdamW Essay Time again! If you're looking for something short and snappy, look elsewhere.

Kamil Paral kindly informs me I'm a chronic sufferer of Graphomania. Always nice to know what's wrong with you.

IMPORTANT NOTE TO INDUSTRY FOLKS: This blog post is aimed at regular everyday folks; it's intended to dispel a few common myths and help regular people understand UEFI a bit better. It is not a low-level fully detailed and 100% technically accurate explanation, and I'm not a professional firmware engineer or anything like that. If you're actually building an operating system or hardware or something, please don't rely on my simplified explanations or ask me for help; I'm just an idiot on the internet. If you're doing that kind of thing and you have money, join the UEFI Forum or ask your suppliers or check your reference implementation or whatever. If you don't have money, try asking your peers with more experience, nicely. **END IMPORTANT NOTE**

You've probably read a lot of stuff on the internet about UEFI. Here is something important you should understand: 95% of it was probably garbage. If you think you know about UEFI, and you derived your knowledge anywhere other than the UEFI specifications, mjg59's blog or one of a few other vaguely reliable locations/people – Rod Smith, or Peter Jones, or Chris Murphy, or the documentation of the relatively few OSes whose developers actually know what the hell they're doing with UEFI – what you think you know is likely a toxic mix of misunderstandings, misconceptions, half-truths, propaganda and downright lies. So you should probably forget it all.

Good, now we've got that out of the way. What I mostly want to talk about is bootloading, because that's the bit of firmware that matters most to most people, and the bit news sites are always banging on about and wildly misunderstanding.

Terminology

First, let's get some terminology out of the way. Both BIOS and UEFI are types of firmware for computers. BIOS-style firmware is (mostly) only ever found on IBM PC compatible computers. UEFI is meant to be more generic, and can be found on systems which are not in the 'IBM PC compatible' class.

You do not have a 'UEFI BIOS'. No-one has a 'UEFI BIOS'. Please don't ever say 'UEFI BIOS'. BIOS is not a generic term for all PC firmware, it is a particular type of PC firmware. Your computer has a firmware. If it's an IBM PC compatible computer, it's almost certainly either a BIOS or a UEFI firmware. If you're running Coreboot, congratulations, Mr./Ms. Exception. You may be proud of yourself.

Secure Boot is not the same thing as UEFI. Do not ever use those terms interchangeably. Secure Boot is a single effectively optional element of the UEFI specification, which was added in version 2.2 of the UEFI specification. We will talk about precisely what it is later, but for now, just remember it is not the same thing about UEFI. You need to understand what Secure Boot is, and what UEFI is, and which of the two you are actually talking about at any given time. We'll talk about UEFI first, and then we'll talk about Secure Boot as an 'extension' to UEFI, because that's basically what it is.

Bonus Historical Note: UEFI was not invented by, is not controlled by, and has never been controlled by Microsoft. Its predecessor and basis, EFI, was developed and published by Intel. UEFI is managed by the UEFI Forum. Microsoft is a member of the UEFI forum. So is Red Hat, and so is Apple, and so is just about every major PC manufacturer, Intel (obviously), AMD, and a laundry list of other major and minor hardware, software and firmware companies and organizations. It is a broad consensus specification, with all the messiness that entails, some of which we'll talk about specifically later. It is no one company's Evil Vehicle Of Evilness.

References

If you really want to understand UEFI, it's a really good idea to go and read the UEFI specification. You can do this. It's very easy. You don't have to pay anyone any money. I am not going to tell you that reading it will be the most fun you've ever had, because it won't. But it won't be a waste of your time. You can find it right here on the official UEFI site. You have to check a couple of boxes, but you are not signing your soul away to Satan, or anything. It's fine. As I write this, the current version of the spec is 2.4 Errata A, and that's the version this post is written with regard to.

There is no BIOS specification. BIOS is a *de facto* standard – it works the way it worked on actual IBM PCs, in the 1980s. That's kind of one of the reasons UEFI exists.

Now, to keep things simple, let's consider two worlds. One is the world of IBM PC compatible computers – hereafter referred to just as PCs – before UEFI and GPT (we'll come to GPT) existed. This is the world a lot of you are probably familiar with and may understand quite well. Let's talk about how booting works on PCs with BIOS firmware.

BIOS booting

It works, in fact, in a very, very simple way. On your bog-standard old-skool BIOS PC, you have one or more disks which have an MBR. The MBR is another de facto standard; basically, the very start of the disk describes the partitions on the disk in a particular format, and contains a 'boot loader', a very small piece of code that a BIOS firmware knows how to execute, whose job it is to boot the operating system(s). (Modern bootloaders frequently are much bigger than can be contained in the MBR space and have to use a multi-stage design where the bit in the MBR just knows how to load the next stage from somewhere else, but that's not important to us right now).

All a BIOS firmware knows, in the context of booting the system, is what disks the system contains. You, the owner of this BIOS-based computer, can tell the BIOS firmware which disk you want it to boot the system from. The firmware has no knowledge of anything beyond that. It executes the bootloader it finds in the MBR of the specified disk, and that's it. The firmware is no longer involved in booting.

In the BIOS world, absolutely all forms of multi-booting are handled above the firmware layer. The firmware layer doesn't really know what a bootloader is, or what an operating system is. Hell, it doesn't know what a partition is. All it can do is run the boot loader from a disk's MBR. You also cannot configure the boot process from outside of the firmware.

UEFI booting: background

OK, so we have our background, the BIOS world. Now let's look at how booting works on a UEFI system. Even if you don't grasp the details of this post, grasp this: *it is completely different*. Completely and utterly different from how BIOS booting works. You cannot apply any of your understanding of BIOS booting to native UEFI booting. You cannot make a little tweak to a system designed for the world of BIOS booting and apply it to *native* UEFI booting. You need to understand that it is a completely different world.

Here's another important thing to understand: many UEFI firmwares implement some kind of *BIOS compatibility mode*, sometimes referred to as a *CSM*. Many UEFI firmwares can boot a system just like a BIOS firmware would – they can look for an MBR on a disk, and execute the boot loader from that MBR, and leave everything subsequently up to that bootloader. People sometimes

incorrectly refer to using this feature as 'disabling UEFI', which is *linguistically* nonsensical. You cannot 'disable' your system's firmware. It's just a stupid term. Don't use it, but understand what people really mean when they say it. They are talking about using a UEFI firmware's ability to boot the system 'BIOS-style' rather than native UEFI style.

What I'm going to describe is *native* UEFI booting. If you have a UEFI-based system whose firmware has the BIOS compatibility feature, and you decide to use it, and you apply this decision consistently, then as far as booting is concerned, you can pretend your system is BIOS-based, and just do everything the way you did with BIOS-style booting. If you're going to do this, though, just make sure you *do* apply it consistently. I really can't recommend strongly enough that you do *not* attempt to mix UEFI-native and BIOS-compatible booting of permanently-installed operating systems on the same computer, and *especially* not on the same disk. It is a terrible terrible idea and will cause you heartache and pain. If you decide to do it, don't come crying to me.

For the sake of sanity, I am going to assume the use of disks with a GPT partition table, and EFI FAT32 EFI system partitions. Depending on how deep you're going to dive into this stuff you *may* find out that it's not strictly speaking the case that you can *always* assume you'll be dealing with GPT disks and EFI FAT32 ESPs when dealing with UEFI native boot, but the UEFI specification is quite strongly tied to GPT disks and EFI FAT32 ESPs, and this is what you'll be dealing with in 99% of cases. Unless you're dealing with Macs, and quite frankly, *screw* Macs.

Edit note: the following sections (up to *Implications and Complications*) were heavily revised on 2014-01-26, a few hours after the initial version of this post went up, based on feedback from Peter Jones. Consider this to be v2.0 of the post. An earlier version was written in a somewhat less accurate and more confusing way.

UEFI native booting: how it actually works – background

OK, with that out of the way, let's get to the meat. This is how native UEFI booting actually works. It's probably helpful to go into this with a bit of high-level background.

UEFI provides *much* more infrastructure at the firmware level for handling system boot. It's nowhere near as simple as BIOS. Unlike BIOS, UEFI certainly does understand, to varying degrees, the concepts of 'disk partitions' and 'bootloaders' and 'operating systems'.

You can sort of look at the BIOS boot process, and look at the UEFI process, and see how the UEFI process extends various bits to address specific problems.

The BIOS/MBR approach to finding the bootloader is pretty janky, when you think about it. It's very 'special sauce': this particular tiny space at the front of the disk contains magic code that only really makes much sense to the system firmware and special utilities for writing it. There are several problems with this approach.

- It's inconvenient to deal with – you need special utilities to write the MBR, and just about the only way to find out what's in one is to dd the contents out and examine them.
- As noted above, the MBR itself is not big enough for many modern bootloaders. What they do is install a small part of themselves to the MBR proper, and the rest to the empty space on the disk between where the conventional MBR ends and the first partition begins. There's a rather big problem with this (well, the whole design is a big problem, but never mind), which is that there's no reliable convention for where the first partition should begin, so it's difficult to be sure there'll be enough space. One thing you usually can rely on is that there won't be *enough* space for some bootloader configurations.
- The design doesn't provide any standardized layer or mechanism for selecting boot targets other than disks…but people *want* to select boot targets other than disks. i.e. they want to have multiple bootable 'things' – usually operating systems – per disk. The only way to do this, in the BIOS/MBR world, is for the bootloaders to handle it; but there's no widely accepted convention for the right way to do this. There are many many different approaches, none of which is particularly interoperable with any of the others, none of which is a widely accepted standard or convention, and it's very difficult to write tooling at the OS / OS installation layer that handles multiboot cleanly. It's just a very messy design.

- The design doesn't provide a standard way of booting from anything *except* disks. We're not going to really talk about that in this article, but just be aware it's another advantage of UEFI booting: it provides a standard way for booting from, for instance, a remote server.
- There's no mechanism for levels above the firmware to configure the firmware's boot behaviour.

So you can imagine the UEFI Elves sitting around and considering this problem, and coming up with a solution. Instead of the firmware only knowing about disks and one 'magic' location per disk where bootloader code might reside, UEFI has much more infrastructure at the firmware layer for handling boot loading. Let's look at all the things it defines that are relevant here.

EFI executables

The UEFI spec defines an executable format and requires all UEFI firmwares be capable of executing code in this format. When you write a bootloader for native UEFI, you write in this format. This is pretty simple and straightforward, and doesn't need any further explanation: it's just a Good Thing that we now have a firmware specification which actually defines a common format for code the firmware can execute.

The GPT (GUID partition table) format

The GUID Partition Table format is very much tied in with the UEFI specification, and again, this isn't something particularly complex or in need of much explanation, it's just a good bit of groundwork the spec provides. GPT is just a standard for doing partition tables – the information at the start of a disk that defines what partitions that disk contains. It's a better standard for doing this than MBR/'MS-DOS' partition tables were in many ways, and the UEFI spec requires that UEFI-compliant firmwares be capable of interpreting GPT (it also requires them to be capable of interpreting MBR, for backwards compatibility). All of this is useful groundwork: what's going on here is the spec is establishing certain capabilities that everything above the firmware layer can rely on the firmware to have.

EFI system partitions

I actually really wrapped my head around the EFI system partition concept while revising this post, and it was a great 'aha!' moment. Really, the concept of 'EFI system partitions' is just an answer to the problem of the 'special sauce' MBR space. The concept of some undefined amount of empty space at the start of a disk being 'where bootloader code lives' is a pretty crappy design, as we saw above. EFI system partitions are just UEFI's solution to that.[1]

The solution is this: we require the firmware layer to be capable of reading some specific types of filesystem. The UEFI spec requires that compliant firmwares be capable of reading the FAT12, FAT16 and FAT32 variants of the FAT format, in essence. In fact what it does is codify a particular interpretation of those formats as they existed at the point UEFI was accepted, and say that UEFI compliant firmwares must be capable of reading those formats. As the spec puts it:

"The file system supported by the Extensible Firmware Interface is based on the FAT file system. EFI defines a specific version of FAT that is explicitly documented and testable. Conformance to the EFI specification and its associate reference documents is the only definition of FAT that needs to be implemented to support EFI. To differentiate the EFI file system from pure FAT, a new partition file system type has been defined."

An 'EFI system partition' is really just any partition formatted with one of the UEFI spec-defined variants of FAT and given a specific GPT partition type to help the firmware find it. And the purpose of this is just as described above: allow everyone to rely on the fact that the firmware layer will definitely be able to read data from a pretty 'normal' disk partition. Hopefully it's clear why this is a better design: instead of having to write bootloader code to the 'magic' space at the start of an MBR disk, operating systems and so on can just create, format and mount partitions in a widely understood format and put bootloader code and anything else that they might want the firmware to read there.

The whole ESP thing seemed a bit bizarre and confusing to me at first, so I hope this section explains why it's actually a very sensible idea and a good design – the bizarre and confusing thing is really the BIOS/MBR design, where the only way for you to write something from the OS layer that you knew the firmware layer could consume was to write it into some (but you didn't know how much) Magic Space at the start of a disk, a convention which isn't actually codified anywhere. That really *isn't* a very sensible or understandable design, if you step back and take a look at it.

As we'll note later, the UEFI spec tends to take a 'you must at least do these things' approach – it rarely prohibits firmwares from doing anything else. It's not against the spec to write a firmware that can execute code in other formats, read other types of partition table, and read partitions formatted with filesystems other than the UEFI variants of FAT. But a UEFI compliant firmware must *at least* do all these things, so if you are writing an OS or something else that you want to run on *any* UEFI compliant firmware, this is why the EFI system partition concept is so important: it gives you (at least in theory) 100% confidence that you can put an EFI executable on a partition formatted with the UEFI FAT implementation and the correct GPT partition type, and the system firmware will be able to read it. This is the thing you can take to the bank, like 'the firmware will be able to execute some bootloader code I put in the MBR space' was in the BIOS world.

So now we have three important bits of groundwork the UEFI spec provides: thanks to these requirements, any other layer can confidently rely on the fact that the firmware:

- Can read a partition table
- Can access files in some specific filesystems
- Can execute code in a particular format

This is much more than you can rely on a BIOS firmware being capable of. However, in order to complete the vision of a firmware layer that can handle booting multiple targets – not just disks – we need one more bit of groundwork: there needs to be a mechanism by which the firmware *finds* the various possible boot targets, and a way to configure it.

The UEFI boot manager

The UEFI spec defines something called the *UEFI boot manager*. (Linux distributions contain a tool called `efibootmgr` which is used to manipulate the configuration of the UEFI boot manager). As a sample of what you can expect to find if you do read the UEFI spec, it defines the UEFI boot manager thusly:

"The UEFI boot manager is a firmware policy engine that can be configured by modifying architecturally defined global NVRAM variables. The boot manager will attempt to load UEFI drivers and UEFI applications (including UEFI OS boot loaders) in an order defined by the global NVRAM variables."

Well, that's that cleared up, let's move on. 😉 No, not really. Let's translate that to Human. With only a reasonable degree of simplification, you can think of the UEFI boot manager as being a boot menu. With a BIOS firmware, your firmware level 'boot menu' is, necessarily, the disks connected to the system at boot time – no more, no less. This is not true with a UEFI firmware.

The UEFI boot manager can be configured – simply put, you can add and remove entries from the 'boot menu'. The firmware can also (it fact the spec requires it to, in various cases) effectively 'generate' entries in this boot menu, according to the disks attached to the system and possibly some firmware configuration settings. It can also be examined – you can look at what's in it.

One rather great thing UEFI provides is a mechanism for doing this *from other layers*: you can configure the system boot behaviour from a booted operating system. You can do all this by using the `efibootmgr` tool, once you have Linux booted via UEFI somehow. There are Windows tools for it too, but I'm not terribly familiar with them. Let's have a look at some typical `efibootmgr` output, which I stole and slightly tweaked from the Fedora forums:

```
[root@system directory]# efibootmgr -v
BootCurrent: 0002
Timeout: 3 seconds
BootOrder: 0003,0002,0000,0004
Boot0000* CD/DVD Drive  BIOS(3,0,00)
Boot0001* Hard Drive    HD(2,0,00)
Boot0002* Fedora        HD(1,800,61800,6d98f360-cb3e-4727-8fed-5ce0c040365d)File(\EFI\fedora\grubx64.efi)
Boot0003* opensuse      HD(1,800,61800,6d98f360-cb3e-4727-8fed-5ce0c040365d)File(\EFI\opensuse\grubx64.efi)
Boot0004* Hard Drive    BIOS(2,0,00)P0: ST1500DM003-9YN16G        .
[root@system directory]#
```

This is a nice clean example I stole and slightly tweaked from the Fedora forums. We can see a few things going on here.

The first line tells you which of the 'boot menu' entries you are *currently* booted from. The second is pretty obvious (if the firmware presents a boot menu-like interface to the UEFI boot manager, that's the timeout before it goes ahead and boots the default entry). The BootOrder is the order in which the entries in the list will be tried. The rest of the output shows the actual boot entries. We'll describe what they actually do later.

If you boot a UEFI firmware entirely normally, without doing any of the tweaks we'll discuss later, what it ought to do is try to boot from each of the 'entries' in the 'boot menu', in the order listed in *BootOrder*. So on this system it would try to boot the entry called 'opensuse', then if that failed, the one called 'Fedora', then 'CD/DVD Drive', and then the second 'Hard Drive'.

UEFI native booting: how it actually works – boot manager entries

What does these entries actually *mean*, though? There's actually a *huge* range of possibilities that makes up rather a large part of the complexity of the UEFI spec all by itself. If you're reading the spec, pour yourself an extremely large shot of gin and turn to the EFI_DEVICE_PATH_PROTOCOL section, but note that this is a generic protocol that's used for other things than booting – it's UEFI's Official Way Of Identifying Devices For All Purposes, used for boot manager entries but also for all sorts of other purposes. Not every possible EFI device path makes sense as a UEFI boot manager entry, for obvious reasons (you're probably not going to get too far trying to boot from your video adapter). But you can certainly have an entry that points to, say, a PXE server, not a disk partition. The spec has lots of bits defining valid non-disk boot targets that can be added to the UEFI boot manager configuration.

For our purposes, though, lets just consider fairly normal disks connected to the system. In this case we can consider three types of entry you're likely to come across.

BIOS compatibility boot entries

Boot0000 and Boot0004 in this example are actually BIOS compatibility mode entries, not UEFI native entries. They have not been added to the UEFI boot manager configuration by any external agency, but generated by the firmware itself – this is a common way for a UEFI firmware to implement BIOS compatibility booting, by generating UEFI boot manager entries that trigger a BIOS-compatible boot of a given device. How they *present this to the user* is a different question, as we'll see later. Whether you see any of these entries or not will depend on your particular firmware, and its configuration. Each of these entries just gives a name – 'CD/DVD Drive', 'Hard Drive' – and says "if this entry is selected, boot this disk (where 'this disk' is `3,0,00` for Boot0000 and `2,0,00` for Boot0004) in BIOS compatibility mode".

‘Fallback path’ UEFI native boot entries

Boot0001 is an entry (fictional, and somewhat unlikely, but it’s for illustrative purposes) that tells the firmware to try and boot from a particular disk, and in UEFI mode not BIOS compatibility mode, but doesn’t tell it anything more. It doesn’t specify a particular boot target on the disk – it just says to boot the disk.

The UEFI spec defines a sort of ‘fallback’ path for booting this kind of boot manager entry, which works in principle somewhat like BIOS drive booting: it looks in a standard location for some boot loader code. The details are different, though.

What the firmware will actually do when trying to boot in this way is reasonably simple. The firmware will look through each EFI system partition on the disk in the order they exist on the disk. Within the ESP, it will look for a file with a specific name and location. On an x86-64 PC, it will look for the file `\EFI\BOOT\BOOTx64.EFI`. What it actually looks for is `\EFI\BOOT\BOOT{machine type short-name}.EFI` – ‘x64’ is the “machine type short-name” for x86-64 PCs. The other possibilities are `BOOTIA32.EFI` (x86-32), `BOOTIA64.EFI` (Itanium), `BOOTARM.EFI` (AArch32 – that is, 32-bit ARM) and `BOOTAA64.EFI` (AArch64 – that is, 64-bit ARM). It will then execute the first qualifying file it finds (obviously, the file needs to be in the executable format defined in the UEFI specification).

This mechanism is not designed for booting permanently-installed OSes. It’s more designed for booting hotpluggable, device-agnostic media, like live images and OS install media. And this is indeed what it’s usually used for. If you look at a UEFI-capable live or install medium for a Linux distribution or other OS, you’ll find it has a GPT partition table and contains a FAT-formatted partition at or near the start of the device, with the GPT partition type that identifies it as an EFI system partition. Within that partition there will be a \EFI\BOOT directory with at least one of the specially-named files above. When you boot a Fedora live or install medium in UEFI-native mode, this is the mechanism that is used. The BOOTx64.EFI (or whatever) file handles the rest of the boot process from there, booting the actual operating system contained on the medium.

Full UEFI native boot entries

Boot0002 and Boot0003 are ‘typical’ entries for operating systems permanently installed to permanent storage devices. These entries show us the full power of the UEFI boot mechanism, by not just saying “boot from this disk”, but “boot this specific bootloader in this specific location on this specific disk”, using all the ‘groundwork’ we talked about above.

Boot0002 is a boot entry produced by a UEFI-native Fedora installation. Boot0003 is a boot entry produced by a UEFI-native OpenSUSE installation. As you may be able to tell, all they’re saying is “load this file from this partition”. The partition is the `HD(1,800,61800,6d98f360-cb3e-4727-8fed-5ce0c040365d)` bit: that’s referring to a specific partition (using the EFI_DEVICE_PATH_PROTOCOL, which I’m really not going to attempt to explain in any detail – you don’t necessarily need to know it, if you interact with the boot manager via the firmware interface and `efibootmgr`). The file is the `File(\EFI\opensuse\grubx64.efi)` bit: that just means “load the file in this location on the partition we just described”. The partition in question will almost always be one that qualifies as an EFI system partition, because of the considerations above: that’s the type of partition we can trust the firmware to be able to access.

This is the mechanism the UEFI spec provides for operating systems to make themselves available for booting: the operating system is intended to install a bootloader which loads the OS kernel and so on to an EFI system partition, and add an entry to the UEFI boot manager configuration with a name – obviously, this will usually be derived from the operating system’s name – and the location of the bootloader (in EFI executable format) that is intended for loading that operating system.

Linux distributions use the `efibootmgr` tool to deal with the UEFI boot manager. What a Linux distribution actually does, so far as bootloading is concerned, when you do a UEFI native install is really pretty simple: it creates an EFI system partition if one does not already exist, installs an EFI boot loader with an appropriate configuration – often grub2-efi, but there are others – into a correct path in the EFI system partition, and calls `efibootmgr` to add an appropriately-named UEFI boot manager entry pointing to its boot loader. Most distros will use an existing EFI system partition if there is one, though it’s perfectly valid to create a new one and use

that instead: as we've noted, UEFI is a permissive spec, and if you follow the design logically, there's really no problem with having just as many EFI system partitions as you want.

Configuring the boot process (the firmware UI)

The above describes the basic mechanism the UEFI spec defines that manages the UEFI boot process. It's important to realize that your firmware user interface may well not represent this mechanism very clearly. Unfortunately, the spec intentionally refrains from defining how the boot process should be represented to the user or how the user should be allowed to configure it, and what that means – since we're dealing with firmware engineers – is that every firmware does it differently, and some do it insanely.

Many firmwares do have fairly reasonable interfaces for boot configuration. A good firmware design will at least show you the boot order, with a reasonable representation of the entries on it, and let you add or remove entries, change the order, or override the order for a specific boot (by changing it just for that boot, or directly instructing the firmware to boot a particular menu entry, or even giving you the option to simply say "boot this disk", either in BIOS compatibility mode or UEFI 'fallback' mode – my firmware does this). Such an interface will often show 'full' UEFI native boot entries (like the Fedora and openSUSE examples we saw earlier) only by their name; you have to examine the `efibootmgr -v` output to know precisely what these entries will actually try and *do* when invoked.

Some firmwares try to abstract and simplify the configuration, and may do a good or a bad job of it. For instance, if you have an option to 'enable or disable' BIOS compatibility mode, what it'll really likely *do* is configure whether the firmware adds BIOS compatibility entries for attached drives to the UEFI boot manager configuration or not. If you have an option to 'enable or disable' UEFI native booting, what likely really happens when you 'disable' it is that the firmware changes the UEFI boot manager configuration to leave all UEFI-native entries out of the BootOrder.

The key point to remember is that any configuration option inside your firmware interface which is to do with booting is really, behind the scenes, configuring the behaviour of the UEFI boot manager. If you understand all the stuff we've discussed above, you may well find it easier to figure out what's *really* happening when you twiddle the knobs your firmware interface exposes.

In the BIOS world, you'll remember, you don't always find that systems are configured to try and boot from removable drives – CD, USB – before booting from permanent drives. Some are, and some aren't. Some will try CD before the hard disks, but not USB. People have got fairly used to having to check the BIOS configuration to ensure the boot order is 'correct' when trying to install a new operating system.

This applies to the UEFI world too, but because of the added flexibility/complexity of the UEFI boot manager mechanism, it can look unfamiliar and scary.

If you want to ensure that your system tries to boot from removable devices using the 'fallback' mechanism before it tries to boot 'permanent' boot entries – as you will want to do if you want to, say, install Fedora – you need this to be the default for your firmware, or you need to be able to tell the firmware this. Depending on your firmware's interface, you may find there is a 'menu entry' for each attached removable device and you just have to adjust the boot order to put it at the top of the list, or you may find that there is the mechanism to directly request 'UEFI fallback boot of this particular disk', or you may find that the firmware tries to abstract the configuration somehow. We just don't know, and that makes writing instructions for this quite hard. But now you broadly understand how things work behind the scenes, you may find it easier to understand your firmware user interface's representation of that.

Configuring the boot process (from an operating system)

As we've noted above, unlike in the BIOS world, you can actually configure the UEFI boot process from the operating system level. If you have an insane firmware, you may *have* to do this in order to achieve what you want.

You can use the `efibootmgr` tool mentioned earlier to add, delete and modify entries in the UEFI boot manager configuration, and actually do quite a lot of other stuff with it too. You can change the boot order. You can tell it to boot some particular entry in the list on the next boot, instead of using the *BootOrder* list (if you or some other tool has configured this to happen, your `efibootmgr -v` output will include a *BootNext* item stating which menu entry will be loaded on the next boot). There are tools for Windows that can do this stuff from Windows, too. So if you're really struggling to manage to do whatever it is you want to do with UEFI boot configuration from your firmware interface, but you *can* boot a UEFI native operating system of some kind, you may want to consider doing your boot configuration from that operating system rather than from the firmware UI.

So to recap:

- Your UEFI firmware contains something very like what you think of as a boot menu.
- You can query its configuration with `efibootmgr -v`, from any UEFI-native boot of a Linux OS, and also *change* its configuration with `efibootmgr` (see the man page for details).
- This 'boot menu' can contain entries that say 'boot this disk in BIOS compatibility mode', 'boot this disk in UEFI native mode via the fallback path' (which will use the 'look for BOOT(something).EFI' method described above), or 'boot the specific EFI format executable at this specific location (almost always on an EFI system partition)'.
- The nice, clean design that the UEFI spec is trying to imply is that all operating systems should install a bootloader of their own to an EFI system partition, add entries to this 'boot menu' that point to themselves, and butt out from trying to take control of booting anything else.
- Your firmware UI has free rein to represent this mechanism to you in whatever way it wants, and it may do this well, or it may do this poorly.

Installing operating systems to UEFI-based computers

Let's have a quick look at some specific consequences of the above that relate to installing operating systems on UEFI computers.

UEFI native and BIOS compatibility booting

Here's a very very simple one which people sometimes miss:

- If you boot the installation medium in 'UEFI native' mode, it will do a UEFI native install of the operating system: it will try to write an EFI-format bootloader to an EFI system partition, and attempt to add an entry to the UEFI boot manager 'boot menu' which loads that bootloader.
- If you boot the installation medium in 'BIOS compatibility' mode, it will do a BIOS compatible install of the operating system: it will try to write an MBR-type bootloader to the magic MBR space on a disk.

This applies (with one minor caveat I'm going to paper over for now) to all OSes of which I'm aware. So you probably want to make sure you understand how, in your firmware, you can choose to boot a removable device in UEFI native mode and how you can choose to boot it in BIOS compatibility mode, and make sure you pick whichever one you actually want to use for your installation.

You really cannot do a completely successful UEFI-native installation of an OS if you boot its installation medium in BIOS compatibility mode, because the installer will not be able to configure the UEFI boot manager (this is only possible when booted UEFI-native).

It is theoretically possible for an OS installer to install the OS in the BIOS style – that is, write a bootloader to a disk's MBR – after being booted in UEFI native mode, but most of them *won't* do this, and that's probably sensible.

Finding out which mode you're booted in

It is possible that you might find yourself with your operating system installer booted, and not sure whether it's actually booted in UEFI native mode or BIOS compatibility mode. Don't panic! It's pretty easy to find out which, in a few different ways. One of the easiest is just to try and talk to the UEFI boot manager. If what you have booted is a Linux installer or environment, and you can get to a shell (ctrl-alt-f2 in the Fedora installer, for instance), run `efibootmgr -v`. If you're booted in UEFI native mode, you'll get your UEFI boot manager configuration, as shown above. If you're booted in BIOS compatibility mode, you'll get something like this:

```
Fatal: Couldn't open either sysfs or procfs directories for accessing EFI variables.
Try 'modprobe efivars' as root.
```

If you've booted some other operating system, you can try running a utility native to that OS which tries to talk to the UEFI boot manager, and see if you get sensible output or a similar kind of error. Or you can examine the system logs and search for 'efi' and/or 'uefi', and you'll probably find some kind of indication.

Enabling UEFI native boot

To be bootable in UEFI native mode, your OS installation medium must obviously actually comply with all this stuff we've just described: it's got to have a GPT partition table, and an EFI system partition with a bootloader in the correct 'fallback' path – \EFI\BOOT\BOOTx64.EFI (or the other names for the other platforms). If you're having trouble doing a UEFI native boot of your installation medium and can't figure out why, check that this is actually the case. Notably, when using the `livecd-iso-to-disk` tool to write a Fedora image to a USB stick, you must pass the `--efi` parameter to configure the stick to be UEFI bootable.

Forcing BIOS compatibility boot

If your firmware seems to make it very difficult to boot from a removable medium in BIOS compatibility mode, but you really want to do that, there's a handy trick you can use: just make the medium not UEFI native bootable at all. You can do this pretty easily by just wiping all the EFI system partitions. (Alternatively, if using `livecd-iso-to-disk` to create a USB stick from a Fedora image, you can just leave out the `--efi` parameter and it won't be UEFI bootable). If at that point your firmware refuses to boot it in BIOS compatibility mode, commence swearing at your firmware vendor (if you didn't already).

Disk formats (MBR vs. GPT)

Here's another very important consideration:

- If you want to do a 'BIOS compatibility' type installation, you probably want to install to an MBR formatted disk.
- If you want to do a UEFI native installation, you probably want to install to a GPT formatted disk.

Of course, to make life complicated, many firmwares *can* boot BIOS-style from a GPT formatted disk. UEFI firmwares are in fact technically *required* to be able to boot UEFI-style from an MBR formatted disk (though we are not particularly confident that they all really can). But you really should avoid this if at all possible. This consideration is quite important, as it's one that trips up quite a few people. For instance, it's a bad idea to boot an OS installer in UEFI native mode and then attempt to install to an MBR formatted disk without reformatting it. This is very likely to fail. Most modern OS installers will automatically reformat the disk in the correct format if you allow them to completely wipe it, but if you try and tell the installer 'do a UEFI native installation to this MBR formatted disk and don't reformat it because it has data on it that I care about', it's very likely to fail, even though this configuration is technically covered in the UEFI specification. Specifically, Windows and Fedora at least explicitly disallow this configuration.

Checking the disk format

You can use the `parted` utility to check the format of a given disk:

```
[adamw@adam Downloads]$ sudo parted /dev/sda
GNU Parted 3.1
Using /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) p
Model: ATA C300-CTFDDAC128M (scsi)
Disk /dev/sda: 128GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End    Size   Type     File system  Flags
 1      1049kB  525MB  524MB  primary  ext4         boot
 2      525MB   128GB  128GB  primary               lvm

(parted)
```

See that `Partition table: msdos`? This is an MBR/MS-DOS formatted disk. If it was GPT-formatted, that would say `gpt`. You can reformat the disk with the other type of partition table by doing `mklabel gpt` or `mklabel msdos` from within parted. **This will destroy the contents of the disk.**

With most OS installers, if you pick a disk configuration that blows away the entire contents of the target disk, the installer will automatically reformat it using the most appropriate configuration for the type of installation you're doing, but if you want to use an existing disk without reformatting it, you're going to have to check how it's formatted and take this into account.

Handling EFI system partition if doing manual partitioning

I can only give authoritative advice for Fedora here, but the gist may be useful for other distros / OSes.

If you allow Fedora to handle partitioning for you when doing a UEFI native installation – and you use a GPT-formatted disk, or allow it to reformat the disk (by deleting all existing partitions) – it will handle the EFI system partition stuff for you.

If you use custom partitioning, though, it will expect you to provide an EFI system partition for the installer to use. If you don't do this, the installer will complain (with a somewhat confusing error message) and refuse to let you start the installation.

So if you're doing a UEFI native install and using custom partitioning, you need to ensure that a partition of the 'EFI system partition' type is mounted at `/boot/efi` – this is where Fedora expects to find the EFI system partition it's using. If there is an existing EFI system partition on the system, just set its mount point to `/boot/efi`. If there is not an EFI system partition yet, create a partition, set its type to *EFI system partition*, make it at least 200MB big (500MB is good), and set its mount point to `/boot/efi`.

A specific example

To boil down the above: if you bought a Windows 8 or later system, you *almost* certainly have a UEFI native install of Windows to a GPT-formatted disk. This means that if you want to install another OS alongside that Windows install, you almost certainly want to do a UEFI-native installation of your other OS. If you don't like all this UEFI nonsense and want to go back to the good old world you're familiar with, you will, I'm afraid, have to blow away the UEFI-native Windows installation, and it would be a good idea to reformat the disk to MBR.

Implications and Complications

So, that's how UEFI booting works, at least a reasonable approximation. When I describe it like that, it almost all makes sense, right?

However, all is not sweetness and light. There are problems. There always are.

Attentive readers may have noticed that I've talked about the UEFI spec *providing* a mechanism. This is accurate, and important. As the UEFI spec is a 'broad consensus' sort of thing, one of its major shortcomings (looked at from a particular perspective) is that it's nowhere near prescriptive enough.

If you read the UEFI spec critically, its basic approach is to define a set of functions that UEFI compliant firmwares must support. What it doesn't do a lot of at all is strictly requiring things to be done in any particular way, or not done in any particular way.

So: the spec says that a system firmware must do all the stuff I've described above, in order to be considered a UEFI-compliant firmware. The spec, however, doesn't talk about what operating systems 'should' or 'must' do at all, and it doesn't say that firmwares must *not* support (or no-one may expect them to support, or whatever)…anything at all. If you're making a UEFI firmware, in other words, you have to support GPT formatted disks, and FAT-formatted EFI system partitions, and you must read UEFI boot manager entries in the standard format, and you must do this and that and the other – but you can also do any *other* crap you like.

It's pretty easy to read certain implications from the spec – it carefully sets up this nice mechanism for handling OS (or other 'bootable thing') selection at the firmware level, for instance, with the clear implication "hey, it'd be great if all OSes were written to this mechanism". But the UEFI spec doesn't *require* that, and neither does any other widely-respected specification.

So, what happens in the real world is that we wind up with really dumb crap. Apple, for instance, ships at least *some* Macs with their bootloaders in an HFS+ partition. The spec says a UEFI-compliant firmware must support UEFI FAT partitions with the specific GPT partition type that identifies them as an "EFI system partition", but it doesn't say the firmware can't *also* recognize some other filesystem type and load a bootloader from that. (Whether you consider such a partition to be an "EFI system partition" or not is an interesting philosophical conundrum, but let's skate right over that for now).

The world would pretty clearly be a better place if everyone just damn well used the EFI system partition format the spec goes to such great pains to define, but Apple is Apple and we can't have nice things, so Apple went right ahead and wrote firmwares that *also* can read and load code from HFS+ partitions, and now everyone else has to deal with that or tell Macs to go and get boned. Apple also goes quite a long way beyond the spec in its boot process design, and if you want your alternative OS to show up on its graphical boot menu with a nice icon and things, you have to do more than what the UEFI spec would suggest.

There are various similar incredibly annoying corner cases we've come across, but let's not go into them all right now. This post is long enough.

Also, as we noted earlier, the spec makes no requirements as to how the mechanism should be represented to the user. So if a couple of software companies write OSes to behave 'nicely' according to the conventions the spec is clearly designed to back, and

install EFI boot loaders and define EFI boot manager entries with nice clear names – like, oh say, “Fedora” and “Windows” – they are implicitly relying on the firmware to then give the user *some* kind of sane interface *somewhere* relatively discoverable that lets them choose to boot “Windows” or “Fedora”. The more firmwares don’t do a good job of this, the less willing OS engineers will be to rely on the ‘proper’ conventions, and the more likely they’ll be to start rebuilding ugly hacks above the firmware level.

To be fair, we could do somewhat more at the OS level. We could present all those neat efibootmgr capabilities rather more obviously – we can use that ‘don’t respect BootOrder on the next boot, but instead boot *this*‘ capability, for instance, and have ‘Reboot to Windows’ as an option. It’d be kinda nice if someone looked at exposing all this functionality somewhere more obvious than efibootmgr. Windows 8 systems do use this, to some extent – you can reboot your system to the firmware UI from the Windows 8 settings menus, for instance. But still.

All this is really incredibly frustrating, because UEFI is so *close* to making things really a lot better. The BIOS approach doesn’t provide any kind of convention or standard for multibooting at all – it *has* to be handled entirely above the firmware level. We (the industry) could have come up with some sort of convention for handling multiboot, but we never did, so it just became a multiple-decade epic fail, where each operating system came up with its own approach and lots of people wrote their own bootloaders which tried to subsume all the operating systems and all the operating systems and independent bootloaders merrily fought like cats in a sack. I mean, pre-UEFI multibooting is such a clusterf**k it’s not even worth going into, it’s broken sixteen ways from Sunday by definition.

If UEFI – or a spec built on top of it – had just *mandated* that everybody follow the conventions UEFI carefully establishes, and *mandated* that firmwares provide a sensible user interface, the win would have been epic. But it doesn’t, so it’s entirely possible that in a UEFI world things will be *even worse than they were in the BIOS world*. If many more firmwares show up that don’t present a good UI for the UEFI boot manager mechanism, what could happen is that OS vendors give up on the UEFI boot manager mechanism (or decide to support it *and* alternatives, because choice!) and just reinvent the entire goddamn nightmare of BIOS multibooting *on top of UEFI* – and we’ll all have to deal with all of that, *plus* the added complication of the UEFI boot manager layer. You’ll have multiple bootloaders fighting to load multiple operating systems all on top of the whole UEFI boot manager mechanism which is just throwing a whole bunch of other variables into the equation.

This is not a prospect filling the mind of anyone who’s had to think about it with joy.

Still, it’s important to recognize that the sins of UEFI in this area are sins of omission – they are not sins of commission, and they’re not really the result of evil intent on anyone’s part. The entity you should really be angry with if you have an idiotic system firmware that doesn’t give you good access to the UEFI boot manager mechanism is not the UEFI forum, or Microsoft, and it *certainly* isn’t Fedora and even more *certainly* isn’t me ;). The entity you should be angry at is your system/motherboard manufacturer and the goddamn incompetents they hired to write the firmware, because the UEFI spec makes it really damn clear to anyone with two brain cells to rub together that it would be a very good idea to provide some kind of useful user interface to the UEFI boot manager, and any firmware which doesn’t do so is crap code by definition. Yes, the UEFI forum should’ve realized that firmware engineers couldn’t code their way out of a goddamned paper bag and just ordered them to do so, but still, it’s ultimately the firmware engineers who should be lined up against the nearest wall.

Wait, we can simplify that. “Any firmware is crap code”. Usually pretty accurate.

Secure Boot

So now we come, finally, to Secure Boot.

Secure Boot is not magic. It’s not complicated. OK, that’s a lie, it’s incredibly complicated, but the *theory* isn’t very complicated. And no, Secure Boot itself is not evil. I am entirely comfortable stating this as a fact, and you should be too, unless you think GPG is evil.

Secure Boot is defined in chapter 28 of the UEFI spec (2.4a, anyway). It's actually a pretty clever mechanism. But what it does can be described very, very simply. It says that the firmware can contain a set of signatures, and refuse to run any EFI executable which is not signed with one of those signatures.

That's it. Well, no, it really isn't, but that's a reasonably acceptable simplification. Security is hard, so there are all kinds of wibbly bits to implementing a really secure bootchain using Secure Boot, and mjg59 can tell you all about them, or you can pour another large shot of gin and read the whole of chapter 28. But that's the basic idea.

Using public key cryptography to verify the integrity of something is hardly a radical or evil concept. Pretty much all Linux distributions depend on it – we sign our packages and have our package managers go AWOOGA AWOOGA if you try to install a package which isn't signed with one of our keys. This isn't us being evil, and I don't think anyone's ever accused an OS of being evil for using public key cryptographic signing to establish trust in this way. Secure Boot is literally this exact same perfectly widely accepted mechanism, applied to the boot chain. Yet because a bunch of journalists *wildly* grasped the wrong end of the stick, it's widely considered to be slightly more evil than Hitler.

Secure Boot, as defined in the UEFI spec, says nothing at all about what the keys the firmware trusts should be, or where they should come from. I'm not going to go into all the goddamn details, because it gets stultifyingly boring and this post is too long already. But the executive summary is that the spec is utterly and entirely about defining a *mechanism* for doing cryptographic verification of a boot chain. It does not really even consider any kind of icky questions about the *policy* for doing so. It does nothing evil. It is as flexible as it could reasonably be, and takes care to allow for all the mechanisms involved to be configurable at multiple levels. The word 'Microsoft' is not mentioned. It is not in any way, shape, or form a secret agenda for Microsoft's domination of the world. If you doubt this, at the very bloody least, go and *read* it. I've given you all the necessary pointers. There is literally not a single legitimate reason I can think of for anyone to be *angry* with the idea "hey, it'd be neat if there was a mechanism for optional cryptographic verification of bootloader code in this firmware specification". None. Not one.

Secure Boot in the real world

Most of the unhappiness about Secure Boot is not really about Secure Boot the mechanism – whether the people expressing that unhappiness *think* it is or not – but about specific implementations of Secure Boot in the real world.

The only one we really care about is Secure Boot as it's implemented on PCs shipped with Microsoft Windows 8 or higher pre-installed.

Microsoft has these things called the Windows Hardware Certification Requirements. There they are. They are not Top Secret, Eyes Only, You Will Be Fed To Bill Gates' Sharks After Reading – they're right there on the Internet for anyone to read.

If you want to get cheap volume licenses of Windows from Microsoft to pre-install on your computers and have a nice "reassuring" 'Microsoft Approved!' sticker or whatever on the case, you have to comply with these requirements. That's all the force they have: they are not *actually* a part of the law of the United States or any other country, whatever some people seem to believe. Bill Gates cannot feed you to his sharks if you sell a PC that doesn't comply with these requirements, so long as you don't want a cheap copy of Windows to pre-install and a nice sticker. There is literally no requirement for a PC sold *outside* the Microsoft licensing program to configure Secure Boot in any particular way, or include Secure Boot at all. A PC that claims to have a UEFI 2.2 or later compliant firmware must implement Secure Boot, but can ship with it configured in literally absolutely any way it pleases (including turned off).

If you're going to have very loud opinions about Secure Boot, you have zero excuse for not going and reading the Microsoft certification requirements. Right now. I'll wait. You can search for "Secure Boot" to get to the relevant bit. It starts at "System.Fundamentals.Firmware.UEFISecureBoot".

You should read it. But here is a summary of what it says.

Computers complying with the requirements must:

- Ship with Secure Boot turned on (except for servers)
- Have Microsoft's key in the list of keys they trust
- Disable BIOS compatibility mode when Secure Boot is enabled (actually the UEFI spec requires this too, if I read it correctly)
- Support signature blacklisting

x86 computers complying with the requirements must additionally:

- **Allow** a physically present person to disable Secure Boot
- **Allow** a physically present person to enable Custom Mode, and modify the list of keys the firmware trusts

ARM computers complying with the requirements must additionally:

- **NOT allow** a physically present person to disable Secure Boot
- **NOT allow** a physically present person to enable Custom Mode, and modify the list of keys the firmware trusts

Yes. You read that correctly. The Microsoft certification requirements, for x86 machines, *explicitly require implementers to give a physically present user complete control over Secure Boot* – turn it off, or completely control the list of keys it trusts. Another important note here is that while the certification requirements state that the out-of-the-box list of trusted keys must *include* Microsoft's key, they don't say, for e.g., that it must *not include* any other keys. The requirements explicitly and intentionally allow for the system to ship with any number of other trusted keys, too.

These requirements aren't present entirely out of the goodness of Microsoft's heart, or anything – they're present in large part because other people explained to Microsoft that if they weren't present, it'd have a hell of a lawsuit on its hands[2] – but they are present. Anyone who actually understands UEFI and Secure Boot cannot possibly read the requirements any other way, they are extremely clear and unambiguous. They both clearly *intend to* and *succeed in* ensuring the owner of a certified system has complete control over Secure Boot.

If you have an x86 system that claims to be Windows certified but does not allow you to disable Secure Boot, it is in direct violation of the certification requirements, and you should certainly complain very loudly to someone. If a lot of these systems exist then we clearly have a problem and it might be time for that giant lawsuit, but so far I'm not aware of this being the case. All the x86-based, Windows-certified systems I've seen *have* had the 'disable Secure Boot' option in their firmwares.

Now, for ARM machines, the requirements are significantly more evil: they state exactly the opposite, that it must *not* be possible to disable Secure Boot and it must *not* be possible for the system owner to change the trusted keys. This is bad and wrong. It makes Microsoft-certified ARM systems into a closed shop. But it's worth noting it's no *more* bad or wrong than most other major ARM platforms. Apple locks down the bootloader on all iDevices, and most Android devices also ship with locked bootloaders.

If you're planning to buy a Microsoft-certified ARM device, be aware of this, and be aware that you will not be in control of what you can boot on it. If you don't like this, don't buy one. But also don't buy an iDevice, or an Android device with a locked bootloader (you can buy Android devices with unlocked or unlockable bootloaders, still, but you have to do your research).

As far as x86 devices go, though, right now, Microsoft's certification requirements actually *explicitly protect* your right to determine what can boot on your system. This is good.

Recommendations

The following are AdamW's General Recommendations On Managing System Boot, offered with absolutely no guarantees of accuracy, purity or safety.

- **If you can possibly manage it, have one OS per computer.** If you need more than one OS, buy more computers, or use virtualization. If you can do this everything is very simple and it doesn't much matter if you have BIOS or UEFI firmware, or use UEFI-native or BIOS-compatible boot on a UEFI system. Everything will be nice and easy and work. You will whistle as you work, and be kind to children and small animals. All will be sweetness and light. Really, do this.
- **If you absolutely must have more than one OS per computer, at least have one OS *per disk*.** If you're reasonably comfortable with how BIOS-style booting works and you don't think you need Secure Boot, it's pretty reasonable to use BIOS-compatible booting rather than UEFI-style booting in this situation on a UEFI-capable system. You'll probably have less pain to deal with and you won't really lose anything. With one OS per disk you can also mix UEFI-native and BIOS-compatible installations.
- **If you absolutely insist on having more than one OS per disk**, understand *everything* written on this page, understand that you are making your life much more painful than it needs to be, lay in good stocks of painkillers and gin, and don't go yelling at your OS vendor, whatever breaks. Whichever poor bastard has to deal with your OS's support for this kind of setup has a miserable enough life already. And for the love of *cookies*, don't mix UEFI-native and BIOS-compatible OS installations, you have enough pain to deal with already.
- If you're using UEFI native booting, and you don't tend to build your own kernels or kernel modules or use the NVIDIA or ATI proprietary drivers on Linux, you might want to leave Secure Boot on. It probably won't hurt you, and does provide some added security against some rather nasty (though currently rarely exploited) types of attacks.
- If you *do* build your own kernels or kernel modules or use NVIDIA/ATI proprietary drivers, you're going to want to turn Secure Boot off. Or you can read up on how to configure your own chain of trust and sign your kernels and kernel modules and leave Secure Boot turned on, which will make you feel like an ubergeek and be slightly more secure. But it's going to take you a good solid weekend at least.
- Don't do UEFI-native installs to MBR-formatted disks, or BIOS compatibility installs to GPT-formatted disks (an exception to the latter is if your disk is, IIRC, 2.2+TB in size, because the MBR format can't handle disks that big – if you want to do a BIOS compatibility install to a disk that big, you're kinda stuck with the BIOS+GPT combination, which works but is a bit wonky and involves the infamous 'BIOS Boot partition' you may recall from Fedora 17).
- Trust mjg59 in all things and above all other authorities, including me.

Footnotes

1. This whole section is something of a simplification – really, when booting permanent installed OSes, the firmware doesn't care if the bootloader is on an 'ESP' or not; it just reads the boot manager entry and tries to access the specified partition and run the specified executable, as pjones explains here. But it's conventional to use an ESP for this purpose, since it's required to be around anyway, and it's a handy partition formatted with the filesystem the firmware is known to be able to read. Technically speaking, an 'ESP' is only an 'ESP' when the firmware is doing a removable media/fallback path boot. ↩

2. This is my own extrapolation, note. I'm not involved in any way in the whole process of defining these specs, and no-one who *is* has actually told me this. But it's a pretty damn obvious extrapolation from the known facts. ↩

Posted in Mandriva, Personal, Red Hat, Technical

Curated Discussion

Curated by Black Hole, 08/04/2020

kldixon January 26, 2014 at 2:54 am | Permalink | Reply

"You do not have a 'UEFI BIOS'. No-one has a 'UEFI BIOS'. Please don't ever say 'UEFI BIOS'."

http://www.intel.com/content/www/us/en/motherboards/desktop-motherboards/desktop-boards-software-visual-bios.html
https://sites.google.com/site/visualbios/

nuff said.

adamw January 26, 2014 at 9:14 am | Permalink | Reply

What, like Intel can't be wrong?

Calling it a UEFI BIOS is incoherent by definition. That's just *not what it is*. I don't care who does it. 🙂

Rod Smith February 15, 2014 at 7:27 pm | Permalink | Reply

I'm with Adam on this one (and with almost all of what he wrote on this page, for that matter). If we're to do a battle of the links, try this one:

https://en.wikipedia.org/wiki/Uefi

To quote: "UEFI is meant to replace the Basic Input/Output System (BIOS) firmware interface"

…or, from the UEFI Forum (http://www.uefi.org/faq):

"BIOS is typically used to refer to an Intel® Architecture firmware implementation rooted in the IBM PC design. Based on older standards and methods, BIOS was originally coded in 16-bit real mode x86 assembly code and remained substantially unchanged until its recent decline in use.

By contrast, UEFI standards reflect the past 30 years of PC evolution by describing an abstract interface set for transferring control to an operating system or building modular firmware from one or more silicon and firmware suppliers. The abstractions of UEFI Forum specifications are designed to decouple development of producer and consumer code, allowing each to innovate more independently and with faster time-to-market for both. UEFI also overcame the hardware scaling limitations that the IBM PC design assumed, allowing its broad deployment across high-end enterprise servers to the embedded devices. UEFI is "processor architecture-agnostic," supporting x86, x64, ARM and Itanium."

I suspect that manufacturers use "BIOS" because the general public is familiar with the acronym and wouldn't know what's meant by "EFI" or "UEFI." Unfortunately, I believe that this choice is doing a lot of harm — people think that their new computers' "BIOSes" are basically the same as they've always been, just with a new "UEFI feature" that might be something like AHCI support or USB keyboard support in previous BIOS

iterations. In other words, the “BIOS” terminology masks the fact that EFI is fundamentally different from BIOS, as Adam has gone to great lengths to explain on this page.

adamw February 15, 2014 at 7:50 pm | Permalink | Reply

Wow, thanks a lot for nailing *exactly* the problem with calling it a BIOS, far more clearly than I did! That’s precisely it: calling it a ‘BIOS’ is yet another ‘confusion vector’. I mean, if you think of your UEFI firmware as ‘the BIOS’, how exactly is your brain going to process the concept that your BIOS has a BIOS compatibility mode? It’s not going to end well. 🙂

Neil Darlow January 26, 2014 at 3:15 am | Permalink | Reply

Hi Adam,

I first came across UEFI in November 2010 when I decided to build a new home server. I purchased an ASRock E350-M1 EPIA motherboard and was oblivious to the fact it had UEFI firmware.

Back at that time there were few GNU/Linux distributions that could actually complete a native UEFI install. Both Fedora and OpenSUSE could boot DVD media in UEFI mode but the installations always failed. I actually ended-up using ArchLinux to manage the UEFI components and ran everything else in KVM virtual machines under that.

Of course things are much different now and UEFI distribution installs are often successful. I was just pleased to note that, after reading this excellent tome, that I had somehow managed to implement UEFI boot corectly using my manual process. Actually there were a cartload of other technologies I implemented at the same time (e.g. unified login using OpenLDAP and winbind etc.) which have all become mainstream after I did them the hard way.

The one thing I have learned from this experience, which you pointed-out, is to always use the definitive reference material for a particular technology. Guides and HOWTOs found on the Internet are always a source of either inaccurate information or information based on experimentation and deduction rather than hard fact.

Keep up the good work my friend!

Regards,
Neil Darlow

emmet.curran January 26, 2014 at 4:29 am | Permalink | Reply

Thanks Adam.

I’ve had some headaches at work trying to fix broken machines running Win8 and had to wrestle with UEFI for a bit before I got the hang of it. This post makes it a bit clearer what exactly I’m wrestling with.

Win8, UEFI and Secure Boot etc seem to work perfectly fine when everything is working correctly, but can become a bit of a nightmare (at the start) when you’re trying to do something so simple as boot to a live cd or even safe mode when the

stupid thing won't start. Most people hate Win8 for its Metro UI – I hate it because I can't get into safe mode without pleading with it first.

Somebody January 26, 2014 at 7:05 am | Permalink | Reply

So effectively what uefi is, is ramming two more levels of complicated bootloader code down into the board's soldered on flashrom, and requiring that those two complicated bootloaders follow a set of poorly described specifications.

I really don't see how this makes anything easier than bios, particularly since a good SSD is a heck of a lot faster to read than a crappy flashrom.

Now for my story;
I have a laptop that supports some form of EFI/UEFI. Up to Fedora 17, I was able to pass the installer's kernel a parameter or two (might have been something like noefi or nogpt) to force the installer to work in legacy bios mode, and that would work. It would set up the standard partition format boot/root/home/swap, and the firmware would happily boot in bios compatibility mode.

So couple of days ago, I found out that the F20 installer absolutely does not support this any more. So I tried wiping the disks and letting it do it the way it wanted to, and a very funny thing happened; it would only actually boot about 50% of the time. Completely unacceptable. So I pulled the disk and shoved it into another machine that has a most-definitely-not-[U]EFI firmware and did the install there before transferring back to the laptop. That works 100%.

It would seem that Fedora is using [U]EFI to load some weird kind of shim that loads GRUB, effectively using EFI *as* a BIOS.

Either use it correctly, or don't use it at all. If working as [U]EFI, grub really shouldn't exist at all.

adamw January 26, 2014 at 11:13 am | Permalink | Reply

"So effectively what uefi is, is ramming two more levels of complicated bootloader code down into the board's soldered on flashrom"

Two more? No, it's only one level. The implementation isn't really soldered on anything – any given firmware's implementation can be adjusted / fixed with a firmware update, like any other element of the firmware. The *configuration* of the UEFI boot manager is done via NVRAM variables (that's what efibootmgr is actually twiddling).

"I really don't see how this makes anything easier than bios, particularly since a good SSD is a heck of a lot faster to read than a crappy flashrom."

You're not going to notice a speed difference in loading something like a couple of KiB of configuration. The reason it *potentially* makes it better than BIOS land is that the UEFI level is a much more sensible level for 'boot target selection' to take place than 'in the first few bytes of whichever disk you're booting from', a concept which comes with massively obvious layering violations. There are about fifty different ways to do boot target selection with MBR-based booting, none of which is compatible with any of the others, and all of which will happily fight with each other like cats in a sack if you don't know exactly what you're doing as you install all your OSes.

"Now for my story; […] "

You're misunderstanding quite a few things there, but the headline is that 'noefi' isn't a Fedora installer parameter. It's a Linux kernel parameter. It's kind of a problematic thing to do, really, because you're essentially doing a UEFI native boot and then pretending you didn't. I'd recommend avoiding it. What you really want to do in your case is instruct your firmware to do a BIOS compatibility mode boot of your install media, not a UEFI native boot. The complications of this are discussed in the post. If your firmware absolutely does not allow you to do this, and you can't use efibootmgr or similar to do it, what you can do is write your install medium in such a way that it's not EFI bootable – doesn't contain an ESP – and then the firmware will usually 'automatically' boot it in BIOS compatibility mode when you ask the firmware to boot it. To do this with a Fedora image when writing to USB, for instance, use livecd-iso-to-disk and do *not* pass –efi.

When doing a UEFI native install, Fedora uses grub2-efi as its OS-specific bootloader. You *can* do 'direct' boot of a Linux kernel using UEFI – i.e. have your Linux kernel be an EFI executable and write a UEFI boot manager entry which points to that kernel, so you booted direct from the firmware layer to the OS kernel. (Really, the kernel contains a 'stub' EFI bootloader which does the job of loading the kernel – it's much like having grub2-efi, just that the bits are all baked into the kernel). This is the 'UEFI stubs' thing a later commenter mentions. But it's not very common to do this, and it's really not an Inarguably Better approach than having an EFI bootloader between the firmware and the OS kernel.

Somebody January 26, 2014 at 7:11 pm | Permalink | Reply

I'm not actually misunderstanding. Regardless of which level the parameter affected, the end result was something that *worked*. "that" of course, being past-tense.

Yes, obviously I could build my own install disk, but for an end user to have to do that is really pushing too far, and in my case, far far more complicated than what I did to solve the issue. Like you've mentioned, the installer really has to be able to deal with all the corner cases of inappropriate behavior.

Also, yes, I do mean two extra levels of bootloader. MBR and whatever actually does the multiboot.

I am not disagreeing about there being some interesting aspects of UEFI, but it is quite a major problem that all of the implementations of it are proprietary. The less proprietary code I have to depend on, the better, even if the end result is marginally less efficient. My laptop is a perfect illustration of this.

adamw January 26, 2014 at 7:36 pm | Permalink | Reply

"Regardless of which level the parameter affected, the end result was something that *worked*. "that" of course, being past-tense."

The problem with this is: https://xkcd.com/1172/

You can find all kinds of ways of doing things that work, and are bad. Sometimes those ways stop working, and no-one wants to fix them, because they're bad. This may annoy you, but it's not wrong.

If you want to do a BIOS compatibility mode install of your OS, boot its installation medium in BIOS compatibility mode. Really. That's the *right way* to do it. Your firmware should make this possible relatively easily, and if it doesn't, this post covers all kinds of things that should help you achieve it. Doing a UEFI native boot of your install media and then attempting to fool it into thinking you didn't boot in UEFI native mode is really a messy way to go about it. If noefi really isn't working for you any

more that's some kind of bug somewhere, sure, but I don't develop an immediate urge to investigate that and fix it, because it's just not the mechanism you really want to be using for what you're trying to achieve.

adamw January 26, 2014 at 7:38 pm | Permalink | Reply

"Also, yes, I do mean two extra levels of bootloader. MBR and whatever actually does the multiboot."

What?

In the BIOS/MBR world you have some very simple logic in the firmware layer, and very complex bootloaders in MBRs.
In the UEFI world you have more complex logic in the firmware layer – but really it's still just executing bootloader code it finds on the hard disk, there's just more potential to configure this – and, ideally, somewhat simpler bootloaders on EFI system partitions.

Neither system viewed as a whole is inherently more complex. All the complexity that exists in the UEFI layers I describe in this post *also exists in the BIOS/MBR world*, often duplicated many, many times between different implementations, and with no standardization.

Have you read the modifications to the page I've made over the last day or so? They may explain this more clearly.

adamw January 27, 2014 at 8:27 pm | Permalink | Reply

As it happens, someone else complained about noefi not working, and I was poking about in that code today anyway, so I went and looked at it.

Turns out the kernel's behaviour with 'noefi' changed in a way which broke anaconda's check for UEFI in this case (UEFI-native boot, but 'noefi' passed on the cmdline). This kind of thing is exactly why I'd suggest not relying on noefi. 🙂

But if you try booting the installer with 'noefi inst.updates=http://www.happyassassin.net/updates/updates-1047904.img' , it should work. See https://bugzilla.redhat.com/show_bug.cgi?id=1047904 .

kevin June 20, 2017 at 7:16 am | Permalink | Reply

Bit of a necro revival here, but I'm install RHEL7.1 and booting anaconda with tboot (1.8.2). It seems that tboot requires 'noefi' to be on the cmdline or the kernel panics. Unfortunately, I can't find much information on why that is. Of course adding 'noefi' also causes anaconda to break because as you said, it's a native UEFI boot and so anaconda is trying to use grub2-efi, which I guess fails when EFI runtime services are disabled. Does that all sound correct? Any suggestions here? What's the correct solution in this type of case?

I'm thinking the code issue you called out in the linked bugreport is not the problem. I have not

tried supplying the updates image you linked, however I have modified that exact code manually inside the squashfs image and based on the stacktrace I'm seeing, it's actually grub2 that's failing – specifically, when pyanaconda tries to execute "grub2-install –no-floppy /dev/sda", because I'm getting: "/usr/lib/grub/x86_64-efi/modinfo.sh doesn't exist." At least, that's the error I get when running the command manually that I think pyanaconda is executing

kevin June 20, 2017 at 7:31 am | Permalink | Reply

Also, per your question from that bug report: "What would be the actual use case for the behaviour you describe? (Genuine question, not snark – I wish to know). Who would want to boot in UEFI native mode, pass 'noefi', and have the installer do what you suggest?"

The case that I have is a necessity to boot in UEFI mode because of 4k sector drives, and needing to use tboot (apparently requires 'noefi'), and yet anaconda fails. Maybe this is a use case you were wondering about.

adamw June 29, 2017 at 2:03 pm | Permalink | Reply

"The case that I have is a necessity to boot in UEFI mode because of 4k sector drives"

I'm not quite sure what you mean here; to my knowledge there's nothing about 4K sector sizes that requires the use of UEFI. Could you clarify?

q2dg January 26, 2014 at 7:30 am | Permalink | Reply

Hello. Using UEFIStubs is a good idea?

adamw January 26, 2014 at 6:13 pm | Permalink | Reply

It's an *interesting* idea. I haven't played with it enough myself to tell you whether it's a good one or not. I guess I'd say that in my experience all the really tricky problems and misunderstandings people have with UEFI happen at the firmware layer; I haven't really seen people have much trouble at the EFI bootloader layer. So I don't think using an EFI stub kernel boot approach is something that would make the lives of most people struggling with UEFI any easier. But it's sure a technically interesting approach, and if you feel like playing with it, have fun.

Tom February 10, 2014 at 7:12 am | Permalink | Reply

The gummiboot and refind boot managers use the efi stub

Rod Smith February 15, 2014 at 7:42 pm | Permalink | Reply

There are system-to-system differences in how specific boot loaders work. A couple of years ago, GRUB 2 was hideously unreliable, in my experience; it would often fail to boot kernels, would hang, and would otherwise misbehave. It's settled down a lot recently and tends to be much more reliable these days, but it's still the most ungainly and difficult-to-configure boot loader I've ever seen. It has the benefit of being worked on by many smart people so that it works reasonably well "out of the box" on MOST peoples' systems; but in those cases when it doesn't work, GRUB becomes a nightmare.

In such cases, using almost anything else is likely to be easier than struggling with GRUB. My Web page on the options can be helpful: http://www.rodsbooks.com/efi-bootloaders/

The EFI stub loader is one of these options, and in my experience it's quite reliable; however, some people on the Arch Linux forum have reported sporadic difficulties with it with 3.7 and later kernels. I have yet to see a single report of such problems on other distributions, though, so I suspect that there's something odd about the way the Arch kernels are being built that's contributing to those problems.

adamw February 15, 2014 at 7:52 pm | Permalink | Reply

I'm tempted to channel Churchill in describing grub2 – it's the worst bootloader we have, except for all the others…

Rod Smith February 15, 2014 at 8:34 pm | Permalink | Reply

Personally, I'd leave out the last five words. I'm not a fan of GRUB 2. Under EFI, I prefer using ANYTHING else to boot Linux.

eggplant January 26, 2014 at 9:32 am | Permalink | Reply

Congratulations! you finished the boring spec giving ?? tries! 🙂

A huge "thanks.rpm" for filtering & accumulating just what end users need to know for a good workable knowledge.

The main things to remember for a prospective computing device buyer nowadays:
1. "… it's entirely possible that in a UEFI world things will be even worse than they were in the BIOS world … … you should be angry at is your system/motherboard manufacturer …".

2. The big evil declaration is "… Disable BIOS compatibility mode when Secure Boot is enabled …" to be Microsoft Certified specially for ARM computers.

3. "Apple locks down the bootloader … most Android devices also ship with locked bootloaders"

4. Microsoft compliant Secure Boot is not the only evil, there is FOTA looking for you. Probably it already caught you. 😉

I think just “painkillers and gin” are not sufficient enough in this modern age <(;-)

adamw January 26, 2014 at 11:01 am | Permalink | Reply

BIOS compatibility mode really isn’t relevant to the ARM world. No-one’s ever shipped an ARM system with a BIOS. I don’t think any ARM UEFI firmwares implement BIOS compatibility at all, it would be a ridiculous thing to do.

The evil thing about the Windows ARM case is that the Windows certification requirements *for ARM* specifically state that the user must not be able to disable or re-configure Secure Boot – precisely the opposite of the requirements *for Windows*, which state that the user *must* be able to do those things. Understood from a ‘mobile world’ perspective, where people are comfortable with the concept of a locked bootloader, what that effectively does is enshrine bootloader locking on *ARM* Windows devices as a part of the platform definition. To be a Microsoft-certified ARM Windows device, the bootloader must be locked. As I said, that is indeed bad, though only exactly *as* bad as Apple.

chip January 26, 2014 at 10:13 am | Permalink | Reply

You base several arguments that microsoft (and others) have nothing to do with conspiring to produce and support poor technology.. I would like to introduce you to a new concept: http://en.wikipedia.org/wiki/Cartel …. Historically: http://en.wikipedia.org/wiki/Wintel …

Nothing should be written about UEFI other than it is constraining and complicated solution to projects like coreboot (Just give us the specs and docs please, thank you)… This does nothing to help motherboard manufacturers that will continue to produce crap bioses and boards.

adamw January 26, 2014 at 11:03 am | Permalink | Reply

Thanks for the condescension. I have a degree in history including several economic history modules. I am comfortable with the concept of a cartel.

Intel designed EFI as a technical solution to a range of technical problems. BIOS is not a good standard. It isn’t even a standard, in the first place, just a convention. It has its own huge set of problems and limitations which I didn’t go into in this article because it was out of scope. There is a genuine need on a technical level for a replacement for the BIOS, and UEFI is what we wound up with. Its imperfections are cock-ups, not conspiracies.

All the bits in the post where I complain about how the UEFI spec isn’t sufficiently prescriptive? Well, that goes in spades for BIOS because *there is no BIOS spec*. All you have to go on in writing a BIOS is ‘make it behave like other BIOSes’. This is ridiculous, of course. There are certainly BIOS implementations as idiotic as any UEFI implementation.

chip January 26, 2014 at 11:21 am | Permalink | Reply

Bios doesn't need a spec. The hardware does.

The manufacturers must clear the communication lines to people who develop with this hardware (from small to large organizations).

UEFI doesn't help this. It increases reliance on black box technology.

adamw January 26, 2014 at 6:14 pm | Permalink | Reply

Your comment is entirely incoherent and contains no information or argument of any value.

chip January 29, 2014 at 3:06 pm | Permalink | Reply

What i was referring to was your tone on coreboot and that BIOS is not a good standard, or that it needs a standard.

As i said, BIOS does not need a standard or specification. The hardware which it initializes should be clearly specified and documented.

Standardizing a BIOS does not help anyone.

Rod Smith February 15, 2014 at 7:50 pm | Permalink | Reply

By the logic that "Bios doesn't need a spec," it should be fine to write a BIOS that loads the boot loader code from the LAST sector on the disk, or the SECOND sector on the disk, rather than from the first sector on the disk. This would work, of course, if boot loader code were found there, but it would be completely incompatible with every disk that holds a BIOS boot loader. That's just one example of the need for standards in the BIOS arena; the BIOS includes dozens of system calls that are documented in an ad-hoc manner.

Mantas January 26, 2014 at 10:39 pm | Permalink | Reply

"Space for the MBR" is a bit wrong. The MBR is always exactly 512 bytes, the bootstrap code 440 bytes (rest is the partition table). The space before first partition is not part of the MBR. (Which just makes it even worse to assume that it'll be available…)

On the other hand, not all bootloaders use that space. For example, syslinux does not – its MBR code jumps directly to a file in the boot partition.

adamw January 26, 2014 at 11:41 pm | Permalink | Reply

Oh, you're right, I do believe – we usally call it the 'bootloader embedding space' or something like that, right? I'll have to refresh my memory on that and come back to it tomorrow, I'm just going off my memory of all the 'fun' we had with that crap several releases back. Thanks for the note.

Mantas January 27, 2014 at 8:39 am | Permalink | Reply

Personally I call it the space where my GPT is located… I think "embedding" is a common term with GRUB, yes.

Tom February 10, 2014 at 7:16 am | Permalink | Reply

'something like that' = post-mbr gap = "gap" between the end of the mbr and the beginning of the first partition

Rod Smith February 15, 2014 at 7:53 pm | Permalink | Reply

Today, though, not all disks use 512-byte sectors. I don't know offhand how a BIOS treats such disks from a boot perspective, although I'd wager that a lot of true BIOSes can't boot from such disks at all.

adamw February 15, 2014 at 7:57 pm | Permalink | Reply

Back when we tried doing GPT by default in Fedora 16 (IIRC), I think we found that only disks larger than 2TB were using 4K logical sectors. I *think* this still holds true today – smaller disks use 512B logical sectors, i.e. they present a 512B sector size to the system even if they really use 4K sectors internally (4K physical sectors).

Rod Smith February 15, 2014 at 8:31 pm | Permalink | Reply

Lots of USB enclosures these days seem to be using 4KiB logical sectors, no matter what the disk's size. At least, I've seen problem reports in online forums related to this. For instance, some enclosures translate to 4KiB logical sectors when using USB interfaces, but present the drive's native (usually 512-byte) logical sector size when connected via eSATA. Of course, that's a recipe for disaster!

PoMo February 1, 2014 at 6:44 am | Permalink | Reply

Thank your for this great essay, I enjoyed reading it a lot.

* You strongly discourage multibooting from the same disk. Could you shed some light on this quote in particular: "understand that you are making your life much more painful than it needs to be, [..], and don't go yelling at your OS vendor,

whatever breaks."
From what I read in your essay, your main beef with implementations of uefi is that motherboards may _display_ the boot-choice badly or not at all. Apart from this UI issue, is there anything else that's painfull? And what else is likely/known to break?

Your text gave me the (possbily wrong) impression that you could do everything with efibootmgr, and with that, bypassing whatever downsides the motherboard UI has.

* This leads me to a dreamy crazy question: is it imaginable that we'll see a grub (or something that looks like grub but isn't), that massively uses what efibootmgr uses. I could imagine this having its own entry in the efi list, and being the default boot entry. The displayed list would just show the efi entries, selecting an entry would use the nextboot efi feature, a timeout would nextboot a custom default (not changing the efi default, which would keen pointing to this grub-like thing).

Letting my mind roam freely, this could bypass the problem of motherboards not displaying the list in the same fashion, and OSes being able to rely on it being there and knowing how it presents choice to the user, making it not being every individual OSes headache. (Also gone be the days of OSes overwriting each other's bootmanagers). Any installed OS would just need to add itself to the efi boot list. Done. (And add such grub-like thing if there isn't one yet, and make it the default).

With a few naming conventions, it could even add temporary boot entries (like for one-time modifying kernel params) which it would delete again on next occasion.

The one obvious downside would be that you're always rebooting at least once. first boot going into this grub-like thing, and second into the os you pick in it. But I'd accept that for all the advantages I can think of 😉

adamw February 1, 2014 at 9:05 am | Permalink | Reply

"Apart from this UI issue, is there anything else that's painfull? And what else is likely/known to break?"

If you manage to get a given multiboot setup working, and don't go poking it with any sticks, it *ought* to keep working. It's mainly the deployment time where people have trouble, and then understanding what actually constitutes their boot config (and hence should not be poked with sticks) after that. I mean, it can certainly work; I just see so many people struggling with it and misunderstanding stuff that I get concerned. That section was slightly tongue-in-cheek, but with a serious point: it really is easier if you can just stick to an OS per machine or per disk if you can. It may just be part of my general inclination after years of fiddling with PCs and trying to help other people fiddle with them: I really, really believe in the 'choice is an excellent way to shoot yourself in the foot' argument, and try to keep my setups as simple as possible. There's enough damn complexity in dealing with computers without you going out and voluntarily adding more on top, IMO 🙂

"Your text gave me the (possbily wrong) impression that you could do everything with efibootmgr, and with that, bypassing whatever downsides the motherboard UI has."

Yes, this is basically true – *as long as you make sure you can get into a UEFI OS to poke efibootmgr*, of course 🙂 If you take the time to learn efibootmgr, and you make sure you always have a handy way of running it, yes, it's a very handy get-out-of-jail-free card to have around.

"This leads me to a dreamy crazy question:…"

That's actually a rather interesting design. I don't know of anyone who's done it yet, but I don't know everything. Rod

(if he's reading? hi, Rod) may do.

People have certainly written things that are meant to sit at the UEFI bootloader level and intermediate between you and all this craziness. I have to admit that I tend to view them as yet another layer of craziness ;), but some people prefer to take the approach of picking one and making it their primary interface to the whole shebang. Rod's Refind – http://www.rodsbooks.com/efi-bootloaders/refind.html – is one of these, and he has a general page on UEFI bootloaders at http://www.rodsbooks.com/efi-bootloaders/ .

PoMo February 15, 2014 at 10:40 am | Permalink | Reply

Thanks for the Rodbooks.com links. Unfortunately those bootmanagers/loaders (refind, gummyboot, ..) are, as you say, adding "yet another layer layer of craziness". In short, they seem to mostly leave available features aside and instead focus on what EFI requires an OS to conform to and make use of that. And they're adding some more requirements on top of that (like naming schemes, certain min kernel version, or even limitations to certain OSes).

Why don't those bootmanagers "simply" manage efi boot entries? (and present them in a nice way)

I can only assume that that wouldn't always work, but i'd love to hear/read from someone who knows. (like Rod)

I fear that as long as there's no bootmanager that works with the motherboard's efi boot entries (reading from it, possibly also with the ability to add/modify/delete/backup/restore entries), and that can boot all entries that can than be booted from the motherboard's efi bootmanager, OSes cannot rely on available features and by that will always bring along their own bootmanager/loader that will confuse/add complexity by only working with their OS and perhaps 2-3 others, and that will possibly also mess with what other OSes put in place.

If there was such a bootmanager (and it was free and opensource ;)), each OS would only have to care about adding a working EFI-entry for itself. Of course the OS-installer could (and should) also offer to optionally install that bootmanager during installation. (and not in a grub-way where they roll their own version with adjustements for their OS)

I know it's a lot of wishful thinking 😉 I'd love to read about that topic from someone who knows this stuff though 😉

Rod Smith February 15, 2014 at 7:17 pm | Permalink | Reply

A boot manager that presents a menu based on what's in the firmware's NVRAM is certainly do-able. I've toyed with the idea of adding such a mode of operation to rEFInd, but I haven't done so because I don't believe it would add anything to what rEFInd already does, and in many ways it would in fact be limiting. For instance, if you use rEFInd to boot Linux kernels directly, the approach you suggest would require using efibootmgr every time you add a kernel. Since no distribution does this, it would be an extra manual step for the user — in other words, it would DEGRADE functionality.

Another problem with this approach — and a flaw with EFI generally — is that placing boot loaders on the hard disk and information about them in NVRAM creates two points of failure in the process of launching boot loaders. This is a practical real-world problem — I've both heard of and personally encountered systems that "forget" their boot

loader entries on a regular basis. On such systems, the only practical solution is to use the fallback boot loader (EFI/BOOT/bootx64.efi), and if it were to rely on the NVRAM entries, it would fail miserably. Even systems that don't USUALLY forget their NVRAM entries can do so occasionally. One of my computers does so if a hard disk is temporarily unplugged, for instance. There can also be OS-driven bugs. Some versions of shim can create an ever-expanding boot list, for instance, and this would be a nightmare for a boot loader following your suggested design.

That's not to say that such an approach doesn't have its merits, and might not appeal to some people — it does have merits, like being configurable from any OS using the standard EFI mechanisms. IMHO, though, it's not preferable to the auto-scanning that rEFInd does, at least not in general. I'll consider adding it as an option to rEFInd; in fact, in writing this reply, I'm contemplating ways that the NVRAM-based data might be integrated with the auto-scanned data in ways that might be useful.

Some additional comments on your post, and on Adam's response to it:

* My own "main beef" with EFI is with the number of bugs found in the EFI world. The poor boot manager menus that some implementations present are an issue, and they represent a good argument for using rEFInd, gummiboot, GRUB, or some other boot manager. (Note that rEFInd and gummiboot are both add-on boot managers only, whereas GRUB does double duty as both a boot manager and a boot loader.) Other bugs, like EFIs that forget NVRAM entries, are less common but often much more painful than poor user interfaces.
* The type of boot manager you describe would NOT require rebooting to boot another OS; it could, like rEFInd, gummiboot, or even GRUB, boot the next boot loader directly. (Yes, GRUB can chainload to another EFI program.)
* IMO, relying on efibootmgr and the firmware's built-in boot manager is not a good approach to managing a dual-boot computer, except possibly in those rare cases when the built-in boot manager is really good. The built-in boot managers are usually extremely limited in what they can do. Most notably, most of them require hitting a key (which varies from one computer to another, no less!) at JUST the right time in the boot process to boot anything but the default OS. This sort of design has always struck me as brain-dead, and because of system-to-system differences, documenting it is a nightmare. Fedora's philosophy at one point was to rely on the built-in EFI boot manager for multi-booting, but if I'm not mistaken, the Fedora developers have come to their senses on this one.
* I understand the view of rEFInd or gummiboot as being "another layer of craziness," but I believe it's misplaced. Recall the point I made that GRUB is both a boot loader and a boot manager. Thus, when dual-booting Linux and Windows, the boot path for Linux is likely to be EFI->GRUB->Linux kernel; and when booting Windows, it will be either EFI->Windows boot loader->Windows kernel or EFI->GRUB->Windows boot loader->Windows kernel, depending on whether the installation relies on the EFI boot manager (bad choice) or GRUB (better choice) to select which OS to boot. The boot path with rEFInd or gummiboot will look EXACTLY THE SAME, except that you replace GRUB with rEFInd or gummiboot, and the details of how the Linux kernel launches are different. rEFInd CAN launch Linux via GRUB, but GRUB can also launch Linux via rEFInd. Thus, when analyzed fully and configured optimally, neither rEFInd nor gummiboot is "another layer of craziness"; they're both alternatives to GRUB, at least functionally. Yes, rEFInd and gummiboot are both boot managers but not boot loaders; but with a boot loader built into 3.3.0 and later kernels, you don't really need a separate boot loader program to launch Linux.
* You mentioned the minimum kernel version as a limitation of rEFInd and gummiboot. It's true that both rely on the EFI stub loader, which was added with the 3.3.0 kernel. Not many distributions ship with kernels older than that at this point, though. Thus, this is no longer really a practical concern.
* Compared to GRUB, file naming is NOT a limitation of rEFInd; both boot programs can be configured with manual boot stanzas, which can launch kernels and boot loaders with any filename. The difference is that rEFInd can auto-detect boot loaders (including Linux kernels) in certain locations and with certain filenames. This is an ADDITIONAL FEATURE of rEFInd compared to GRUB. Thus, when comparing the two, it's GRUB that has the deficit, not rEFInd!

Really, my intent in designing rEFInd (or expanding rEFIt's OS-scanning features; it's only fair to give credit to Christoph Pfisterer, rEFIt's creator, for designing the basic framework) is the same as what you want, at a broad level: To launch any OS's boot loader with little or no extra configuration. rEFInd does this without relying on the

NVRAM entries because they're fallible, but rEFInd scans the locations where boot loaders are supposed to go, plus some other locations where they (especially Linux kernels) are commonly found in practice. People who use rEFInd tell me that it finds boot loaders and Linux kernels quite reliably; most of the problem reports I see relate to EFI bugs or difficulties installing rEFInd in exotic setups, not to the boot loader scanning features.

Steve Riley March 27, 2014 at 5:48 pm | Permalink | Reply

At Kubuntu Forums (where I'm an admin), I have documented my UEFI experiences over time. I have come to appreciate its superiority over BIOS. Initially, I had preferred booting the kernel directly via NVRAM variables until I discovered rEFInd. I concur with everything Rod writes about how rEFInd simplifies managing multi-boot. On one system I have set up rEFInd to boot Kubuntu, openSUSE, Arch, and Windows 8.

Actually, "set up" implies too much work — the only default I changed was to use text mode boot rather than graphical. Otherwise, rEFInd is a cinch because its autodetection is so good. I no longer need to mess with NVRAM variables because rEFInd presents a "just right" abstraction layer. For these reasons, I've been encouraging our members to ditch GRUB completely and go with rEFInd instead. (Yes, I am gushing. Oh well!)

Doug February 3, 2014 at 1:22 pm | Permalink | Reply

One thing I _don't_ understand: these FAT partitions:
are they little bitty ones that just hold UEFI code? They can't be system partitions, because
FAT doesn't support permissions. So what and where are they?

adamw February 3, 2014 at 3:11 pm | Permalink | Reply

"EFI system partition" is the name the spec uses to refer to them. They're not "system partitions" in the sense of Unix file permissions or something like that. The sense is more "the stuff here is just uninteresting bits necessary to make the system boot, not anything you're likely to be interested in".

The article explains what they're for: put very generically, it's a place where the OS layer can write stuff for the firmware layer to read. The obvious thing that falls into that category is bootloader code, and indeed that is mostly what is put in ESPs. There are other potential uses for it, but the main thing is bootloaders. As the article (I hope) explains, the role of the ESP is essentially to do the job of:

* the MBR
* the empty space between the MBR/partition table and the first partition

in a BIOS/MBR boot. In a BIOS/MBR boot, the bootloader is located in those spaces. In a UEFI boot, it's located on the EFI system partition.

The spec explicitly states that you can have as many EFI system partitions as you like on whatever disks you like in whatever locations you like on those disks. There is a specific GPT partition type that identifies them as ESPs.

BTW, as a Fun Educational Side Note, this isn't specific to UEFI – lots of other things do this, in fact it seems like every boot method invented since the BIOS/MBR style has done something like this. If you were to dig a PowerPC system out of your local junk heap and install Fedora on it, you'd find you needed either a 'PReP Boot Partition' or an 'Apple Bootstrap Partition'. ARM systems that use uboot need a 'U-Boot Partition'.

Rod Smith February 15, 2014 at 8:05 pm | Permalink | Reply

At least some EFIs give the option to read firmware updates from the ESP, so that's another example of what can go there. (Apple's EFI does this, as does the firmware in my ASUS motherboard.)

It's possible to place EFI drivers on the ESP so that the firmware can read them. At the moment, this is probably most common in conjunction with a rEFInd installation; rEFInd comes with several EFI filesystem drivers so that rEFInd can read Linux kernels from the Linux /boot directory. In principle, though, you could put a driver for a plug-in Ethernet card, a video card, or whatever on the ESP so as to give the firmware the ability to use that hardware, even if the hardware lacks EFI-compatible firmware itself. A few such drivers do exist, but they're pretty rare.

Some Linux boot loaders, such as ELILO, require that the Linux kernel reside on an EFI-readable partition, which in practice makes the ESP the easiest location. Along those lines, in some cases it makes sense to mount the ESP at /boot in Linux, so that the kernel will go in the ESP's root. Some distributions don't like this because they rely on symbolic links or other Unix filesystem features in /boot, but this approach is quite popular with Arch Linux users. The freedesktop.org boot loader proposal (http://www.freedesktop.org/wiki/Specifications/BootLoaderSpec/) would encourage, but not require, mounting the ESP at /boot.

SteveSi February 7, 2014 at 5:30 am | Permalink | Reply

Hi
Very nice article. I am unclear about 'Removable devices' and 'Removable Media'.
Most USB hard disks are of the 'Fixed disk' type – the SCSI Removable Media Bit (RMB) is 0.
Most USB Flash drives are of the 'Removable' type – the SCSI RMB is 1.
Some USB Flash drives (e,g, WinToGo Certified drives) have RMB=0.

What does the UEFI BIOS consider to be a 'Removable' drive – does it look at what the drive reports in the RMB – or does it merely assume any mass storage device on an external interface (eSATA, USB, etc.) is a 'Removable' drive?

adamw February 7, 2014 at 9:29 am | Permalink | Reply

That's an interesting question, and I don't have the answer on hand.

Looking through the spec, I don't believe it ever defines what should be considered 'removable' or 'non-removable'. The sensible thing for a firmware to do would of course be to respect the RMB. My opinions on how commonly firmwares do sensible things, are, I believe, on the public record 😉

It does explicitly list "Hard Drive" in section 12.3.4 – "This section describes how booting from different types of removable media is handled." – so it clearly considers the case of a 'removable hard drive', but it does not quantify precisely what it means by that.

Of course, the distinction isn't actually incredibly important; as I read it, what it basically boils down to is that when everything in the boot list is invalid and the firmware's doing fallback path processing it is required to check removable media before fixed media, and removable media are required in the spec to have only a single EFI system partition.

SteveSi February 7, 2014 at 9:37 am | Permalink | Reply

I suspect the BIOS interrogates devices on external interfaces first. Where the spec. says 'removable devices' it should really say 'external devices'???
It should be simple enough to test, but of course, it will depend on how different BIOS vendors interpret the spec.!

adamw February 7, 2014 at 9:38 am | Permalink | Reply

Yeah, as the spec doesn't clearly state it, you can't assume all firmwares will do the same thing. And you don't mean "the BIOS". It is not a BIOS. 🙂

mossywell March 13, 2014 at 5:58 am | Permalink | Reply

You say that "In the BIOS world, absolutely all forms of multi-booting are handled above the firmware layer". Mikhail Ranish actually wrote a multi-boot loader that fit entirely within the MBR code section (Cylinder 0, Head 0, Sector 1) in the late 1990s. I'm not referring to the XOSL implementation and other similar ones that booted code in the partition boot sector, this work or art fit entirely into the MBR. The "GUI" was limited of course, but even to this day, it is a work of poetry, not IT. 🙂

demon April 13, 2014 at 10:22 pm | Permalink | Reply

One thing that's worth mentioning re: UEFI vs. BIOS booting is the fact that, with a BIOS boot, you're heavily constrained by the fact that the CPU is starting in 16-bit real mode. The same way 8088- and 8086-based IBM PCs booted over 30 years ago, and it hasn't changed. That's a real pain in the ass for loading things like the Linux kernel and initrd images into extended RAM (or as it's called on a real OS, "RAM"). You have to dick around with poking things through the A20 address gate using the keyboard controller, effectively using it to "ship-in-a-bottle" load the kernel and initrd bit by bit into extended RAM, then either jump into the kernel or flip into protected mode (or long mode on an x86_64 processor) and then jump into the kernel. UEFI, however, is running in protected mode (or long mode), so you're not working with your hands tied behind your back.

Otherwise, good and thorough information, thanks!

Dean October 10, 2014 at 6:34 pm | Permalink | Reply

There is a UEFI BIOS practically. For compatibility reasons(some OSes don't support UEFI), recent PC firmware always support both legacy BIOS and UEFI. A UEFI BIOS normally refers to a UEFI-compatible BIOS, or a mixture of UEFI and legacy BIOS.

Steve Riley October 10, 2014 at 7:15 pm | Permalink | Reply

No. You have UEFI _or_ BIOS. If you have UEFI, then you also have a BIOS compatibility mode, in which the UEFI emulates BIOS. “UEFI-compatible BIOS” means nothing. The only “mixture” I’m aware of is that goofy “Hybrid EFI” firmware that Gigabyte sold for a while. It was a nightmare, as Rod explains: http://www.rodsbooks.com/gb-hybrid-efi/

adamw October 11, 2014 at 2:09 am | Permalink | Reply

Dean: Steve is correct, except that some recent systems are now being sold with UEFI firmwares that have no CSM (BIOS compatibility mode).

fisher October 18, 2014 at 11:21 pm | Permalink | Reply

Hi Adam, I have a questuion . If a USB flash drive or an optical disc are not GPT partitioned , but it does have the file EFIBOOTBOOT{machine type short-name}.EFI on it , Can it boot in UEFI-native mode? can uefi detect the bootloader automatically?
Thank you.

adamw October 19, 2014 at 8:40 am | Permalink | Reply

The answer to that is ‘possibly’ =) There’s an MBR partition type for EFI system partitions, and strictly according to the spec, firmwares are supposed to respect it. But it’s not something we think has been very widely used in The Real World and I haven’t tested it for real myself. But yeah, in theory if a disk has a partition with the MBR ESP partition type and it’s correctly laid out, the firmware should be capable of doing a UEFI native boot from it, AIUI.

Honiix August 6, 2015 at 8:46 am | Permalink | Reply

Some tools that create a installation-media for native UEFI system, like Rufus, don’t initialize the USB-stick as GPT but MBR !?! O_o

I’ve tried to create a GPT USB stick with an ESP partition but I failed. On Windows, Diskpart crashes if you try to do so.

I haven’t tested with Linux yet. Because the Rufus’s way works on every laptop and desktop I’ve inserted the MBR-USB-stick into.

Also, I spend alot of time googling for a guide to create a USB in GPT with a ESP partition and found nothing.

Why is that?

My guess is, ESP partition is optional on removable media !?

SteveSi October 19, 2014 at 8:51 am | Permalink | Reply

Any UEFI system should be able to boot from a plain FAT16 or FAT32 MBR Primary partition. That is in the spec. For instance, make a FAT32 USB Drive and extract the files from a Win8 x64 ISO to it and it will UEFI boot.
Easy2Boot uses this feature to boot multiple UEFI payloads from one multiboot USB drive.

The partition should not need to be marked as Active (though some UEFI systems seem to require this!) and it should be the first Primary partition on the drive (though it may still boot on some systems if it is not the 1st partition).

Max October 26, 2014 at 7:17 pm | Permalink | Reply

"If you have a UEFI-based system whose firmware has the BIOS compatibility feature, and you decide to use it, and you apply this decision consistently, then as far as booting is concerned, you can pretend your system is BIOS-based, and just do everything the way you did with BIOS-style booting. If you're going to do this, though, just make sure you do apply it consistently. I really can't recommend strongly enough that you do not attempt to mix UEFI-native and BIOS-compatible booting of permanently-installed operating systems on the same computer, and especially not on the same disk. It is a terrible terrible idea and will cause you heartache and pain. If you decide to do it, don't come crying to me."
I'm metaphorically crying … Hello Adam, I've done a few things of the previous lines, and my computer is behaving kind of weird. I installed Debian as LVM and on BIOS mode with a GPT formatted disk, and It got really slow. After that, I deleted the entire disk and tried to install it again but in UEFI mode. However, when I turned on the UEFI mode, the UEFI boot manager didn't recognize my disk, only the CDDVD (It does recognize the Debian installation cd but does not initialize it) and USB's units. I thought it was because I didn't have an ESP, so I installed it and didn't work. I gave up and changed it to the BIOS mode , and magically, my CDDVD unit started working, but since I have read the UEFI mode advantages, I would like to know how I can fix the apparent problem. For the record, I have an USB with a UEFI Shell which I tried to use through STARTUP.nsh but the script doesn't start and makes me think I don't have the latest version. Also, I have the gdisk tool in a Hiren's CD.
Thanks in advanced for reading and your possible help.

Stinger October 27, 2014 at 4:41 am | Permalink | Reply

Hi Adam
Thanks for a great 'essay' 🙂
It's hard to come by relevant info regarding (U)EFI booting , so I'm glad I found this page. Most of what I have seen on fora's and websites are, as you so nicely put it, half baked truths, propaganda or downright lies.

Many (I don't know how many) potential new Linux users hit a brick wall when they first try a Linux distro because their pc either can't boot the install/live media of the distro or, even worse, if they get it to install they cant see or boot the installed system.
As I see it, this can be caused by two things ?
1. The distro has poor support for UEFI firmware.
2. The pc has a poorly designed UEFI firmware.

Hypothetical, to get as many as possible to enjoy their first encounter with a distro, you would have to have good support

for UEFI in general and beyond that, you would have to do your best to deal with poorly designed UEFI firmware.
All this should, in my opinion be part of the live/installer media and incorporated in the installation routine, preferably dealt with automatically.

With my limited knowledge, the first obstacle is booting the live/installer media on a UEFI system but many distros already have dealt with that so I guess that's not the biggest problem.
As I see it, the biggest problem is that the installation routine of many distros doesn't deal gracefully with installing on a UEFI pc, leaving the user to try and deal with it themselves afterwards (even worse if its a multiboot scenario) .

Now to my question:
What would in your opinion be the best way to get a Linux distro installed on a UEFI system ?
Is it as Rod Smith suggests to use 'the kernel's EFI stub loader (in conjunction with rEFInd or gummiboot, if necessary)' ?
Your opinion would be highly appreciated 🙂

Anon March 8, 2015 at 5:24 am | Permalink | Reply

> If you absolutely insist on having more than one OS per disk, understand everything written on this page, understand that you are making your life much more painful than it needs to be, lay in good stocks of painkillers and gin, and don't go yelling at your OS vendor, whatever breaks.

This seems to imply that UEFI is doing a pretty damn bad job, if something as simple as having a dual boot has to be a massive pain and recommanded-against.

adamw March 8, 2015 at 2:44 pm | Permalink | Reply

Oh, that's intended to apply equally to BIOS. Dual boot isn't really simple and never has been, it's an excellent source of unnecessary annoyances in my book.

Richard April 2, 2015 at 4:55 am | Permalink | Reply

Very well Written. And Clear.

And after reading, I can safely say…. What the hell is INTEL doing! I can understand the desperate need to replace the aging BIOS with a more robust and functional TIER system, but this was FAR from what i was expecting to happen.

Ill share my point, 15 years ago i stubled across "Terabyte Unlimiteds BootIT Next Generation" software. used for boot management. It fits on a 1440kb floppy disk. It installs using the MBR a hugely improved MBR loader, affectionatly named EMBR. Back then it was was EMBR 1.0. its since been improved.

EMBR allowed the Boot Manager (Part of BOOTIT)

Boot any partition from any drive, anywhere. (including removable devices)
Have near UNLIMITED partitions of any kind on near UNLIMIED attached drives.
TRUE hide drive(s) and partition(s) from the Operating System you choose to boot.

And best of all it was Completely Configurable with SECURITY with a SIMPLE and CLEAN LOOKING Bootloader Menu, intergrated PARTITION AND DRIVE MANAGER able to size, resize, Move, copy, create, delete partitions of ANY kind with SPEED and 100% reliablity.

And all this fit on 1 floppy disk which could have been made in to a firmware BIOS.

I still use it today on BLADE SERVERS !! as well as home PC's

Intel should have looked around FIRST before trying to build a better BIOS, because others out there had the STANDARD in place ALREADY.

Im sticking with EMBR, it works without complication, and its a hell of a lot easier to understand.

Richard April 2, 2015 at 5:03 am | Permalink | Reply

I neglected to add, EMBR also blows away the drive space limits i.e 2TB, It was past that issue 9 years ago

SteveSi April 6, 2015 at 11:30 am | Permalink | Reply

UEFI is needed because the old BIOS+MBR spec only copes with disks of <2TB. Also it caters for different machine/CPU architectures and UEFI can be expanded in the future.
If your BIOS will not MBR\CSM boot, then that is the fault of the BIOS/system manufacturer.
Many non-UEFI (MBR) systems are still being produced today that cannot boot from a USB drive that has the active partition past 137GB – this is due to poor coding of the BIOS USB driver and the 'feature' has been around for >10 years.
Some MBR BIOSes even completely ignore the MBR boot code and just execute the PBR code instead – totally against the original IBM PC spec!
You could now blame all this on the UEFI-spec being too vague – i.e. it would have been an *ideal opportunity* to clearly specify both UEFI and MBR booting requirements once and for all and then UEFI\CSM BIOS writers would have a clear spec. to follow. Instead they just copped out by saying 'we will not define the CSM compatibility functions!'
There is no offical MBR-booting spec. (it just sort of grew with all sorts of geometry translation schemes and bugs for 8GB/137GB limits, etc.!) and AFAIK, there was no developers guide produced on how to code a UEFI-BIOS with examples in pseudo-code, function descriptions, etc. plus what mandatory interfaces to provide to the user. Big mistake!

Neill Rutherford July 31, 2015 at 5:57 am | Permalink | Reply

I do not have a UEFI firmware computer. I have an HP G61 64b machine all the rest are desktops and 32b.

So my question is regarding using a USB bootable drive in various machines. As I don't have the new hardware I can not examine the how do i do this myself.

For the old legacy BIOS boot firmware machines there is little problem. providing one can set boot order and it accepts booting from a USB port otherwise booting from an optical disk via chain-loading might be possible (yeah i have machines that old)

For UEFI firmware machines

1) if the machine in question does not have an internal hard drive.

2) if the machine does have an internal hard drive but would prefer that the internal drive is not accessed in any way.

Ideally I would want a bootable USB that would work in any machine. I think this should be possible by booting via MBR on BIOS machines and by booting via the UEFI partition on UEFI machines.

What are you thoughts

Sorry if my terminology is not correct.

Neill

adamw July 31, 2015 at 9:46 am | Permalink | Reply

Well, sure, distributions have already had to deal with this. mjg59 wrote up how we keep the Fedora images bootable on multiple platforms a few years back: https://mjg59.dreamwidth.org/11285.html

There's a lot of detail, but it boils down to giving the media both MBR and GPT partition tables and including a bootloader in the EFI fallback path location for whatever arch it should boot on.

SteveSi July 31, 2015 at 6:07 am | Permalink | Reply

First, if you use VirtualBox+VMUB you can boot from a USB drive using UEFI in the Virtual Machine. So you can easily test UEFI booting. See http://www.rmprepusb.com – Tutorial #4.
Next, if the USB drive is FAT32, then you can make it MBR-bootable (normal BIOS) and by placing the .efi boot files on it, UEFI-bootable. e.g. format a USB drive as FAT32, add a bootmgr boot sector (e.g. using RMPrepUSB or bootsect or just format it using Windows – ensure the parition is marked as Active) and then simply copy the contents of a Windows 8.1 64-bit DVD to it.
To boot in any machine, you would also need to add 32-bit UEFI boot files.
However, there is a fly in the ointment! Many UEFI systems have a 'bug' – if they 'see' a USB drive which has both MBR and EFI files on it, they will NOT allow you to MBR-boot from it (only the UEFI-boot option is offered). What this means is that, on these systems, you can never MBR-boot from the USB drive! To MBR-boot, you would have to delete the EFI boot files first – and then it would not UEFI-boot!

Neill Rutherford July 31, 2015 at 1:45 pm | Permalink | Reply

Hello

Thank you both for your replies.

Just so that you understand where I am coming from I want to create a USB disk or USB disks (One for root / and one for /home)

Then in an unknown computer (x86, X86-64 and maybe Mac) which has no internal hard disk I want the USB disk to boot.

To cope with low spec machines I would choose Lubuntu 14.04 LTS 32b. The USB will contain an installed working operating system not a liveUSB.

Do you see any problems with what I am trying to attempt?

Jenel August 8, 2015 at 7:11 am | Permalink | Reply

I can't get around this: "With a BIOS firmware, your firmware level 'boot menu' is, necessarily, the disks connected to the system at boot time". What is the boot menu, then? Is it "press F10 to choose boot device", or is it lilo / grub / "press F8 for safe boot"?

adamw August 8, 2015 at 8:36 am | Permalink | Reply

"Press F10 to choose boot device". I'm just saying that since all BIOS knows how to do is 'run the boot sector on a disk', the only boot options you have at the BIOS level are 'this disk or that disk'. With UEFI that's not the case, since it's more capable.

Seppe Sol August 24, 2015 at 5:04 pm | Permalink | Reply

The firmware (InsydeH2O) on my Packard Bell easynote TE laptop has no support for Bios compatibility, thus: no VESA, VB2, OpenCL etc.

The UEFI protocols only reveal one graphic and one text resolution for one screen, a primitive blt protocol without clipping feature, and a framebuffer in PCI space.

Yet, windows 8 perfectly knows all resolutions of all the connected graphic devices.
It even knows about the external screen and it's possibilities.

I assume that MS (directx?) uses window device drivers to handle graphic devices without Bios support.

How can UEFI developers obtain similar functionality without Bios support?

Note: Microsoft's bootx64.efi takes 1.5Mb.

Thank you
Seppe

Sanjit Keskar September 19, 2015 at 11:23 pm | Permalink | Reply

Thanks AdamW. This blog article was TREMENOUSLY useful and enlightening for me

But may I ask you for a bottom line statement on a query of mine

I have an acer switch 10 e laptop which has ONLY a UEFI boot (legacy boot not supported and no CSM). Yes it has a Secure boot option BUT THE UEFI BBOT ARCHITECTURE IS 32 BIT ONLY

I cannot boot a multitude of boot disks that claim to be UEFI bootable (for imaging programs, disk management programs for formatting and disk resizing and data recovery. They of course do boot on my other UEFI boot laptop. This is probably

because my uefi is 32 bit only.

SO WHATS THE WORKAROUND (IF ANY)?

Does the responsibility of booting a bootable disk on 32 bit uefi architecture lie with the software company ? i.e they have to write code for the same? say for example acronis

Is it possible to modify the iso image (if it is not illegal) by introducing some ready to use files to get such cds to boot on a 32 bit uefi device?

What is the bootia32.efi? and how is it relevant to my problem?

Thanks

adamw September 19, 2015 at 11:41 pm | Permalink | Reply

Hi, Sanjit, thanks for the kind words.

32-bit UEFI firmwares are oddballs. Up until the last couple of years, the only ones that existed were very early Mac firmwares, which pretty much no-one cared about supporting.

The ones that have shown up recently are mainly tablet/convertible systems based on the Intel 'Baytrail' platform, like yours. These have 64-bit CPUs but shipped with 32-bit firmwares due to some kind of Microsoft fail, the details of which I've forgotten.

It is, indeed, up to OS vendors to make images compatible with such systems – if they actually care to. There are two ways to do this:

- Build 32-bit UEFI bootable images
- Build 64-bit images capable of booting on 32-bit UEFI firmwares (using a bootloader capable of booting a 64-bit OS from a 32-bit firmware)

I actually built a 32-bit UEFI remix of Fedora called Fedlet, because I also have one of these systems and wanted to play with getting Fedora running on it. I have not had much time to maintain it lately, though.

Distributions are generally not very interested in building 32-bit UEFI bootable images. We usually keep 32-bit images around for *old* hardware, and some old hardware has trouble booting images that are set up for UEFI as well as BIOS, and no-one really wants to go to the trouble of building *two sets* of 32-bit images just to cater to some oddball systems that Microsoft screwed up.

Distros are more open in principle to the idea of making their 64-bit images bootable on these oddball 32-bit UEFI firmwares, but someone has to take the time to do the work, and for Fedora none of the few of us who are at all interested in the topic have been interested *enough* yet to do it. Matthew Garrett and Peter Jones did great work to make grub capable of the 64-on-32 trick, but all the distro infra around that still needs to be set up and tested, and it's just a lot of work for comparatively little return.

If you're really invested in making this work you can try running the latest Fedlet release on your system, but I make no guarantees about it, and don't have much time to update it. There are also folks maintaining hacks for other

distros, I haven't kept up with them lately though. For stuff like Acronis, you're basically at the mercy of the company, and I doubt they care enough about the fairly small number of those systems out there to take the trouble.

bootia32.efi is the 'fallback path' bootloader name for 32-bit Intel. As described in the article, the 'fallback path' basically works by defining standard locations where the firmware can look for a bootloader; the spec defines filenames for various arches, because obviously if you want to make a disk bootable via the 'fallback path' on multiple arches, you have to be able to provide an appropriate executable for each arch, and the firmware has to be able to find the right one. So the name for 32-bit Intel is bootia32.efi. The name for 64-bit Intel is bootx64.efi, IIRC.

(Amusingly, I think we've found both firmwares that require the name to be upper-case – BOOTX64.EFI or BOOTIA32.EFI – and firmwares that require it to be lower-case – bootx64.efi or bootia32.efi . Sigh.)

Sanjit Keskar September 20, 2015 at 2:49 am | Permalink | Reply

OK Got that (and more than I bargained for)!
I was always under the impression that if I saw the bootia32.efi file in the boot folder, that such bootable media will always boot on a 32 bit uefi firmware.

Please do address the following doubts I have

So the ball is in he software owner's court to add compatibility for the bootable media to boot on a 32 bit UEFI firmware

How would FEDLET help me in my particular predicament? i.e. the need to run bootable media on a 32bit uefi firmware?

Is there anyway to inspect the iso of an uefi bootable media and know that it will o will not boot on m 32bit uefi baytrail laptop

Can you confirm that this limitation in booting will not be applicable in the following case. I windows install a disk management software on my baytrail laptop. The software allows me to change the size of the system partition. The laptop has to reboot to do the same and perform this task from outside windows so to speak. May I assume that the (re)booting will not be a problem and task will be performed.

Thanks again adamW

Christopher Price October 5, 2015 at 2:37 am | Permalink | Reply

Regarding Secure Boot:

"If you don't like this, don't buy one. But also don't buy an iDevice, or an Android device with a locked bootloader "

Secure Boot is disliked because it gave legitimacy to the practices of Apple and a handful of Android device makers. Before that, Microsoft had to sign a settlement agreement for its similar practices against Be, and others in the 1990s.

Since you wrote this article, Microsoft has given OEMs the ability to revoke the ability to disable Secure Boot on Windows

10 systems. This can make it impossible to tell which machines, in a retail setting, for an average consumer, can/can't run Linux. Average consumers aren't going to spend hours researching each PC to know its Linux viability. And worse, they may learn after the fact (once interested in Linux) that they "bought the wrong one."

Now the entire PC market is at risk of slowly being locked to the OS that shipped with it.

You can argue that slippery slope is counterbalanced by a rise of IoT devices, and Android becoming the most popular OS today. Others will argue that it has ensured Linux on the desktop will be owned by Google. Either way, the critics of Secure Boot do have a sound argument… one time has proven.

The UEFI specification could be modified to require a user setting to disable it. And OS vendors could require Secure Boot to boot their kernel. That would mean Windows 10 in a datacenter could remain secure, while home PCs remain open. The question is, if the UEFI group has the guts to stay open… or not.

Computer Scientist August 20, 2018 at 1:13 pm | Permalink | Reply

Any computer that is Windows certified is required (by virtue of that certification) to *not* being locked into a specific OS, or even being locked into the Secure Boot keys from Microsoft. So no, that's a completely misleading and wrong point to make.

Christopher Price August 20, 2018 at 1:33 pm | Permalink | Reply

My comment was written three years ago, when there was serious concern (with valid reasons) that the Other OS key would be pulled from Windows to enforce secure boot.

That didn't come to pass, as most of the industry that wanted that control has switched to Windows on ARM.

Still, it is a valid thing to be concerned about with Windows certification – it all hinges on being able to bypass or use an Other OS key that many feel could be revoked at the next security FUD event. I don't think it will happen today – but it's always possible, and the FOSS community should stay vigilant toward.

pgmer6809 October 27, 2015 at 4:39 pm | Permalink | Reply

Can anyone explain why after all this time the Linux foundation, (or Red Hat, or …) have not come up with their own key and lobbied the major Mobo makers and PC vendors to include this key alongside the Microsoft key?
That way there would be at least one other entity besides MS that could 'sign' bootloaders.
(Very well written article BTW)

adamw October 27, 2015 at 5:00 pm | Permalink | Reply

I wasn't directly involved in any discussions about that, but what I understand second hand is that it's a fairly complex, expensive and thankless task that involves shouldering quite a chunk of implied legal liability. RH can't justify that to its shareholders, and the Linux Foundation probably literally doesn't have the resources.

pgmer6809 October 27, 2015 at 4:42 pm | Permalink | Reply

I don't think you approve of this configuration but I run one machine with a GPT disk that boots in BIOS compatibility mode.
I use it to test Linux distros. I have about 16 partitions on it (and counting) at the momment, one per distro. No ESP.
Grub has the ability to define a 'extra grub code partition' on GPT disks that make all the problems with 'magic space' go away.
Thanks to GRUB and OSprober I do not have a problem booting multiple OS's from this one GPT disk. And I can create as many partitions as I want without worrying about Logical, Extended etc. etc.

adamw October 27, 2015 at 4:59 pm | Permalink | Reply

BIOS on GPT *can* work, it's just not really a great idea unless you actually *need* GPT, i.e. you're working with a very large disk or the four primary partition limit is somehow important to you (though I really can't think of any situation where that limit would be a problem *and* whatever had the problem would be perfectly happy dealing with GPT).

We made GPT the default for BIOS installs for one Fedora release, and it was a train wreck; some of the issues were user error (people doing custom installs not understanding about the BIOS boot partition), but we also found that there are quite a few firmwares out there that simply won't work with it, they just will not boot.

Julian November 18, 2015 at 5:39 pm | Permalink | Reply

I have new PC, which I assembled myself: Asus motherboard, which during startup displays UEFI BIOS, 8 core AMD processor, 250 GB SSHD, extended partition on my 2TB SATA drive and two partitions drives – E and F. I run Win 7 pro 64 and everything was fine. I've did upgrade to Win 10 and everything was fine until Nov Win 10 update. After rebooting my drives E and F disappeared in fact there was no SATA drive at all in device manager ??? Why? Any light in the darkness wise guys?
I was able one time restore my machine to pre update state but Windows installed updates and again no HD. I had huge troubles to return to Win 7. For some magic power it finally happened and I run Win 7 again.
I am thankful for sharing your knowledge.

Michael C. July 26, 2016 at 9:54 am | Permalink | Reply

Julian you made me laugh (thanks); I too prefer Win 7 Pro; Short long story, my Toshiba crashed, after installing new board and HD, that too crashed after one month usage, techy friend advised it was an inherent problem of Toshiba units with AMD CPU (?); I purchased new unit with Win 10 installed (nobody wanted to sell me new unit with Win 7 of any kind)sooo, I purchased complete full blown Win 7 Pro install disks for new PC's, I yelled, bitched, almost threw the unit under my car until another friend said it's new PC with newer motherboard; I finally accessed the UEFI, disabled it and finally installed my Win 7 Pro; I'm still leery of the MS updates, but now I'm armed, just running out of knowledge.

Rogerio Hilbert December 22, 2015 at 7:01 am | Permalink | Reply

I was wondering about the beginning of the article in the terminology section when it says: “Please don’t ever say ‘UEFI BIOS’”. I’ve been seeing quite a lot that terminology everywhere, including the title of the UEFI Utility Interfaces, like below:

http://rog.asus.com/wp-content/uploads/2012/07/11.png

It doesn’t seem to be wrong, does it?

adamw February 18, 2016 at 2:58 pm | Permalink | Reply

It’s still wrong, but at this point I’m feeling distinctly Canute-like. I still maintain that UEFI and BIOS are types of PC firmware. Saying “UEFI BIOS” is like saying “Linux Windows” or something.

Pugly February 19, 2016 at 4:03 pm | Permalink | Reply

In 2015 i helped my father in law buy a laptop. We bought one under the premise i would nuke the pre-installed win8 and go with win7. Turned out the oem had removed the ability to turn off secure boot from the firmware. There was no way to tell this from either the internet or the packaging. The online help from the oem had screenshots showing how you could simply turn off secure boot from the firmware menu when infact this menu item infact no longer existed.
There should be a clear warning/notice in the product description if secure boot is forever on. It limits the device in a way that was not typical of pcs up till now.

Jacob Burroughs March 9, 2016 at 6:07 pm | Permalink | Reply

I enjoyed reading this article. Having read this article after spending several hours one night making secure boot, Windows, and Fedora (later switched to Ubuntu because reasons, but not really relevant to this comment) off a single laptop SSD, I will say that might life would have been a lot easier had I followed basically any of those last points. That said, where would be the fun in doing things the easy way? I therefore want to point out rEFInd, which I don’t know if you are familiar with. It offers a nice interface for UEFI multi-booting, and supports using Secure Booth through shim, and chain-loading another bootloader and/or detecting Linux kernels. Once setup once, it is actually nice and painless to use, but I spent way too long getting it set up. Of course, the second time I had to do it, it was much easier since I had done it once, but that doesn’t make it any more user-friendly.

Also, on that note, I particularly liked your line “Or you can read up on how to configure your own chain of trust and sign your kernels and kernel modules and leave Secure Boot turned on, which will make you feel like an ubergeek and be slightly more secure.” since that more or less reflects my feelings, even if I am currently not building my own kernel. (Is there a good reason I might someday want to experiment with doing that?)

HAAbackwater Bumpkin October 23, 2016 at 11:54 pm | Permalink | Reply

Thanks AdamW! I’m also a pedandict podunk and while this whole computer business is completely beyond my simplistic nature and etc , you’ve truly explained things in a way me and that other pleb could grasp 🙂

I have a couple other minor clarifications and I'd really appreciate it if you had another moment to offer some of your information.

1.) please tell me if I'm understanding correctly that the manufacturer has mislabeled the updated firmware as bios on a newer Dell Inspiron and whether it does me any good to attempt to "re flash" it again and again. I've done this many times and this blasted malware just returns.

2.) do I also hear you say that the MALICIOUS code is contained within the firmware and/or the media I'm try to use to fix the problems, and which would be the better scenario and more likely for a guy like me to actually pull off? I've heard it is possible to modify an ISO or image file but I hadn't considered that you might ever attempt to fix the firmware itself.

3) (last one) is this a situation that is better dealt with while an operating system is installed on a permanent disk or better handled with the disk removed using a live cd?

Please excuse me my backwards and ordinary ways and let me assure you that not in all my years of blindly meandering through YouTube tutorials that this is a wonderful explanation of how UEFI differs from bios and what it actually does. Maybe you should keep at this writing stuff especially because it's a real treat to read. 👌

adamw October 25, 2016 at 5:57 pm | Permalink | Reply

1) Yeah, it's not uncommon for vendors to refer to a UEFI firmware as 'a BIOS' or something like that. I am in a bit of a minority with my insistence that the colloquial usage of 'BIOS' to mean 'anything that's kind of a system firmware' is stupid and wrong, but I still think I'm right. 🙂

2) I'm not quite sure what you mean by 'malicious code' here. This post is a general explainer about how UEFI works, it's not about fixing some kind of malware issue. It sounds like you're dealing with some kind of malware issue but you didn't actually explain what it is, so I can't really answer your question, I'm afraid.

3) Seems like a ditto to #2.

Azix October 27, 2016 at 6:24 am | Permalink | Reply

So UEFI is a kind of BIOS?

adamw November 1, 2016 at 6:21 pm | Permalink | Reply

Fry meme

Vis November 2, 2016 at 1:28 pm | Permalink | Reply

Good read. Have one question. I understand every firmware could have their own implementation, but what is the expected or general behavior if there are no boot entries in NVRAM, but the ESP have kernel directories and bootloaders present.

I tried this experiment in an ubuntu system.

Created 2 directories inside the boot partition.

EFI/Ubuntu/grubx64.efi
EFI/fedora/grubx64.efi

Deleted the boot entries from NVRAM and did a power cycle.

When the system came back up, could see Ubuntu entries in efibootmgr -v and not fedora. Am curios to know how figured out one entry and not another. (There is no fedora entry in grub.cfg and only Ubuntu, but i believe that is immaterial to create the boot entry)

Thanks.

Vis November 2, 2016 at 2:02 pm | Permalink | Reply

I could get the firmware create the fedora entry, by having a shim.efi file inside \EFI\fedora. When i created another directory say vis (\EFI\vis\, with a bootloader file grubx64.efi and shimx64.efi, firmware doesnt create though. What file does firmware expect to create a boot entry.

adamw November 3, 2016 at 12:37 pm | Permalink | Reply

The only thing the UEFI spec *requires* the firmware to implement is the fallback path, which is basically the thing where there's a designated location where the firmware is supposed to look on the first ESP it finds on a disk. For an x86_64 system, the fallback path is \EFI\BOOT\BOOTx64.EFI , as explained above.

Now, what I'm betting happened in your case is this. Fedora has a trick where when we install, we place a bootloader in the fallback path location . If that bootloader gets loaded, it boots Fedora normally, *and re-creates the 'regular' Fedora EFI boot manager entry*. So if you install Fedora alone to a UEFI system, delete the Fedora boot menu entry, then reboot, you should see the system boot to Fedora, then on the next reboot, and 'efibootmgr' will show the 'Fedora' entry again.

I'm betting Ubuntu has implemented the same trick, and you installed Ubuntu after Fedora, so it was Ubuntu's fallback loader that wound up getting run, not Fedora's. So Ubuntu did the same thing I described, and recreated its boot manager entry.

BTW, there is an open question I'm supposed to be figuring out here (it's come up a few times recently), which is whether the UEFI spec actually requires the firmware to try fallback booting from attached hard disks / SSDs if no boot manager entries are present, or if it only *requires* the firmware to try fallback booting from removable devices. I still need to look that up again. But most firmwares *do* in fact try fallback boot from attached hard disks / SSDs if no other boot entry is present or works.

Vis November 4, 2016 at 12:15 am | Permalink | Reply

In my case, i am not quite sure, it is booting because of fall back path. I believe Firmware is

doing something different. Have shown the sequence below. Assuming that with the fallback bootloader being present, why is it not detecting my directory in ESP(say vis, which has the same contents) and detects only ubuntu and fedora boot loaders. Does firmware creates entries of known/registered vendor.

Below is the sequence:

Before reboot (I have deleted all boot entries using sudo efibootmgr -b # -B

/mnt/1/EFI$ sudo efibootmgr -v | grep -e Ubuntu -e Fedora -e vis -e Vis
/mnt/1/EFI$

Below are my ESP files.

/mnt/1/EFI$ ls -ltr
total 12
drwxr-xr-x 2 root root 4096 Nov 2 20:37 ubuntu
drwxr-xr-x 2 root root 4096 Nov 2 20:54 fedora
drwxr-xr-x 2 root root 4096 Nov 2 21:36 vis

/mnt/1/EFI$ ls -ltr ubuntu/
total 0
-rwxr-xr-x 1 root root 0 Nov 2 20:37 grubx64.efi

/mnt/1/EFI$ ls -ltr fedora/
total 408
-rwxr-xr-x 1 root root 135680 Oct 14 02:48 bootx64.efi
-rwxr-xr-x 1 root root 135680 Nov 2 00:14 grubx64.efi
-rwxr-xr-x 1 root root 135680 Nov 2 20:54 shim.efi

/mnt/1/EFI$ ls -ltr vis/
total 408
-rwxr-xr-x 1 root root 135680 Nov 2 00:15 shimx64.efi
-rwxr-xr-x 1 root root 135680 Nov 2 07:23 grubx64.efi
-rwxr-xr-x 1 root root 135680 Nov 2 21:36 shim.efi

sudo reboot

```
(/dev/pts/0) at 7:00 ...
```

The system is going down for reboot NOW!

Once the system is back up, i see boot entries for Ubuntu and Fedora and not for vis. (I have 3 boot drives. One we are looking at is 965970eb-d8d1-4d73-aa1e-d0c7bad6f948 (/dev/sda1)

/mnt/1/EFI$ ls -ltr /dev/disk/by-partuuid/965970eb-d8d1-4d73-aa1e-d0c7bad6f948
lrwxrwxrwx 1 root root 10 Nov 4 07:02 /dev/disk/by-partuuid/965970eb-d8d1-4d73-aa1e-d0c7bad6f948 -> ../../sda1

/mnt/1/EFI$ sudo efibootmgr -v | grep -e Ubuntu -e Fedora -e vis -e Vis

Boot0010* Fedora HD(1,GPT,965970eb-d8d1-4d73-aa1e-d0c7bad6f948,0x100,0x1ff00)/File(\EFI\FEDORA\SHIM.EFI)..BO
Boot0011* Ubuntu HD(1,GPT,965970eb-d8d1-4d73-aa1e-d0c7bad6f948,0x100,0x1ff00)/File(\EFI\UBUNTU\GRUBX64.EFI)..BO
Boot0012* Ubuntu HD(1,GPT,efd05c7c-c77b-475f-876e-12a585bf9a15,0x100,0x1ff00)/File(\EFI\Ubuntu\grubx64.efi)..BO
Boot0013* Ubuntu HD(1,GPT,caafb2f8-a788-4210-988b-f7cb8ab46954,0x100,0x1ff00)/File(\EFI\Ubuntu\grubx64.efi)..BO

/mnt/1/EFI$ ls -ltr
total 12
drwxr-xr-x 2 root root 4096 Nov 2 20:37 ubuntu
drwxr-xr-x 2 root root 4096 Nov 2 20:54 fedora
drwxr-xr-x 2 root root 4096 Nov 2 21:36 vis

There is no BOOT under EFI.

Curious to see the above behavior.

Irvine Specter November 11, 2016 at 6:55 pm | Permalink | Reply

As a beneficiary of your graphomania, thanks for your work. I hope to have enough time to read all the useful things you write!

Gary Gapinski November 11, 2016 at 8:09 pm | Permalink | Reply

Nicely written.

I learned recommendation #5 the hard way after much puzzlement when trying to get a VirtualBox kernel module loaded with Secure Boot enabled.

Oyvind Overby November 13, 2016 at 2:29 am | Permalink | Reply

It would be nice to know why one should consider changing from the old BIOS/MBR thing to UEFI, assuming the scenario where I already have Win 10 installed on my PC the old fasion BIOS/MBR way. What is there to benefit? Boot Speed? Overall increased average HDD speed? Better protection against, or ability to recover from bad sectors or other intermittent HDD defects?
Less HDD space wasted due to more felxible block sizes (in case of zillions of small files)? Improved read-ahead? Delayed write? An security improvents (apart from secure-boot)?

Nobody should have to read 40 pages of tech stuff and still wonder; whatever for? Why? What is the benefit?

Ravi November 13, 2016 at 10:58 pm | Permalink | Reply

If its working for you no need to change. You never had to read how BIOS boot works since its an old method with lot of tools, troubleshooters and tutorials to help you. But its still a hack. 512 bytes is too little of a space to work. You still need “magic” to load your bootloader, which loads the kernel.

Read the article again. UEFI takes a break and provides a clean implementation which can be replicated. Kernels can boot directly using it.

Daniel Scott February 2, 2019 at 4:41 pm | Permalink | Reply

Oyvind Overby: One of the chief “practical” benefits of UEFI is that it will allow you to boot an operating system that is installed on a partition larger than 2TB. BIOS/MBR cannot do this.

adamw February 6, 2019 at 10:30 am | Permalink | Reply

Well, that’s a benefit of GPT, strictly speaking. BIOS + GPT can do it (with a special partition, and if the BIOS in question doesn’t choke on GPT). Anaconda supports doing this.

Overmind November 15, 2016 at 4:10 am | Permalink | Reply

I disagree with the statement that ‘MBR itself is not big enough’. it does not have to be. The bootloader is not and should not act like an operating system. The bootloader is a simple small file that if needed can just load a bigger file. No problem here whatsoever.

Ajay June 1, 2017 at 11:58 am | Permalink | Reply

Well, when we say BIOS, we refer a program that starts the computer system. It is of course stored as firmware. It initializes hardware and invokes booting process. UEFI does the same what traditional BIOS programs (distributed by Award, American Megatrends etc.) have been doing. So, Computers keep evolving. After 8 bit 16 bit machines we have have 32 bit and 64 bit machines. It is still computer. Saying UEFI is not a BIOS program is like saying like 16-bit machines were not computers.

Computer Scientist August 20, 2018 at 1:01 pm | Permalink | Reply

You still misunderstand the term BIOS, which makes your analogy flawed. BIOS is a specific type of firmware, BIOS is not a generic way to refer to a program that starts the commuter. You may be using it that way, but you would be wrong to do so. Firmware is the program that starts the computer, BIOS is a specific implementation of that. To call every program that starts a computer a BIOS is just plain wrong, and shows limited experience with computers beyond your average IBM PC compatible machine.

Shaun Clarke September 26, 2017 at 5:11 am | Permalink | Reply

Just wanted to thank the author for the write up. Very informative and I certainly learned a lot, even though I'm a Windows guy :).

One thing that has had me curious, and actually led to me researching UEFI in more detail is why UEFI uses a FAT (compatible) file system, as FAT is a proprietary file system by Microsoft. Do UEFI implementations have to pay royalties as a result?

SteveSi September 26, 2017 at 5:38 am | Permalink | Reply

http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc
Microsoft granted rights specifically for UEFI.

Chas B September 6, 2018 at 9:09 am | Permalink | Reply

How does the UEFI firmware find the ESP partition location on disk ?
Where does it get the disk partition information ?
Apologies if you mentioned this already.

adamw September 6, 2018 at 11:05 am | Permalink | Reply

Simple – it reads the partition table, like anything else would. 🙂

Chas B September 6, 2018 at 2:37 pm | Permalink | Reply

So how does the UEFI firmware know where the partition table is ?
I know that for legacy BIOS it is always in the first disk sector (MBR)
And also does it look for MBR partition table and GPT partition table ?

adamw September 6, 2018 at 2:53 pm | Permalink | Reply

MBR and GPT both live in defined positions on the disk. The primary GPT from LBA 1 to LBA 33, the secondary GPT (basically a backup) at the end of the disk, in the last 33 blocks. See https://en.wikipedia.org/wiki/GUID_Partition_Table .

UEFI firmwares *can* read both MBR and GPT, usually, and many BIOS firmwares can read and boot from GPT. Typically, though, UEFI is and should be used with GPT. The Fedora installer actually refuses to let you do a UEFI install to an MBR-labelled disk, we just don't want to support that.

Night Eagle November 8, 2018 at 7:19 pm | Permalink | Reply

Wow, my head is spinning. I have learned a great deal reading all this. Unfortunately not enough to understand why on my HP laptop I can't boot from my DVD/cd in anything but "Legacy" mode thus when I do I cant use the UEFI to

reinstall windows. When I boot from the DVD in UEFI the install disk cant be read. I think my computer is haunted.

Night Eagle November 8, 2018 at 7:38 pm | Permalink | Reply

It has to do with my install disk, doesn't it?

adamw November 8, 2018 at 10:52 pm | Permalink | Reply

It's possible, yes. The disc you're trying to boot could simply not be set up to be UEFI bootable at all. An image can legacy bootable, UEFI bootable, or both.

Porfy May 6, 2019 at 8:51 am | Permalink | Reply

I'm new to UEFI. I've done many dual-boot installations, but all based on BIOS/MBR. I basically tried to do UEFI when I got a new laptop that is dual-disc. It came only with the HDD, and with my curiosity piqued, I bought the NVMe SSD then re-installed Windows using UEFI firmware, as this is supposed to bring out the speed and all intended for the NVMe SSD. Did I understand that correctly? At any rate, I managed to get this particular machine to do dual-boot.

Secondly, I have older laptops, like 3-4 years old, and some have the mSATA SSD slot. Some I am able to turn off Legacy support and install Windows under UEFI firmware. No problem with that. My question is on those machines that I am not able to turn off Legacy/CSM. I have tried and I am still able to install Windows using UEFI firmware (yes, my installer is UEFI-enabled created using Rufus). Is this okay, or am I headed in the wrong direction? I am asking as once I am done with Windows installation, I will be attempting to do dual-boot: install Ubuntu Linux (UEFI mode), and finally, Grub Customizer, which I found in previous installs is able to read and use the EFI boot manager.

I hope my short explanation is clear. Let me know please on my queries. Thank you!

adamw May 29, 2019 at 4:32 pm | Permalink | Reply

Hi Porfy!

I am not sure about this whole NVMe SSD thing, that's new to me. But to answer your main question: you don't need to turn CSM support off entirely to do a native UEFI boot, in theory. If the firmware is actually a UEFI firmware, there *ought* to be a way in there for you to do a UEFI native boot of your OS install medium. But because the firmware UIs aren't standardized at all and vary a lot between systems, that's about as specific as I can get. Good luck!

Paul Bonneau July 1, 2019 at 7:22 pm | Permalink | Reply

I am somewhat amused at the recommendations, e.g. "If you can possibly manage it, have one OS per computer." Here I was reading all about the wonderfulness of UEFI allowing me to flexibly and powerfully boot anything at all – and then the number one recommendation is to hobble yourself the same way you'd do with a BIOS setup, just to have a fair

chance the thing will work. Yes, I know you and Adam are two distinct people, but you have to admit this is pretty funny.

adamw October 18, 2019 at 11:25 am | Permalink | Reply

"Yes, I know you and Adam are two distinct people"

Er…are we? I was pretty sure I was just me. 😛

I mean, that's my practical recommendation for how to have a simple and happy life, born of painful experience. It applies to BIOS and UEFI (and every other case I've encountered, honestly). UEFI makes a game *effort* to improve things, and really does improve some things, and if it had all been implemented in the best way possible by everybody would *really* make multibooting less of a pain. But in the sad reality we live in, it gets implemented by firmware engineers and OS developers, so it is of course awful just like everything else…

Alexander October 24, 2019 at 2:18 am | Permalink | Reply

Gawd… What the Secure Boot has to do with ATi/NVIDIA proprietary drivers or your own kernels???

I didn't get that comletely.

Doesn't Secure Boot just define that partition too boot from is encrypted with some known to firmware key??

adamw December 10, 2019 at 6:04 pm | Permalink | Reply

No, it's not about encryption. It's about signing. Rather than "encrypted", Secure Boot says "all the bits involved in booting the system are *signed* with a known-to-firmware key".

The relationship to proprietary drivers and your own kernels is pretty simple: those things are part of the boot chain. So if you enable Secure Boot, they must be signed appropriately. If not, the system won't boot or the driver won't load.

Keitb March 7, 2020 at 7:05 am | Permalink | Reply

Love the article. Purchased new mobo and all the fixens to build system. Forget about issues that drove me here in first place. You must forgive us for saying UEFI BIOS when that's exactly what system says after booting into it.

Jo Hutabarat March 28, 2020 at 7:47 pm | Permalink | Reply

Pheeew… Thank you for the "human" explanation of the UEFI technical details contained in those various official documents.

I now understand why I kept failing to boot from a second installation of Debian 10 (BUSTER) on my external HDD.

In addition, this article has given me a fundamentally important expanded view on how I will from now on select my next mobo. Before reading this article, I took matters very simply when it came to choosing a motherboard: Does it support my chosen CPU? Does it have a built-in graphics adapter? What are the ports it has? How many memory slots are there? And so on. But BIOS was a given and never a consideration! I'd take whatever BIOS the mobo's maker had wisely adopted… (hahaha…)

Now, I will parse through as much information as I possible can on the UEFI installed on the mobo being considered.

Thank you again!