Pid Eins

レナート
لينارت

POSTED ON DO 23 SEPTEMBER 2021

Authenticated Boot and Disk Encryption on Linux

The Strange State of Authenticated Boot and Disk Encryption on Generic Linux Distributions

TL;DR: Linux has been supporting Full Disk Encryption (FDE) and technologies such as UEFI SecureBoot and TPMs for a long time. However, the way they are set up by most distributions is not as secure as they should be, and in some ways quite frankly weird. In fact, right now, your data is probably more secure if stored on current ChromeOS, Android, Windows or MacOS devices, than it is on typical Linux distributions.

Generic Linux distributions (i.e. Debian, Fedora, Ubuntu, …) adopted Full Disk Encryption (FDE) more than 15 years ago, with the LUKS/cryptsetup infrastructure. It was a big step forward to a more secure environment. Almost ten years ago the big distributions started adding UEFI SecureBoot to their boot process. Support for Trusted Platform Modules (TPMs) has been added to the distributions a long time ago as well — but even though many PCs/laptops these days have TPM chips on-board it's generally not used in the default setup of generic Linux distributions.

How these technologies currently fit together on generic Linux distributions doesn't really make too much sense to me — and falls short of what they could actually deliver. In this story I'd like to have a closer look at why I think that, and what I propose to do about it.

The Basic Technologies

Let's have a closer look what these technologies actually deliver:

1. LUKS/`dm-crypt`/`cryptsetup` provide disk encryption, and optionally data authentication. Disk encryption means that reading the data in clear-text form is only possible if you possess a secret of some form, usually a password/passphrase. Data authentication means that no one can make changes to the data on disk unless they possess a secret of some form. Most distributions only enable the former though — the latter is a more recent addition to LUKS/cryptsetup, and is not used by default on most distributions (though it probably should be). Closely related to LUKS/`dm-crypt` is `dm-verity` (which can authenticate immutable volumes) and `dm-integrity` (which can authenticate writable volumes, among other things).

2. UEFI SecureBoot provides mechanisms for authenticating boot loaders and other pre-OS binaries before they are invoked. If those boot loaders then authenticate the next step of booting in a similar fashion there's a chain of trust which can ensure that only code that has some level of trust associated with it will run on the system. Authentication of boot loaders is done via cryptographic signatures: the OS/boot loader vendors cryptographically sign their boot loader binaries. The cryptographic certificates that may be used to validate these signatures are then signed by Microsoft, and since Microsoft's certificates are basically built into all of today's PCs and laptops this will provide some basic trust chain: if you want to modify the boot loader of a system you must have access to the private key used to sign the code (or to the private keys further up the certificate chain).

3. TPMs do many things. For this text we'll focus one facet: they can be used to protect secrets (for example for use in disk encryption, see above), that are released only if the code that booted the host can be authenticated in some form. This works roughly like this: every component that is used during the boot process (i.e. code, certificates, configuration, …) is hashed with a cryptographic hash function before it is used. The resulting hash is written to some small volatile memory the TPM maintains that is write-only (the so called Platform Configuration Registers, "PCRs"): each step of the boot process will write hashes of the resources needed by the next part of the boot process into these PCRs. The PCRs cannot be written freely: the hashes written are combined with what is already stored in the PCRs — also through hashing and the result of that then replaces the previous value. Effectively this means: only if every component involved in the boot matches expectations the hash values exposed in the TPM

PCRs match the expected values too. And if you then use those values to unlock the secrets you want to protect you can guarantee that the key is only released to the OS if the expected OS and configuration is booted. The process of hashing the components of the boot process and writing that to the TPM PCRs is called "measuring". What's also important to mention is that the secrets are not only protected by these PCR values but encrypted with a "seed key" that is generated on the TPM chip itself, and cannot leave the TPM (at least so goes the theory). The idea is that you cannot read out a TPM's seed key, and thus you cannot duplicate the chip: unless you possess the original, physical chip you cannot retrieve the secret it might be able to unlock for you. Finally, TPMs can enforce a limit on unlock attempts per time ("anti-hammering"): this makes it hard to brute force things: if you can only execute a certain number of unlock attempts within some specific time then brute forcing will be prohibitively slow.

How Linux Distributions use these Technologies

As mentioned already, Linux distributions adopted the first two of these technologies widely, the third one not so much.

So typically, here's how the boot process of Linux distributions works these days:

1. The UEFI firmware invokes a piece of code called "shim" (which is stored in the EFI System Partition — the "ESP" — of your system), that more or less is just a list of certificates compiled into code form. The shim is signed with the aforementioned Microsoft key, that is built into all PCs/laptops. This list of certificates then can be used to validate the next step of the boot process. The shim is measured by the firmware into the TPM. (Well, the shim can do a bit more than what I describe here, but this is outside of the focus of this article.)

2. The shim then invokes a boot loader (often Grub) that is signed by a private key owned by the distribution vendor. The boot loader is stored in the ESP as well, plus some other places (i.e. possibly a separate boot partition). The corresponding certificate is included in the list of certificates built into the shim. The boot loader components are also measured into the TPM.

3. The boot loader then invokes the kernel and passes it an initial RAM disk image (initrd), which contains initial userspace code. The kernel itself is signed by the distribution vendor too. It's also validated via the shim. The initrd is not validated, though (!). The kernel is measured into the TPM, the initrd sometimes too.

4. The kernel unpacks the initrd image, and invokes what is contained in it. Typically, the initrd then asks the user for a password for the encrypted root file system. The initrd then uses that to set up the encrypted volume. No code authentication or TPM measurements take place.

5. The initrd then transitions into the root file system. No code authentication or TPM measurements take place.

6. When the OS itself is up the user is prompted for their user name, and their password. If correct, this will unlock the user account: the system is now ready to use. At this point no code authentication, no TPM measurements take place. Moreover, the user's password is not used to unlock any data, it's used only to allow or deny the login attempt — the user's data has already been decrypted a long time ago, by the initrd, as mentioned above.

What you'll notice here of course is that code validation happens for the shim, the boot loader and the kernel, but not for the initrd or the main OS code anymore. TPM measurements might go one step further: the initrd is measured sometimes too, if you are lucky. Moreover, you might notice that the disk encryption password and the user password are inquired by code that is not validated, and is thus not safe from external manipulation. You might also notice that even though TPM measurements of boot loader/OS components are done nothing actually ever makes use of the resulting PCRs in the typical setup.

Attack Scenarios

Of course, before determining whether the setup described above makes sense or not, one should have an idea what one actually intends to protect against.

The most basic attack scenario to focus on is probably that you want to be reasonably sure that if someone steals your laptop that contains all your data then this data remains confidential. The model described above probably delivers that to some degree: the full disk encryption when used with a reasonably strong password should make it hard for the laptop thief to access the data. The data is as secure as the password used is strong. The attacker might attempt to brute force the password, thus if the password is not chosen carefully the attacker might be successful.

Two more interesting attack scenarios go something like this:

1. Instead of stealing your laptop the attacker takes the harddisk from your laptop while you aren't watching (e.g. while you went for a walk and left it at home or in your hotel room), makes a copy of it, and then puts it back. You'll never notice they did that. The attacker then analyzes the data in their lab, maybe trying to brute force the password. In this scenario you won't even know that your data is at risk, because for you nothing changed — unlike in the basic scenario above. If the attacker manages to break your password they have full access to the data included on it, i.e. everything you so far stored on it, but not necessarily on what you are going to store on it later. This scenario is worse than the basic one mentioned above, for the simple fact that you won't know that you might be attacked. (This

scenario could be extended further: maybe the attacker has a chance to watch you type in your password or so, effectively lowering the password strength.)

2. Instead of stealing your laptop the attacker takes the harddisk from your laptop while you aren't watching, inserts backdoor code on it, and puts it back. In this scenario you won't know your data is at risk, because physically everything is as before. What's really bad though is that the attacker gets access to anything you do on your laptop, both the data already on it, and whatever you will do in the future.

I think in particular this backdoor attack scenario is something we should be concerned about. We know for a fact that attacks like that happen all the time (Pegasus, industry espionage, …), hence we should make them hard.

Are we Safe?

So, does the scheme so far implemented by generic Linux distributions protect us against the latter two scenarios? Unfortunately not at all. Because distributions set up disk encryption the way they do, and only bind it to a user password, an attacker can easily duplicate the disk, and then attempt to brute force your password. What's worse: since code authentication ends at the kernel — and the initrd is not authenticated anymore —, backdooring is trivially easy: an attacker can change the initrd any way they want, without having to fight any kind of protections. And given that FDE unlocking is implemented in the initrd, and it's the initrd that asks for the encryption password things are just too easy: an attacker could trivially easily insert some code that picks up the FDE password as you type it in and send it wherever they want. And not just that: since once they are in they are in, they can do anything they like for the rest of the system's lifecycle, with full privileges — including installing backdoors for versions of the OS or kernel that are installed on the device in the future, so that their backdoor remains open for as long as they like.

That is sad of course. It's particular sad given that the other popular OSes all address this much better. ChromeOS, Android, Windows and MacOS all have way better built-in protections against attacks like this. And it's why one can certainly claim that your data is probably better protected right now if you store it on those OSes then it is on generic Linux distributions.

(Yeah, I know that there are some niche distros which do this better, and some hackers hack their own. But I care about general purpose distros here, i.e. the big ones, that most people base their work on.)

Note that there are more problems with the current setup. For example, it's really weird that during boot the user is queried for an FDE password which actually protects their data, and then once the system is up they are queried again – now asking for a username, and another password. And the weird thing is that this second authentication that appears to be user-focused doesn't really protect the user's data anymore — at that moment the

data is already unlocked and accessible. The username/password query is supposed to be useful in multi-user scenarios of course, but how does that make any sense, given that these multiple users would all have to know a disk encryption password that unlocks the whole thing during the FDE step, and thus they have access to every user's data anyway if they make an offline copy of the harddisk?

Can we do better?

Of course we can, and that is what this story is actually supposed to be about.

Let's first figure out what the minimal issues we should fix are (at least in my humble opinion):

1. The initrd must be authenticated before being booted into. (And measured unconditionally.)

2. The OS binary resources (i.e. `/usr/`) must be authenticated before being booted into. (But don't need to be encrypted, since everyone has the same anyway, there's nothing to hide here.)

3. The OS configuration and state (i.e. `/etc/` and `/var/`) must be encrypted, and authenticated before they are used. The encryption key should be bound to the TPM device; i.e system data should be locked to a security concept belonging to the system, not the user.

4. The user's home directory (i.e. `/home/lennart/` and similar) must be encrypted and authenticated. The unlocking key should be bound to a user password or user security token (FIDO2 or PKCS#11 token); i.e. user data should be locked to a security concept belonging to the user, not the system.

Or to summarize this differently:

1. Every single component of the boot process and OS needs to be authenticated, i.e. all of shim (done), boot loader (done), kernel (done), initrd (missing so far), OS binary resources (missing so far), OS configuration and state (missing so far), the user's home (missing so far).

2. Encryption is necessary for the OS configuration and state (bound to TPM), and for the user's home directory (bound to a user password or user security token).

In Detail

Let's see how we can achieve the above in more detail.

How to Authenticate the initrd

At the moment initrds are generated on the installed host via scripts (dracut and similar) that try to figure out a minimal set of binaries and configuration data to build an initrd that contains just enough to be able to find and set up the root file system. What is included in the initrd hence depends highly on the individual installation and its configuration. Pretty likely no two initrds generated that way will be fully identical due to this. This model clearly has benefits: the initrds generated this way are very small and minimal, and support exactly what is necessary for the system to boot, and not less or more. It comes with serious drawbacks too though: the generation process is fragile and sometimes more akin to black magic than following clear rules: the generator script natively has to understand a myriad of storage stacks to determine what needs to be included and what not. It also means that authenticating the image is hard: given that each individual host gets a different specialized initrd, it means we cannot just sign the initrd with the vendor key like we sign the kernel. If we want to keep this design we'd have to figure out some other mechanism (e.g. a per-host signature key – that is generated locally; or by authenticating it with a message authentication code bound to the TPM). While these approaches are certainly thinkable, I am not convinced they actually are a good idea though: locally and dynamically generated per-host initrds is something we probably should move away from.

If we move away from locally generated initrds, things become a lot simpler. If the distribution vendor generates the initrds on their build systems then it can be attached to the kernel image itself, and thus be signed and measured along with the kernel image, without any further work. This simplicity is simply lovely. Besides robustness and reproducibility this gives us an easy route to authenticated initrds.

But of course, nothing is really that simple: working with vendor-generated initrds means that we can't adjust them anymore to the specifics of the individual host: if we pre-build the initrds and include them in the kernel image in immutable fashion then it becomes harder to support complex, more exotic storage or to parameterize it with local network server information, credentials, passwords, and so on. Now, for my simple laptop use-case these things don't matter, there's no need to extend/parameterize things, laptops and their setups are not that wildly different. But what to do about the cases where we want both: extensibility to cover for less common storage subsystems (iscsi, LVM, multipath, drivers for exotic hardware…) and parameterization?

Here's a proposal how to achieve that: let's build a basic initrd into the kernel as suggested, but then do two things to make this scheme both extensible and parameterizable, without compromising security.

1. Let's define a way how the basic initrd can be extended with additional files, which are stored in separate "extension images". The basic initrd should be able to discover these extension images, authenticate them and then activate them, thus extending the initrd with additional resources on-the-fly.

2. Let's define a way how we can safely pass additional parameters to the kernel/initrd (and actually the rest of the OS, too) in an authenticated (and possibly encrypted) fashion. Parameters in this context can be anything specific to the local installation, i.e. server information, security credentials, certificates, SSH server keys, or even just the root password that shall be able to unlock the root account in the initrd …

In such a scheme we should be able to deliver everything we are looking for:

1. We'll have a full trust chain for the code: the boot loader will authenticate and measure the kernel and basic initrd. The initrd extension images will then be authenticated by the basic initrd image.

2. We'll have authentication for all the parameters passed to the initrd.

This so far sounds very unspecific? Let's make it more specific by looking closer at the components I'd suggest to be used for this logic:

1. The `systemd` suite since a few months contains a subsystem implementing *system extensions* (v248). System extensions are ultimately just disk images (for example a squashfs file system in a GPT envelope) that can *extend* an underlying OS tree. Extending in this regard means they simply add additional files and directories into the OS tree, i.e. below `/usr/`. For a longer explanation see systemd-sysext(8). When a system extension is activated it is simply mounted and then merged into the main `/usr/` tree via a read-only overlayfs mount. Now what's particularly nice about them in this context we are talking about here is that the extension images may carry *dm-verity* authentication data, and PKCS#7 signatures (once this is merged, that is, i.e. v250).

2. The `systemd` suite also contains a concept called service "credentials". These are small pieces of information passed to services in a secure way. One key feature of these credentials is that they can be encrypted and authenticated in a very simple way with a key bound to the TPM (v250). See LoadCredentialEncrypted= and systemd-creds(1) for details. They are great for safely storing SSL private keys and similar on your system, but they also come handy for parameterizing initrds: an encrypted credential is just a file that can only be decoded if the right TPM is around with the right PCR values set.

3. The `systemd` suite contains a component called systemd-stub(7). It's an EFI stub, i.e. a small piece of code that is attached to a kernel image, and turns the kernel image into a regular EFI binary that can be directly executed by the firmware (or a boot loader). This stub has a number of nice features (for example, it can show a boot splash before invoking the Linux kernel itself and such). Once this work is merged (v250) the stub will support one more feature: it will automatically search

for system extension image files and credential files next to the kernel image file, measure them and pass them on to the main initrd of the host.

Putting this together we have nice way to provide fully authenticated kernel images, initrd images and initrd extension images; as well as encrypted and authenticated parameters via the credentials logic.

How would a distribution actually make us of this? A distribution vendor would pre-build the basic initrd, and glue it into the kernel image, and sign that as a whole. Then, for each supposed extension of the basic initrd (e.g. one for iscsi support, one for LVM, one for multipath, …), the vendor would use a tool such as mkosi to build an extension image, i.e. a GPT disk image containing the files in squashfs format, a Verity partition that authenticates it, plus a PKCS#7 signature partition that validates the root hash for the dm-verity partition, and that can be checked against a key provided by the boot loader or main initrd. Then, any parameters for the initrd will be encrypted using systemd-creds encrypt -T. The resulting encrypted credentials and the initrd extension images are then simply placed next to the kernel image in the ESP (or boot partition). Done.

This checks all boxes: everything is authenticated and measured, the credentials also encrypted. Things remain extensible and modular, can be pre-built by the vendor, and installation is as simple as dropping in one file for each extension and/or credential.

How to Authenticate the Binary OS Resources

Let's now have a look how to authenticate the Binary OS resources, i.e. the stuff you find in `/usr/`, i.e. the stuff traditionally shipped to the user's system via RPMs or DEBs.

I think there are three relevant ways how to authenticate this:

1. Make `/usr/` a `dm-verity` volume. `dm-verity` is a concept implemented in the Linux kernel that provides authenticity to read-only block devices: every read access is cryptographically verified against a *top-level hash value*. This top-level hash is typically a 256bit value that you can either encode in the kernel image you are using, or cryptographically sign (which is particularly nice once this is merged). I think this is actually the best approach since it makes the `/usr/` tree entirely immutable in a very simple way. However, this also means that the whole of `/usr/` needs to be updated as once, i.e. the traditional `rpm`/`apt` based update logic cannot work in this mode.

2. Make `/usr/` a `dm-integrity` volume. `dm-integrity` is a concept provided by the Linux kernel that offers integrity guarantees to writable block devices, i.e. in some ways it can be considered to be a bit like `dm-verity` while permitting write access. It can be used in three ways, one of which I think is particularly relevant here. The first way is with a simple hash function in "stand-alone" mode: this is not too

interesting here, it just provides greater data safety for file systems that don't hash check their files' data on their own. The second way is in combination with `dm-crypt`, i.e. with disk encryption. In this case it adds authenticity to confidentiality: only if you know the right secret you can read and make changes to the data, and any attempt to make changes without knowing this secret key will be detected as IO error on next read by those in possession of the secret (more about this below). The third way is the one I think is most interesting here: in "stand-alone" mode, but with a keyed hash function (e.g. HMAC). What's this good for? This provides authenticity without encryption: if you make changes to the disk without knowing the secret this will be noticed on the next read attempt of the data and result in IO errors. This mode provides what we want (authenticity) and doesn't do what we don't need (encryption). Of course, the secret key for the HMAC must be provided somehow, I think ideally by the TPM.

3. Make `/usr/` a `dm-crypt` (LUKS) + `dm-integrity` volume. This provides both authenticity and encryption. The latter isn't typically needed for `/usr/` given that it generally contains no secret data: anyone can download the binaries off the Internet anyway, and the sources too. By encrypting this you'll waste CPU cycles, but beyond that it doesn't hurt much. (Admittedly, some people might want to hide the precise set of packages they have installed, since it of course does reveal a bit of information about you: i.e. what you are working on, maybe what your job is – think: if you are a hacker you have hacking tools installed – and similar). Going this way might simplify things in some cases, as it means you don't have to distinguish "OS binary resources" (i.e `/usr/`) and "OS configuration and state" (i.e. `/etc/` + `/var/`, see below), and just make it the same volume. Here too, the secret key must be provided somehow, I think ideally by the TPM.

All three approach are valid. The first approach has my primary sympathies, but for distributions not willing to abandon client-side updates via RPM/dpkg this is not an option, in which case I would propose the other two approaches for these cases.

The LUKS encryption key (and in case of `dm-integrity` standalone mode the key for the keyed hash function) should be bound to the TPM. Why the TPM for this? You could also use a user password, a FIDO2 or PKCS#11 security token — but I think TPM is the right choice: why that? To reduce the requirement for repeated authentication, i.e. that you first have to provide the disk encryption password, and then you have to login, providing another password. It should be possible that the system boots up unattended and then only one authentication prompt is needed to unlock the user's data properly. The TPM provides a way to do this in a reasonably safe and fully unattended way. Also, when we stop considering just the laptop use-case for a moment: on servers interactive disk encryption prompts don't make much sense — the fact that TPMs can provide secrets without this requiring user interaction and thus the ability to work in entirely unattended environments is quite desirable. Note that crypttab(5) as implemented by

`systemd` (v248) provides native support for authentication via password, via TPM2, via PKCS#11 or via FIDO2, so the choice is ultimately all yours.

How to Encrypt/Authenticate OS Configuration and State

Let's now look at the OS configuration and state, i.e. the stuff in `/etc/` and `/var/`. It probably makes sense to not consider these two hierarchies independently but instead just consider this to be the root file system. If the OS binary resources are in a separate file system it is then mounted onto the `/usr/` sub-directory of the root file system.

The OS configuration and state (or: root file system) should be both encrypted and authenticated: it might contain secret keys, user passwords, privileged logs and similar. This data matters and contains plenty data that should remain confidential.

The encryption of choice here is `dm-crypt` (LUKS) + `dm-integrity` similar as discussed above, again with the key bound to the TPM.

If the OS binary resources are protected the same way it is safe to merge these two volumes and have a single partition for both (see above)

How to Encrypt/Authenticate the User's Home Directory

The data in the user's home directory should be encrypted, and bound to the user's preferred token of authentication (i.e. a password or FIDO2/PKCS#11 security token). As mentioned, in the traditional mode of operation the user's home directory is not individually encrypted, but only encrypted because FDE is in use. The encryption key for that is a system wide key though, not a per-user key. And I think that's problem, as mentioned (and probably not even generally understood by our users). We should correct that and ensure that the user's password is what unlocks the user's data.

In the `systemd` suite we provide a service systemd-homed(8) (v245) that implements this in a safe way: each user gets its own LUKS volume stored in a loopback file in `/home/`, and this is enough to synthesize a user account. The encryption password for this volume is the user's account password, thus it's really the password provided at login time that unlocks the user's data. `systemd-homed` also supports other mechanisms of authentication, in particular PKCS#11/FIDO2 security tokens. It also provides support for other storage back-ends (such as fscrypt), but I'd always suggest to use the LUKS back-end since it's the only one providing the comprehensive confidentiality guarantees one wants for a UNIX-style home directory.

Note that there's one special caveat here: if the user's home directory (e.g. `/home/lennart/`) is encrypted and authenticated, what about the file system this data is stored on, i.e. `/home/` itself? If that dir is part of the the root file system this would result in double encryption: first the data is encrypted with the TPM root file system key, and then again with the per-user key. Such double encryption is a waste of resources, and

unnecessary. I'd thus suggest to make `/home/` its own `dm-integrity` volume with a HMAC, keyed by the TPM. This means the data stored directly in `/home/` will be authenticated but not encrypted. That's good not only for performance, but also has practical benefits: it allows extracting the encrypted volume of the various users in case the TPM key is lost, as a way to recover from dead laptops or similar.

Why authenticate `/home/`, if it only contains per-user home directories that are authenticated on their own anyway? That's a valid question: it's because the kernel file system maintainers made clear that Linux file system code is not considered safe against rogue disk images, and is not tested for that; this means before you mount anything you need to establish trust in some way because otherwise there's a risk that the act of mounting might exploit your kernel.

Summary of Resources and their Protections

So, let's now put this all together. Here's a table showing the various resources we deal with, and how I think they should be protected (in my idealized world).

Resource	Needs Authentication	Needs Encryption	Suggested Technology	Validation/Encryption Keys/Certificates acquired via	Stored where
Shim	yes	no	SecureBoot signature verification	firmware certificate database	ESP
Boot loader	yes	no	ditto	firmware certificate database/shim	ESP/boot partition
Kernel	yes	no	ditto	ditto	ditto
initrd	yes	no	ditto	ditto	ditto
initrd parameters	yes	yes	systemd TPM encrypted credentials	TPM	ditto
initrd extensions	yes	no	`systemd-sysext` with Verity+PKCS#7 signatures	firmware/initrd certificate database	ditto
OS binary resources	yes	no	`dm-verity`	root hash linked into kernel image, or firmware/initrd certificate database	top-level partition
OS configuration and state	yes	yes	`dm-crypt` (LUKS) + `dm-integrity`	TPM	top-level partition

Resource	Needs Authentication	Needs Encryption	Suggested Technology	Validation/Encryption Keys/Certificates acquired via	Stored where
`/home/` itself	yes	no	`dm-integrity` with HMAC	TPM	top-level partition
User home directories	yes	yes	`dm-crypt` (LUKS) + `dm-integrity` in loopback files	User password/FIDO2/PKCS#11 security token	loopback file inside `/home` partition

This should provide all the desired guarantees: everything is authenticated, and the individualized per-host or per-user data is also encrypted. No double encryption takes place. The encryption keys/verification certificates are stored/bound to the most appropriate infrastructure.

Does this address the three attack scenarios mentioned earlier? I think so, yes. The basic attack scenario I described is addressed by the fact that `/var/`, `/etc/` and `/home/*` are encrypted. Brute forcing the former two is harder than in the status quo ante model, since a high entropy key is used instead of one derived from a user provided password. Moreover, the "anti-hammering" logic of the TPM will make brute forcing prohibitively slow. The home directories are protected by the user's password or ideally a personal FIDO2/PKCS#11 security token in this model. Of course, a password isn't better security-wise then the status quo ante. But given the FIDO2/PKCS#11 support built into `systemd-homed` it should be easier to lock down the home directories securely.

Binding encryption of `/var/` and `/etc/` to the TPM also addresses the first of the two more advanced attack scenarios: a copy of the harddisk is useless without the physical TPM chip, since the seed key is sealed into that. (And even if the attacker had the chance to watch you type in your password, it won't help unless they possess access to to the TPM chip.) For the home directory this attack is not addressed as long as a plain password is used. However, since binding home directories to FIDO2/PKCS#11 tokens is built into `systemd-homed` things should be safe here too — provided the user actually possesses and uses such a device.

The backdoor attack scenario is addressed by the fact that every resource in play now is authenticated: it's hard to backdoor the OS if there's no component that isn't verified by signature keys or TPM secrets the attacker hopefully doesn't know.

For general purpose distributions that focus on updating the OS per RPM/dpkg the idealized model above won't work out, since (as mentioned) this implies an immutable `/usr/`, and thus requires updating `/usr/` via an atomic update operation. For such distros a setup like the following is probably more realistic, but see above.

Resource	Needs Authentication	Needs Encryption	Suggested Technology	Validation/Encryption Keys/Certificates acquired via	Stored where
Shim	yes	no	SecureBoot signature verification	firmware certificate database	ESP
Boot loader	yes	no	ditto	firmware certificate database/shim	ESP/boot partition
Kernel	yes	no	ditto	ditto	ditto
initrd	yes	no	ditto	ditto	ditto
initrd parameters	yes	yes	systemd TPM encrypted credentials	TPM	ditto
initrd extensions	yes	no	`systemd-sysext` with Verity+PKCS#7 signatures	firmware/initrd certificate database	ditto
OS binary resources, configuration and state	yes	yes	`dm-crypt` (LUKS) + `dm-integrity`	TPM	top-level partition
`/home/` itself	yes	no	`dm-integrity` with HMAC	TPM	top-level partition
User home directories	yes	yes	`dm-crypt` (LUKS) + `dm-integrity` in loopback files	User password/FIDO2/PKCS#11 security token	loopback file inside `/home` partition

This means there's only one root file system that contains all of `/etc/`, `/var/` and `/usr/`.

Recovery Keys

When binding encryption to TPMs one problem that arises is what strategy to adopt if the TPM is lost, due to hardware failure: if I need the TPM to unlock my encrypted volume, what do I do if I need the data but lost the TPM?

The answer here is supporting recovery keys (this is similar to how other OSes approach this). Recovery keys are pretty much the same concept as passwords. The main difference being that they are computer generated rather than user-chosen. Because of that they typically have much higher entropy (which makes them more annoying to type in, i.e you want to use them only when you must, not day-to-day). By having higher

entropy they are useful in combination with TPM, FIDO2 or PKCS#11 based unlocking: unlike a combination with passwords they do not compromise the higher strength of protection that TPM/FIDO2/PKCS#11 based unlocking is supposed to provide.

Current versions of systemd-cryptenroll(1) implement a recovery key concept in an attempt to address this problem. You may enroll any combination of TPM chips, PKCS#11 tokens, FIDO2 tokens, recovery keys and passwords on the same LUKS volume. When enrolling a recovery key it is generated and shown on screen both in text form and as QR code you can scan off screen if you like. The idea is write down/store this recovery key at a safe place so that you can use it when you need it. Note that such recovery keys can be entered wherever a LUKS password is requested, i.e. after generation they behave pretty much the same as a regular password.

TPM PCR Brittleness

Locking devices to TPMs and enforcing a PCR policy with this (i.e. configuring the TPM key to be unlockable only if certain PCRs match certain values, and thus requiring the OS to be in a certain state) brings a problem with it: TPM PCR brittleness. If the key you want to unlock with the TPM requires the OS to be in a specific state (i.e. that all OS components' hashes match certain expectations or similar) then doing OS updates might have the affect of making your key inaccessible: the OS updates will cause the code to change, and thus the hashes of the code, and thus certain PCRs. (Thankfully, you unrolled a recovery key, as described above, so this doesn't mean you lost your data, right?).

To address this I'd suggest three strategies:

1. Most importantly: don't actually use the TPM PCRs that contain code hashes. There are actually multiple PCRs defined, each containing measurements of different aspects of the boot process. My recommendation is to bind keys to PCR 7 only, a PCR that contains measurements of the UEFI SecureBoot certificate databases. Thus, the keys will remain accessible as long as these databases remain the same, and updates to code will not affect it (updates to the certificate databases will, and they do happen too, though hopefully much less frequent then code updates). Does this reduce security? Not much, no, because the code that's run is after all not just measured but also validated via code signatures, and those signatures are validated with the aforementioned certificate databases. Thus binding an encrypted TPM key to PCR 7 should enforce a similar level of trust in the boot/OS code as binding it to a PCR with hashes of specific versions of that code. i.e. using PCR 7 means you say "every code signed by these vendors is allowed to unlock my key" while using a PCR that contains code hashes means "only this exact version of my code may access my key".

2. Use LUKS key management to enroll multiple versions of the TPM keys in relevant volumes, to support multiple versions of the OS code (or multiple

versions of the certificate database, as discussed above). Specifically: whenever an update is done that might result changing the relevant PCRs, pre-calculate the new PCRs, and enroll them in an additional LUKS slot on the relevant volumes. This means that the unlocking keys tied to the TPM remain accessible in both states of the system. Eventually, once rebooted after the update, remove the old slots.

3. If these two strategies didn't work out (maybe because the OS/firmware was updated outside of OS control, or the update mechanism was aborted at the wrong time) and the TPM PCRs changed unexpectedly, and the user now needs to use their recovery key to get access to the OS back, let's handle this gracefully and automatically reenroll the current TPM PCRs at boot, after the recovery key checked out, so that for future boots everything is in order again.

Other approaches can work too: for example, some OSes simply remove TPM PCR policy protection of disk encryption keys altogether immediately before OS or firmware updates, and then reenable it right after. Of course, this opens a time window where the key bound to the TPM is much less protected than people might assume. I'd try to avoid such a scheme if possible.

Anything Else?

So, given that we are talking about idealized systems: I personally actually think the ideal OS would be much simpler, and thus more secure than this:

I'd try to ditch the Shim, and instead focus on enrolling the distribution vendor keys directly in the UEFI firmware certificate list. This is actually supported by all firmwares too. This has various benefits: it's no longer necessary to bind everything to Microsoft's root key, you can just enroll your own stuff and thus make sure only what you want to trust is trusted and nothing else. To make an approach like this easier, we have been working on doing automatic enrollment of these keys from the `systemd-boot` boot loader, see this work in progress for details. This way the Firmware will authenticate the boot loader/kernel/initrd without any further component for this in place.

I'd also not bother with a separate boot partition, and just use the ESP for everything. The ESP is required anyway by the firmware, and is good enough for storing the few files we need.

FAQ

Can I implement all of this in my distribution today?

Probably not. While the big issues have mostly been addressed there's a lot of integration work still missing. As you might have seen I linked some PRs that haven't even been merged into our tree yet, and definitely not been released yet or even entered the distributions.

Will this show up in Fedora/Debian/Ubuntu soon?

I don't know. I am making a proposal how these things might work, and am working on getting various building blocks for this into shape. What the distributions do is up to them. But even if they don't follow the recommendations I make 100%, or don't want to use the building blocks I propose I think it's important they start thinking about this, and yes, I think they should be thinking about defaulting to setups like this.

Work for measuring/signing initrds on Fedora has been started, here's a slide deck with some information about it.

But isn't a TPM evil?

Some corners of the community tried (unfortunately successfully to some degree) to paint TPMs/Trusted Computing/SecureBoot as generally evil technologies that stop us from using our systems the way we want. That idea is rubbish though, I think. We should focus on what it can deliver for us (and that's a lot I think, see above), and appreciate the fact we can actually use it to kick out perceived evil empires from our devices instead of being subjected to them. Yes, the way SecureBoot/TPMs are defined puts *you* in the driver seat if you want — and you may enroll your own certificates to keep out everything you don't like.

What if my system doesn't have a TPM?

TPMs are becoming quite ubiquitous, in particular as the upcoming Windows versions will require them. In general I think we should focus on modern, fully equipped systems when designing all this, and then find fall-backs for more limited systems. Frankly it feels as if so far the design approach for all this was the other way round: try to make the new stuff work like the old rather than the old like the new (I mean, to me it appears this thinking is the main raison d'être for the Grub boot loader).

More specifically, on the systems where we have no TPM we ultimately cannot provide the same security guarantees as for those which have. So depending on the resource to protect we should fall back to different TPM-less mechanisms. For example, if we have no TPM then the root file system should probably be encrypted with a user provided password, typed in at boot as before. And for the encrypted boot credentials we probably should simply not encrypt them, and place them in the ESP unencrypted.

Effectively this means: without TPM you'll still get protection regarding the basic attack scenario, as before, but not the other two.

What if my system doesn't have UEFI?

Many of the mechanisms explained above taken individually do not require UEFI. But of course the chain of trust suggested above requires something like UEFI SecureBoot.

If your system lacks UEFI it's probably best to find work-alikes to the technologies suggested above, but I doubt I'll be able to help you there.

rpm/dpkg already cryptographically validates all packages at installation time (`gpg`), why would I need more than that?

This type of package validation happens once: at the moment of installation (or update) of the package, but not anymore when the data installed is actually used. Thus when an attacker manages to modify the package data after installation and before use they can make any change they like without this ever being noticed. Such package download validation does address certain attack scenarios (i.e. man-in-the-middle attacks on network downloads), but it doesn't protect you from attackers with physical access, as described in the attack scenarios above.

Systems such as `ostree` aren't better than rpm/dpkg regarding this BTW, their data is not validated on use either, but only during download or when processing tree checkouts.

Key really here is that the scheme explained here provides *offline* protection for the data "at rest" — even someone with physical access to your device cannot easily make changes that aren't noticed on next use. rpm/dpkg/ostree provide *online* protection only: as long as the system remains up, and all OS changes are done through the intended program code-paths, and no one has physical access everything should be good. In today's world I am sure this is not good enough though. As mentioned most modern OSes provide offline protection for the data at rest in one way or another. Generic Linux distributions are terribly behind on this.

This is all so desktop/laptop focused, what about servers?

I am pretty sure servers should provide similar security guarantees as outlined above. In a way servers are a much simpler case: there are no users and no interactivity. Thus the discussion of `/home/` and what it contains and of user passwords doesn't matter. However, the authenticated initrd and the unattended TPM-based encryption I think are very important for servers too, in a trusted data center environment. It provides security guarantees so far not given by Linux server OSes.

I'd like to help with this, or discuss/comment on this

Submit patches or reviews through GitHub. General discussion about this is best done on the systemd mailing list.

Category: projects

← BACK TO INDEX