Protecting SSH authentication with TPM 2.0

Nicolas Iooss
nicolas.iooss@ledger.fr


Ledger Donjon

Abstract. For quite some time, desktop computers have been embedding a security chip. This chip, named Trusted Platform Module (TPM), provides many features including the ability to protect private keys used in public-key cryptography. Such keys can be used to authenticate users in network protocols such as Secure Shell (SSH).

Software stacks which enable using a TPM to secure the keys used in SSH have been available for several years. Yet their use for practical purposes is often documented only from a high-level perspective, which does not help answering questions such as: can the security properties of keys protected by a TPM be directly applied to SSH keys?

This is a non-trivial question as for example those properties do not apply for disk encryption with sealed keys. Therefore this article fills the documentation gap between the TPM specifications and the high-level documentations about using a TPM to use SSH keys. It does so by studying how SSH keys are stored when using `tpm2-pkcs11` library on a Linux system.

1 Introduction

1.1 SSH, TPM and how they can be used together

When using remote services, users need to be authenticated. Every protocol relies on different mechanisms to implement user authentication: a secret password, an RSA key pair on a smartcard, a shared secret used with a TOTP (Time-based One-Time Password) application on a smartphone, etc. Many authentication mechanisms are considered highly secure nowadays (for example the ones relying on dedicated devices such as smartcards). However when discussing habits with other people, it seems that many still continue to use very weak mechanisms instead, even for protocols such as SSH which can be considered as mainly used by people interested in computer science (developers, system administrators, researchers, etc.).

SSH (Secure Shell) is a network protocol which can be used to access a remote command-line interface (“a shell”), transmit files to a server,

forward other network protocols, etc. It is possible to use public-key cryptography to authenticate to an SSH server, with an unencrypted private key. Such a configuration can be reproduced by running `ssh-keygen -t rsa -N ""` on the client. Doing so, a client RSA private key is generated in `.ssh/id_rsa` and its matching public key is saved in `.ssh/id_rsa.pub` (in the home directory of the user who ran this command). By copying the content of `.ssh/id_rsa.pub` into the list of their authorized keys on an SSH server (for example in remote file `.ssh/authorized_keys`),[1] the user can establish connections to the SSH server without entering any password. However, if someone gets access to the content of `.ssh/id_rsa` (for example by executing malicious software on the victim's computer or by getting the hard drive while the computer is powered down), this attacker can impersonate the victim and connect to the SSH server independently of the victim.

In order to prevent this attack, it is recommended to protect the private key with a secret password. Doing so prevents an attacker from getting the key by directly copying the file, but this does not prevent malicious software from stealing the private key. Indeed, such software can record the keystrokes in order to steal the password, or dump the memory of the SSH agent process while the private key is loaded.

To increase the protection of the private key even more, it is recommended to use dedicated hardware to store it. Several manufacturers have been building such hardware for more than ten years and nowadays users can buy (in alphabetical order) a Ledger Nano, a Nitrokey, a PGP smartcard, a Solo security key, a Titan security key, a Yubikey, or one out of many other products. All these products work in a similar way: they store a private key and they enable users to authenticate (which usually consists in signing a randomly generated message with the private key), possibly after users entered their password or PIN code, after they pressed some buttons and after they verified something on the embedded screen (depending on the product). Nevertheless all these products share a drawback: they cost money.

Can SSH users rely on something more secure than a password-protected file to store their private key for free? Yes, thanks to a component called TPM (Trusted Platform Module) which is likely available on recent computers.

1. The copy of the public key to `.ssh/authorized_keys` can be done using a tool such as `ssh-copy-id` (`https://manpages.debian.org/stretch/openssh-client/ssh-copy-id.1.en.html`).

A TPM can perform many operations, including protecting a private key, while implementing a specification published by the TCG (Trusted Computing Group). Thanks to the Microsoft Logo Program [6], desktop computers which come with Windows 10 (since July 2016) are required to have a component which implements the TPM 2.0 specification. In practice, this component is either a real chip or a *firmware TPM* that runs on an existing chip. For example, some computers powered by Intel processors use the CSME (Converged Security and Manageability Engine) [2] to implement a firmware TPM on the PCH (Platform Controller Hub),[3] a chip located on the motherboard.

On Linux, creating an SSH key stored on a TPM 2.0 can be achieved thanks to software developed on `https://github.com/tpm2-software` and thanks to the PKCS#11 [4] interface of OpenSSH. For example, users running Arch Linux can use the following commands (listing 1):

```
1 sudo pacman -S tpm2-pkcs11
2 tpm2_ptool init
3 tpm2_ptool addtoken --pid=1 --label=ssh --userpin=XXXX --sopin=YYYY
4 tpm2_ptool addkey --label=ssh --userpin=XXXX --algorithm=ecc256
```

Listing 1. Commands to create a key stored on a TPM, for Arch Linux users

These commands install the required software, create a *PKCS#11 token* authenticated by the *user PIN* `XXXX` and the SOPIN (*Security Officer PIN*) `YYYY`, and make the TPM generate a private key on the NIST P-256 curve. In PKCS#11 standard, *Security Officer* is a kind of user who is responsible for administering the normal users and for performing operations such as initially setting and changing passwords. The Security Officers authenticate with a specific password, called SOPIN, and have the power to modify the *user PIN* for example when it has been lost.

The public key associated with this new key is available by running (listing 2):

```
1 ssh-keygen -D /usr/lib/pkcs11/libtpm2_pkcs11.so
```

Listing 2. Command to query the public parts of keys stored on a TPM

2. `https://en.wikipedia.org/wiki/Intel_Management_Engine`
3. `https://en.wikipedia.org/wiki/Platform_Controller_Hub`
4. Public-Key Cryptography Standards #11 is a standard which defines a programming interface to create and manipulate cryptographic tokens, `https://en.wikipedia.org/wiki/PKCS_11`

In order to use the generated key when connecting to a server, it is either possible to use a command-line option, `ssh -I /usr/lib/pkcs11/libtpm2_pkcs11.so`, or to add a line in the configuration file of the client, `PKCS11Provider /usr/lib/pkcs11/libtpm2_pkcs11.so`.

Doing so, the private key is no longer directly stored in a file. However the documentation of `tpm2-pkcs11` states that this key is stored in a database, located in `.tpm2_pkcs11/tpm2_pkcs11.sqlite3` in the home directory of the user. Does this mean that stealing this file is enough to impersonate the user? Is there any software (which could be compromised) that sees the private key when the user uses it to connect to a server? How is the TPM actually used to authenticate the user?

Surprisingly, answering these questions is not straightforward at all. This document aims at studying these questions in order to provide concise and precise answers which can be used to better understand how this works.

1.2 TPM in the literature

TPM components are not new: the TPM specification was first standardized in 2009 as ISO/IEC 11889, even though it was possible to use TPM before. The specifications for TPM 1.2 revision were published in 2011 and those for TPM 2.0 in 2014. These specifications include many features.

First, a TPM contains some non-persistent memory which is called PCR (Platform Configuration Registers). These registers hold cryptographic digests computed from the boot code and the boot configuration data. If any of this code or configuration changes, the digests change. In order to prove that the content of the PCR really comes from the TPM, the TPM is able to sign the content of the PCR using a special key which is stored in it, called the AIK (Attestation Identity Key) in TPM 1.2 or AK (Attestation Key) in TPM 2.0.

Second, a TPM contains some private key material which can be used to decrypt or sign data, with public-key encryption schemes[5] (RSA,[6] ECDSA,[7] etc.). A TPM also contains secret key material used with symmetric encryption algorithms. This enables a TPM to work with a large number of keys while having a limited amount of persistent memory: when

5. `https://en.wikipedia.org/wiki/Public-key_cryptography#Examples`
6. `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`
7. `https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm`

a key pair is generated by the TPM, the private key is encrypted using symmetric encryption and the result can be stored outside of the TPM. To use such a key, the software first needs to load the encrypted private key into the TPM, which decrypts it using a secret key which never leaves the TPM.

Third, a TPM can encrypt some data in a way that the result can only be decrypted when some conditions happen (for example "someone entered some kind of password" or "some PCR hold some specific values"). This function is called *sealing data* when the data is encrypted and *unsealing data* when it is decrypted.

Fourth, a TPM contains some storage named *NV Indexes* (Non-Volatile). This storage can contain certificates for the public keys associated with the private keys held by the TPM, as well as other information. The access to a NV Index can be restricted using several checks in a similar way as the one used in sealing operations.

These features are represented in figure 1.

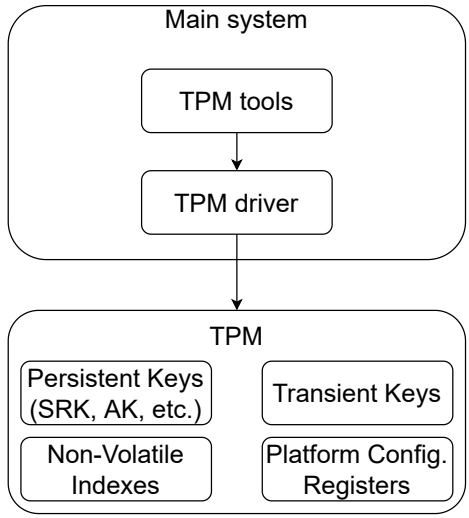

Fig. 1. Architecture of a system with a TPM

These features and many more (such as firmware upgrade, integration with Intel SGX, etc.) have been well studied over the last decade.

For example several people used a TPM as a way to improve the security of full-disk encryption by sealing a password used to decrypt the disk (using the content of some PCRs and eventually a password

called *TPM PIN code* to unseal the password). This is what Microsoft implemented in BitLocker, which was presented at SSTIC in 2006 [10] and in 2011 [2]. During the past two years, some people have been proposing to perform something similar in `cryptsetup` for Linux [8] and there was recent activity on this topic.[9]

It is possible to restrict some operations on a TPM 2.0 using an *E/A policy* (Enhanced Authorization policy). This complex mechanism was presented by Andreas Fuchs during Linux Security Europe 2020 [4].

Regarding the way TPM stores private keys, James Bottomley from IBM gave a talk at the Kernel Recipes 2018 conference [3] and he repeatedly sent patches in order to store GnuPG keys in the TPM.[10] Such patches also help using SSH, as SSH can be configured to use GnuPG keys for authentication. However at the time of writing, none of these patches were accepted by GnuPG's developers, which is why this document will not talk about GnuPG at all.

Regarding using a TPM to store SSH keys, several websites already document the same commands as the one presented in the introduction (for example `https://medium.com/google-cloud/google-cloud-ssh-with-os-login-with-yubikey-opensc-pkcs11-and-trusted-platform-module-tpm-based-86fa22a30f8d`, `https://incenp.org/notes/2020/tpm-based-ssh-key.html` and `https://linuxfr.org/news/utilisation-d-un-tpm-pour-l-authentification-ssh`). But none of these websites dig into the details on how the key is stored or how the SOPIN is actually implemented.

Even though many websites document how to use a TPM with SSH, only a few people seem to actually use this. One of the reasons could be that `tpm2-pkcs11` is a recent project which was not properly packaged in Debian (and Ubuntu) before January 2021.[11] The author of this document helped fixing this and his contribution was acknowledged by the package maintainer.[12] Hopefully the future release of Debian 11 and Ubuntu 21.04

8. `https://gitlab.com/cryptsetup/cryptsetup/-/merge_requests/51`
9. `https://gitlab.com/cryptsetup/cryptsetup/-/merge_requests/98`
10. In 2018 `https://lists.gnupg.org/pipermail/gnupg-devel/2018-January/033350.html` and in 2020 `https://lists.gnupg.org/pipermail/gnupg-devel/2020-June/034621.html`
11. `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968310`
12. `https://salsa.debian.org/debian/tpm2-pkcs11/-/commit/f76eb1d484dea1a38d0ad3fbdca779f84d1d9248`

will provide usable packages. A list of Linux distributions which package tpm2-pkcs11 can be found on Repology.[13]

On the hardware level, several people took a look at TPM chips and their communication channels. At Black Hat DC 2010, Christopher Tarnovsky presented how he managed to dump the code running on an Infineon TPM through advanced hardware attacks [11]. Jeremy Boone from NCC Group presented at CanSecWest 2018 how to build a device which sits between some TPM and the CPU, called *TPM Genie* [1]. This device enabled attackers to unseal the secrets used to encrypt a hard drive encrypted with Microsoft's BitLocker. This attack was reproduced in 2020 by F-Secure [9].

On the cryptographic level, several vulnerabilities were discovered throughout the years. In 2017, it was discovered that some TPM from Infineon were generating RSA keys with a bias that enabled cracking them in a reasonable amount of time (The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli, ACM CCS 2017 [8]). In 2019, it was discovered that the ECDSA implementations of some TPM from Intel and STMicroelectronics were vulnerable to an attack which enabled attackers to recover the private key [7]. These attacks could compromise the SSH keys protected by the TPM. Nevertheless these attacks do not mean that TPM and dedicated secure hardware are worthless to store private keys: these attacks remain much more complex to perform than stealing the private key stored in a file.

2 Configuring a system to use a TPM 2.0 to secure SSH keys

2.1 Finding out whether a system has a TPM 2.0

In order to study how a TPM 2.0 is used for SSH authentication, it is necessary to have a software layer which implements a TPM 2.0 interface. There are several ways of doing this.

But first, how is it possible to determine whether a TPM is available? The usual tools that enable enumerating hardware components can help:

— The BIOS user interface/setup menu (available at boot time) might contain some configuration options related to the TPM, for example to enable it.

— The filesystem might contain a device named `/dev/tpm0` and a non-empty directory `/sys/class/tpm`.

13. `https://repology.org/project/tpm2-pkcs11/versions`

— The kernel logs (command `dmesg`) might contain a line such as: `tpm_tis NTC0702:00: 2.0 TPM (device-id 0xFC, rev-id 1)`.
— Commands such as `fwupdmgr get-devices -show-all-devices` might give information about an existing TPM.
— The author of this document also created a tool for Linux machines which represents in a graph the devices of a computer (`https://github.com/fishilico/home-files/blob/master/bin/graph-hw`). This tool was presented in a rump session at SSTIC 2018 [5].

When a TPM is present, a file could be present (since Linux 5.5) to request the major version of the specification which is used (listing 3):

```
$ cat /sys/class/tpm/tpm0/tpm_version_major
2
```

Listing 3. Query the major version of the TPM used by the system, when using TPM 2.0

Information such as the manufacturer of the TPM and product information can be queried using *TPM capabilities*. A command provided by project `tpm2-tools` can be used to perform such a query on a TPM 2.0 (listing 4):

```
$ tpm2_getcap --tcti device:/dev/tpmrm0 properties-fixed
TPM2_PT_FAMILY_INDICATOR:
  raw: 0x322E3000
  value: "2.0"
TPM2_PT_LEVEL:
  raw: 0
TPM2_PT_REVISION:
  value: 1.38
TPM2_PT_DAY_OF_YEAR:
  raw: 0x8
TPM2_PT_YEAR:
  raw: 0x7E2
TPM2_PT_MANUFACTURER:
  raw: 0x4E544300
  value: "NTC"
TPM2_PT_VENDOR_STRING_1:
  raw: 0x4E504354
  value: "NPCT"
TPM2_PT_VENDOR_STRING_2:
  raw: 0x37357800
  value: "75x"
TPM2_PT_VENDOR_STRING_3:
  raw: 0x2010024
  value: ""
TPM2_PT_VENDOR_STRING_4:
  raw: 0x726C7300
  value: "rls"
TPM2_PT_VENDOR_TPM_TYPE:
```

```
29   raw: 0x0
30 TPM2_PT_FIRMWARE_VERSION_1:
31   raw: 0x70002
32 TPM2_PT_FIRMWARE_VERSION_2:
33   raw: 0x10000
```

Listing 4. Query all fixed properties from a TPM 2.0

Why is `/dev/tpmrm0` used in the command line? When issuing commands to a TPM, it is recommended to use a *TPM Resource Manager*. This is because a TPM has a very limited capacity which limits the number of cryptographic keys it can hold in memory. The *TPM Resource Manager* acts as a proxy to the TPM and enables using any number of keys. It works by issuing `ContextSave`, `ContextLoad` and `FlushContext` commands to export, restore and destroy data in its non-persistent memory. Doing so, the *TPM Resource Manager* gives the impression of using a TPM without any capacity limit.

In practice, since Linux 4.12 (released in 2017) the kernel has been implementing a *TPM Resource Manager* which can be used through device `/dev/tpmrm0`. Before, it was recommended to use a user-space *TPM Access Broker and Resource Manager Daemon* (project `tpm2-abrmd`[14]) instead of communicating with the device through `/dev/tpm0`, but this recommendation does not apply any more.[15]

In order to query all the information which is available without authentication from a TPM, the author of this document wrote a Python script (`https://github.com/fishilico/home-files/blob/master/bin/tpm-show`).

2.2 Emulating a TPM 2.0

If the system does not have a TPM 2.0 chip or if the user wants to perform tests on a development TPM without breaking their real TPM, it is possible to use a simulator. At the time of writing, there are mainly two projects that can be used to launch a software TPM:

— `swtpm`,[16] which implements a front-end for `libtpms`,[17] a library which targets the integration of TPM functionality into hypervisors, primarily into QEMU.

14. `https://github.com/tpm2-software/tpm2-abrmd`
15. cf. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fdc915f7f71939ad5a3dda3389b8d2d7a7c5ee66` for details
16. `https://github.com/stefanberger/swtpm`
17. `https://github.com/stefanberger/libtpms`

— `tpm_server`,[18] which defines itself as an implementation of the TCG Trusted Platform Module 2.0 specification.

Both simulators are maintained by IBM, and the second one is based on the TPM specification source code donated by Microsoft (according to its `README`). In order to be able to use complex commands with the simulators, it is required to use a TPM Resource Manager such as `tpm2-abrmd`, which provides a D-Bus service. Last but not least, each simulator uses a different protocol to encapsulate TPM 2.0 commands. The protocol used by TPM tools and libraries is configured through a mechanism called TCTI (TPM Command Transmission Interface).

In short, launching a software TPM is quite complex but once all those requirements are known, it is possible to document how it can be done.

Both simulators are packaged on several Linux distribution, including Arch Linux. Readers who are interested in reproducing the instructions of this section can start a container for example with `podman run -rm -ti docker.io/library/archlinux`.[19]

— To use `swtpm` (with TCTI library `/usr/lib/libtss2-tcti-swtpm.so`, listing 5):

```
pacman -Syu swtpm tpm2-abrmd tpm2-tools
swtpm socket --tpm2 --daemon \
    --server port=2321 --ctrl type=tcp,port=2322 \
    --flags not-need-init --tpmstate dir=/tmp \
    --log file=/tmp/swtpm.log,level=5
mkdir -p /run/dbus && dbus-daemon --system --fork
tpm2-abrmd --allow-root --tcti swtpm:host=127.0.0.1,port=2321 &
export TPM2TOOLS_TCTI=tabrmd:bus_type=system
```

Listing 5. Install and launch a TPM 2.0 simulator on Arch Linux, with `swtpm`

— To use `tpm_server` (with TCTI library `/usr/lib/libtss2-tcti-mssim.so`,[20] listing 6):

```
pacman -Syu ibm-sw-tpm2 tpm2-abrmd tpm2-tools
tpm_server -port 2321 > /tmp/tpm_server.log &
mkdir -p /run/dbus && dbus-daemon --system --fork
tpm2-abrmd --allow-root --tcti mssim:host=127.0.0.1,port=2321 &
export TPM2TOOLS_TCTI=tabrmd:bus_type=system
```

Listing 6. Install and launch a TPM 2.0 simulator on Arch Linux, with `tpm_server`

18. `https://github.com/kgoldman/ibmswtpm2`

19. Users more familiar with Docker can instead use: `sudo docker run -rm -ti docker.io/library/archlinux`

20. MSSIM means *Microsoft Simulator*. A few years ago, Microsoft published the source code of a TPM simulator and this code was modified to run on Linux in a program which became `tpm_server`. `libtss2-tcti-mssim.so` implements the protocol used by this simulator.

A third alternative consists in creating virtual devices very similar to `/dev/tpm0` and `/dev/tpmrm0` using a module called the *virtual TPM proxy* available since Linux 4.8 (listing 7):

```
pacman -Syu swtpm tpm2-tools
modprobe tpm_vtpm_proxy
swtpm chardev --tpm2 --vtpm-proxy --tpmstate dir=/var/lib/swtpm
```

Listing 7. Install and launch a TPM 2.0 simulator on Arch Linux, with `swtpm` and the virtual TPM proxy

In order to check that the software TPM launched by any of these alternatives works fine, it is possible to query the TPM with `tpm2_getcap properties-fixed`, `tpm2_pcrread`, etc.

The TPM Software Stack (TSS) includes a high-level interface called FAPI (TSS 2.0 Feature Application Programming Interface). It is not possible to directly use it with a software TPM because the default configuration requires the presence of an Endorsement Key Certificate. In order to use FAPI, a specific configuration file can be written to remove this requirement (listing 8):

```
echo > /etc/tpm2-tss/stpm_fapi_config.json \
    '{"profile_name": "P_ECCP256SHA256",' \
    '"profile_dir": "/etc/tpm2-tss/fapi-profiles/",' \
    '"user_dir": "~/.local/share/tpm2-tss/user/keystore",' \
    '"system_dir": "/var/lib/tpm2-tss/system/keystore",' \
    '"log_dir": "/run/tpm2_tss",' \
    '"tcti": "'"${TPM2TOOLS_TCTI}"'",' \
    '"system_pcrs": [],' \
    '"ek_cert_less": "yes"}'
export TSS2_FAPICONF=/etc/tpm2-tss/stpm_fapi_config.json
tss2_provision
```

Listing 8. Configure FAPI with a software TPM 2.0

The last command creates a SRK (Storage Root Key) usable by `tpm2-pkcs11`, at the handle `0x81000001`. This key is used to store private keys and secrets in the TPM, in a way which guarantees some security properties. Its public key can be read with `tpm2_readpublic -c 0x81000001`.

3 tpm2-pkcs11 storage of the SSH key

3.1 Storage of the public key

Back to `tpm2-pkcs11`: where is the private SSH key stored and how is it decrypted?

The readers who are familiar with how TPMs are used in disk encryption are likely to make the guess that the private key is simply *unsealed* from the TPM. That would mean that the key is known by the software (the SSH client or one of its libraries) and that the TPM is only used to store a passphrase for a private key file. However a TPM can also directly load a private key and use it, without exposing it to the software. Using this feature would strengthen the security of the key storage. Therefore there appears to be a contradiction between some intuition (that keys could be *unsealed*) and the features of TPMs. How is the private key processed?

The analysis of `tpm2-pkcs11` source code reveals that private keys are indeed used by the TPM when performing signature operations (using function `Esys_Sign` in `https://github.com/tpm2-software/tpm2-pkcs11/blob/1.5.0/src/lib/tpm.c#L1190`).

Nevertheless another file, `src/lib/utils.c` contains calls to software implementation of AES-GCM, in function `aes256_gcm_encrypt` and `aes256_gcm_decrypt`. These functions appear to be used to *wrap* and *unwrap* (which mean *encrypt* and *decrypt*) some data named `objauth`, using the AES key which is unsealed from the TPM. In order to understand what this `objauth` is, the persistent storage of `tpm2-pkcs11` can be analyzed after the three previous `tpm2_ptool` commands from listing 1 are issued.

This storage consists in a SQLite database which by default is created in the home directory of the current user. It contains 5 tables when using `tpm2-pkcs11` version 1.5.0 (listing 9):

```
$ sqlite3 "$HOME/.tpm2_pkcs11/tpm2_pkcs11.sqlite3"
sqlite> .tables
pobjects     schema       sealobjects  tobjects     tokens
```

Listing 9. Tables in `tpm2-pkcs11` database

These tables are used to link PKCS#11 concepts to the TPM world.

In PKCS#11, a slot may contain a token, which contains several objects such as keys and certificates. Information about slots, tokens and objects can be queried using command `pkcs11-tool` from package `opensc` (listing 10):

```
$ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
    --show-info
Cryptoki version 2.40
Manufacturer     tpm2-software.github.io
Library          TPM2.0 Cryptoki (ver 0.0)
Using slot 0 with a present token (0x1)
```

```
7
8  $ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
9      --list-token-slots
10 Available slots:
11 Slot 0 (0x1): ssh                                IBM
12   token label        : ssh
13   token manufacturer : IBM
14   token model        : SW   TPM
15   token flags        : login required, rng, token initialized,
16 PIN initialized
17   hardware version   : 1.50
18   firmware version   : 23.25
19   serial num         : 0000000000000000
20   pin min/max        : 0/128
21 Slot 1 (0x2):                                    IBM
22   token state:   uninitialized
23
24 $ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
25     --list-objects
26 Using slot 0 with a present token (0x1)
27 Public Key Object; EC  EC_POINT 256 bits
28   EC_POINT:   0441043eef05ada9dc42f69ffca066adfc374ec94aaba63bfa
29 9383c2a563d847f31ac250702adc8e1081d1b633a1e1d6278b4613ba20cf5fd8
30 af0b8c3c8b4a765b9387
31   EC_PARAMS:  06082a8648ce3d030107
32   label:
33   ID:         353864613830613533633663665366439 35
34   Usage:      encrypt, verify
35   Access:     local
```

Listing 10. Output of `pkcs11-tool` on a system using a software TPM

These commands did not interact with the TPM, even though the content of the generated public key was displayed.[21] This information is indeed stored in the SQLite database. More precisely, the `tobjects` table contains information about *transient objects*, including all their associated PKCS#11 attributes. These attributes can be decoded using constants defined in `tpm2-pkcs11`'s code [22] and for example the elliptic curve public key is stored in attribute `CKA_EC_POINT = 0x181`.

`tpm2-pkcs11` defines three *vendor attributes*: `CKA_TPM2_OBJAUTH_ENC`, `CKA_TPM2_PUB_BLOB` and `CKA_TPM2_PRIV_BLOB`. In the SQLite database used for tests, there are 2 objects:

— One with attribute 0 set to 3, which means that its `CKA_CLASS` is `CKO_PRIVATE_KEY`: it is a private key. This object also contains the three *vendor attributes* of `tpm2-pkcs11`.

21. This was observed by recording the system calls issued by the commands using `strace`. The commands did not interact with any device related to TPM.

22. `https://github.com/tpm2-software/tpm2-pkcs11/blob/1.5.0/tools/tpm2_pkcs11/pkcs11t.py#L39-L97`

— The other one with attribute 0 set to 2, which means that its `CKA_CLASS` is `CKO_PUBLIC_KEY`: it is a public key. This object only has `CKA_TPM2_PUB_BLOB` as *vendor attribute*.

For both objects, attribute `CKA_TPM2_PUB_BLOB` contains hexadecimal-encoded data which includes the elliptic curve public key. In fact, this attribute stores data encoded according to a structure which is defined in TPM 2.0 specification as `TPM2B_PUBLIC` (listing 11, from `https://github.com/stefanberger/libtpms/blob/v0.7.5/src/tpm2/TpmTypes.h#L1682-L1695`):

```
typedef struct {
    TPMI_ALG_PUBLIC          type;
    TPMI_ALG_HASH            nameAlg;
    TPMA_OBJECT              objectAttributes;
    TPM2B_DIGEST             authPolicy;
    TPMU_PUBLIC_PARMS        parameters;
    TPMU_PUBLIC_ID           unique;
} TPMT_PUBLIC;
typedef struct {
    UINT16                   size;
    TPMT_PUBLIC              publicArea;
} TPM2B_PUBLIC;
```

Listing 11. Structures `TPMT_PUBLIC` and `TPM2B_PUBLIC` from TPM 2.0 specification

For example, when the content of attribute `CKA_TPM2_PUB_BLOB` is (listing 12):

```
00560023000b000600720000001000100003001000203eef05ada9dc42f69ffc
a066adfc374ec94aaba63bfa9383c2a563d847f31ac2002050702adc8e1081d1
b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387
```

Listing 12. Example of generated public key blob

This content can be decoded as (listing 13):

```
struct TPM2B_PUBLIC {
  size = 0x0056,
  publicArea = {
    type = 0x0023, // = TPM_ALG_ECC
    nameAlg = 0x000b, // = TPM_ALG_SHA256
    objectAttributes = 0x00060072,
    authPolicy = { size = 0x0000 },
    parameters.eccDetail = {
      symmetric = 0x0010, // = TPM_ALG_NULL
      scheme = 0x0010, // = TPM_ALG_NULL
      curveID = 0x0003, // = TPM_ECC_NIST_P256
      kdf = 0x0010 // = TPM_ALG_NULL
    },
    unique.ecc = {
      x = {
```

```
          size = 0x0020,
          bytes = "3eef05ada9dc42f69ffca066adfc374e"
                  "c94aaba63bfa9383c2a563d847f31ac2"
        },
        y = {
          size = 0x0020,
          bytes = "50702adc8e1081d1b633a1e1d6278b46"
                  "13ba20cf5fd8af0b8c3c8b4a765b9387"
        }
      }
    }
}
```

Listing 13. Deserialization of an example of generated public key blob

So attribute `CKA_TPM2_PUB_BLOB` directly consists in the public key generated with `tpm2_ptool addkey`, serialized for the TPM. Does attribute `CKA_TPM2_PRIV_BLOB` directly contains the associated private key? The answer should of course be negative, and some further analysis was conducted in order to understand why something related to the private key is stored in the database.

3.2 Storage of the private key

From a functional point of view, a TPM only has a limited amount of persistent memory but it is able to use many keys. This is made possible because the private keys are stored outside of the TPM and are encrypted with a secret which never leaves the TPM. When a private key is used for example to perform some signing operations, the key first needs to be loaded into the TPM. The TPM decrypts the private key before loading it.

In practice, the encrypted private key is serialized with a structure defined in TPM 2.0 specification as `TPM2B_PRIVATE`, which only states "a size and some bytes". When using the TPM simulator `swtpm`, it is possible to retrieve the encryption key and to decrypt the private key.

In the tests, the content of attribute `CKA_TPM2_PRIV_BLOB` is (listing 14):

```
007e002093b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc
41cc01f50010627e422444e01671fe6b2e3a771634d64d64599bc3129fb57f10
2bb89244e6d7c6c029a9a53b27bddbb0ba5b5fa0497c3286364b50fce3757615
c895de4fce053c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c70908a8
```

Listing 14. Example of generated private key blob

This can be decoded as (listing 15):

```
struct TPM2B_PRIVATE {
  size = 0x007e,
  buffer = {
    integrity = {
      size = 0x0020,
      bytes = "93b2e33a7ff39879229e35afeb86ec61"
              "bca0aaee057c0d56bee354bc41cc01f5"
    },
    iv = {
      size = 0x0010,
      bytes = "627e422444e01671fe6b2e3a771634d6"
    },
    encrypted =
      "4d64599bc3129fb57f102bb89244e6d7c6c029a9a53b27bd"
      "dbb0ba5b5fa0497c3286364b50fce3757615c895de4fce05"
      "3c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c709"
      "08a8"
  }
}
```

Listing 15. Deserialization of an example of generated private key blob

In order to decrypt the data, the persistent storage of the software TPM needs to be analyzed. This storage is located in a file named `tpm2-00.permall` in the directory specified by option `-tpmstate` when launching `swtpm`. This file contains the public and sensitive structures (`TPMT_PUBLIC` and `TPMT_SENSITIVE` in TPM specification) related to the SRK (Storage Root Key) used by `tpm2-pkcs11` and defined by handle `0x81000000`. A sensitive structure contains the following fields (listing 16):

```
typedef struct {
    TPMI_ALG_PUBLIC              sensitiveType;
    TPM2B_AUTH                   authValue;
    TPM2B_DIGEST                 seedValue;
    TPMU_SENSITIVE_COMPOSITE     sensitive;
} TPMT_SENSITIVE;
```

Listing 16. Structure `TPMT_SENSITIVE` from TPM 2.0 specification

In the file used in the tests, the content of `seedValue` is in hexadecimal (listing 17):

```
07f5b590a03d66e2225274698323ccfe59a7356e9cc14436091fe9d49b3e577c
```

Listing 17. `seedValue` of the SRK used in tests

Using this value, it is possible to derive a HMAC key and an AES key, to verify the integrity tag and to decrypt the data.

Here is a Python 3.8 session showing how to compute those values (listing 18):

```
>>> import hashlib, hmac

>>> pub_blob = bytes.fromhex("""
... 005600230000b000600720000001000100003001000203eef05ada9dc42f69ffc
... a066adfc374ec94aaba63bfa9383c2a563d847f31ac2002050702adc8e1081d1
... b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387
... """)
>>> priv_blob = bytes.fromhex("""
... 007e002093b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc
... 41cc01f50010627e422444e01671fe6b2e3a771634d64d64599bc3129fb57f10
... 2bb89244e6d7c6c029a9a53b27bddbb0ba5b5fa0497c3286364b50fce3757615
... c895de4fce053c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c70908a8
... """)
>>> srk_seed = bytes.fromhex("""
... 07f5b590a03d66e2225274698323ccfe59a7356e9cc14436091fe9d49b3e577c
... """)

# Compute the public name, prefixed by TPM_ALG_SHA256 = 0x000b
>>> pub_name = b'\x00\x0b' + hashlib.sha256(pub_blob[2:]).digest()
>>> pub_name.hex()
'000bcac322c64b1a31d7806bc84570090949f898cea8c2c9a258761659dfb1de'
'713d'

# Compute HMAC key with KDFa
>>> hashstate = hmac.new(srk_seed, None, "sha256")
>>> hashstate.update(int.to_bytes(1, 4, "big"))  # counter
>>> hashstate.update(b'INTEGRITY\0')  # label
>>> hashstate.update(int.to_bytes(256, 4, "big"))  # sizeInBits
>>> hmac_key = hashstate.digest()
>>> hmac_key.hex()
'7f861102ab2854de213e6ffa9ae2a2521c73abf49b697618736d85615d27389b'

# Compute the integrity tag
>>> hashstate = hmac.new(hmac_key, None, "sha256")
>>> hashstate.update(priv_blob[0x24:])
>>> hashstate.update(pub_name)
>>> computed_integrity = hashstate.digest()
>>> computed_integrity.hex()
'93b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc41cc01f5'

# Check the integrity
>>> computed_integrity == priv_blob[4:0x24]
True

# Compute the AES key with KDFa
>>> hashstate = hmac.new(srk_seed, None, "sha256")
>>> hashstate.update(int.to_bytes(1, 4, "big"))  # counter
>>> hashstate.update(b'STORAGE\0')  # label
>>> hashstate.update(pub_name)  # contextU = name
>>> hashstate.update(int.to_bytes(128, 4, "big"))  # sizeInBits
>>> aes_key = hashstate.digest()[:16]
>>> aes_key.hex()
'9052599459c554ee409ffdba6311b2ce'

# Decrypt private blob using library cryptography.io
>>> from cryptography.hazmat.primitives.ciphers import \
...   Cipher, algorithms, modes
```

```
>>> from cryptography.hazmat.backends import default_backend
>>> iv = priv_blob[0x26:0x36]
>>> cipher = Cipher(algorithms.AES(aes_key), modes.CFB(iv),
...   backend=default_backend())
>>> sensitive = cipher.decryptor().update(priv_blob[0x36:])
>>> sensitive.hex()
'004800230002030363431326236376166633837633037656261323663346530 31'
'6165366235300000020e136a90d627a7b2ea404ed671a7717cb04b13f54f9df'
'478ff54ced6fd3275048'
```

Listing 18. Python session which decrypts a private key blob using the `seedValue` of the SRK and the *public name* associated with the key

The decrypted sensitive structure can be decoded as (listing 19):

```
struct TPM2B_SENSITIVE {
  size = 0x0048,
  sensitiveArea = {
    sensitiveType = 0x0023, // = TPM_ALG_ECC
    authValue = {
      size = 0x0020,
      buffer = "3036343132623637616663383763303765626132366334653031"
               "6562613236633465303161653662353 0"
    },
    seedValue = { size = 0x0000 },
    sensitive.ecc = {
      size = 0x0020,
      buffer = "e136a90d627a7b2ea404ed671a7717cb"
               "04b13f54f9df478ff54ced6fd3275048"
    }
  }
}
```

Listing 19. Deserialization of the decryption of a generated private key blob

The last buffer, `sensitive.ecc`, contains the private key associated with the elliptic curve public key (listing 20):

```
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.backends import default_backend
>>> sensitive_ecc_buffer = bytes.fromhex(
...  "e136a90d627a7b2ea404ed671a7717cb"
...  "04b13f54f9df478ff54ced6fd3275048")
>>> privkey = ec.derive_private_key(
...  int.from_bytes(sensitive_ecc_buffer, "big"),
...  curve=ec.SECP256R1(),
...  backend=default_backend())
>>> pubkey = privkey.public_key()
>>> hex(pubkey.public_numbers().x)
'0x3eef05ada9dc42f69ffca066adfc374ec94aaba63bfa9383c2a563d847f31ac2'
>>> hex(pubkey.public_numbers().y)
'0x50702adc8e1081d1b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387'
```

Listing 20. Python session which computes the public key associated with the recovered private key

This confirms that `tpm2-pkcs11`'s database contains an encrypted version of the private key, stored in attribute `CKA_TPM2_PRIV_BLOB` of the PKCS#11 object associated with the private key. This attribute is encrypted using the `seedValue` of the used SRK, which is a secret supposed to never leave the TPM. Therefore this analysis also confirms that only the TPM itself can decrypt this attribute.

Now, there is something strange with this analysis: neither the user PIN nor the SOPIN were used to decrypt the private key. And indeed they are not needed to load the key (listing 21):

```
# Load the key with file "pub_blob" containing the content
# of CKA_TPM2_PUB_BLOB and "priv_blob" the content of
# CKA_TPM2_PRIV_BLOB
$ tpm2_load -c /tmp/context -C 0x81000000 \
    -u pub_blob -r priv_blob
name: 000bcac322c64b1a31d7806bc84570090949f898cea8c2c9a2587
61659dfb1de713d
```

Listing 21. Loading private key blob and public key blob in the TPM

But using this key does not directly work to sign data (listing 22):

```
$ echo hello | tpm2_sign -c /tmp/context \
    -g sha256 -s ecdsa -o signature.out
WARNING:esys:src/tss2-esys/api/Esys_Sign.c:311:Esys_Sign_Finish()
Received TPM Error
ERROR:esys:src/tss2-esys/api/Esys_Sign.c:105:Esys_Sign()
Esys Finish ErrorCode (0x0000098e)
ERROR: Eys_Sign(0x98E) - tpm:session(1):the authorization
HMAC check failed and DA counter incremented
ERROR: Unable to run tpm2_sign
```

Listing 22. Trying to use the key to sign a message produces errors

The error suggests an authentication failure, with the DA counter (Dictionary Attack) of the TPM being incremented.[23]

In the decrypted sensitive structure associated with the private key (listing 19), the `authValue` contains 32 bytes which are represented in hexadecimal. In practice these bytes consist in 32 hexadecimal characters: `06412b67afc87c07eba26c4e01ae6b50`. This value can be directly used with command `tpm2_sign` to sign a message without any error (listing 23):

```
$ echo hello | tpm2_sign -c /tmp/context \
    -g sha256 -s ecdsa -o signature \
    -p 06412b67afc87c07eba26c4e01ae6b50
```

23. The Dictionary Attack counter is a mechanism which prevents brute-force attacks on TPM. After some number of authentication failures, the TPM becomes locked and rejects any further authentication try.

```
$ xxd -p -c32 signature
0018000b00201f076fa127366b9d9cc36155652751545115e4ce35749ed75638
7e68f058d35d00203bd83b9086a7876948fcc4728c4141b30a0fe94cada03147
76052933888802a8

# Verifying the signature does not require the authValue
$ echo hello > msg
$ tpm2_verifysignature -c /tmp/context -s signature -m msg
```

Listing 23. Trying to use the key to sign a message with the `authValue` (parameter `-p` succeeds

Therefore the private key generated with `tpm2_ptool addkey` is protected by an authorization value. In the presented tests, this authorization value was retrieved by decrypting the private blob exported by the TPM. Doing so was possible only because a software TPM was used and the decryption key could be retrieved. With a hardware TPM, this should not be possible. There should be another way to retrieve it, `tpm2-pkcs11` needs to be able to use the key.

4 Linking the PIN of the PKCS#11 token with the authorization value of the key

4.1 Unsealing a wrapping key from the PIN or the SOPIN

The previous section presented that the private SSH key generated with `tpm2_ptool addkey` was stored in a PKCS#11 attribute (named `CKA_TPM2_PRIV_BLOB`) in table `tobjects` of `tpm2-pkcs11`'s SQLite database. This private key was (of course) encrypted by the TPM and in order to use it, the software has to provide an authorization value to the TPM.

Taking a step back, the PIN and the SOPIN should be linked to this authorization value: both are some kind of secret that the user is required to enter in order to use the key. But the PIN and the SOPIN are two independent secrets: the PIN can be used without the SOPIN when using the key and if the PIN is forgotten, the SOPIN can be used to reset the PIN.

In order to help understanding how this works, `tpm2-pkcs11` provides a command, `tpm2_ptool verify`. This command checks the PIN or the SOPIN (or both) and displays some hexadecimal values such as `seal-auth` and `wrappingkey`. None of these values match the authorization value which was found in the previous section. While investigating why, the author of this document found a bug in the Python code of the tool (a variable was not initialized when some options were provided) and fixed

it.[24] But this fix did not change the fact that `tpm2_ptool verify` did not show the authorization value, so it was necessary to dig a little bit more.

The SQLite database used by `tpm2-pkcs11` includes a table named `sealobjects` which is used to store information for the PIN and SOPIN (listing 24):

```
$ sqlite3 "$HOME/.tpm2_pkcs11/tpm2_pkcs11.sqlite3"
sqlite> .dump sealobjects
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE sealobjects(
    id INTEGER PRIMARY KEY,
    tokid INTEGER NOT NULL,
    userpub BLOB,
    userpriv BLOB,
    userauthsalt TEXT,
    sopub BLOB NOT NULL,
    sopriv BLOB NOT NULL,
    soauthsalt TEXT NOT NULL,
    FOREIGN KEY (tokid) REFERENCES tokens(id) ON DELETE CASCADE
);
INSERT INTO sealobjects VALUES(1,1,
X'002e0008000b000000520000001000020b0c383025b2418e95f530707ba7f28
a29b4bf55d65f004c8365c68400ae3cc60',
X'00be0020835e6bbbc97ff76714b0b9cc7352d823cc250741ecb2817c7ad28b
44d958cfc3001084d9eb99781ba29b9e2dfc601ae5bec4fdcbce5055be161244
5f67e390b54328ae4b47f126746393ba7dcc9dc7b93b766f761473d68d581dfd
aed3d6a365ce9bb90d7d2cb1118363f4416b1770dbdbfa726b480c760f113b69
6556b064ebce1b05ac8d80511c83f753f5aeb342257b5b561ba746dc2ccafd0f
5e2824c3f7838c235115b75d1665c7938a0a50999990a1399194ee9aa0eb03f8
36',
X'3732383265363234656164316465633161376165353938623436306565633
6335663736633730646562623234663335376664613531313437653937333365
34',
X'002e0008000b000000520000001000020df41af69b73d88f829c60fe0e27687
62f43be4e831cc0a5d0af5508b1752cecf',
X'00be002013579162beec11b58bbd5ac9d4db3b1f2de8a70f276f75b1925111
06ad76ff100010f937771c1214098ad9a19d49a211757f2d1f9d48195624e87c
526ad8ccb229479a474aa7ba3b010058ad64f33560aad3529c6e4a1c10092304
5cddd249fec9d565bb037712ffc267c9837d8ca561f6d720d84ddf019dc8fe45
c059e34dc20b258f0f2c959aca09cb580eadecc1f3fdae44587d51f9028f50b4
6b9e7007538751508d49f52a21426357c72671f4915562ab9cd7b18cb28a77ac
6c',
X'626332353331636434346334663331663132396634376137386366643338343
656363653836353334366263393662326363323962313530626230646264636
238');
```

Listing 24. Content of `sealobjects` database in `tpm2-pkcs11` database

Columns `userpub` and `sopub` both store a `TPM2B_PUBLIC` structure (listing 25):

24. https://github.com/tpm2-software/tpm2-pkcs11/pull/635

```
struct TPM2B_PUBLIC {
  size = 0x002e,
  publicArea = {
    type = 0x0008, // = TPM_ALG_KEYEDHASH
    nameAlg = 0x000b, // = TPM_ALG_SHA256
    objectAttributes = 0x00000052,
    authPolicy = { size = 0x0000 },
    parameters.keyedHashDetail = {
      scheme = 0x0010 // = TPM_ALG_NULL
    },
    unique.keyedHash = {
      size = 0x0020,
      bytes = "..."
    }
  }
}
```

Listing 25. Extract of the deserialization of the content of `userpub` and `sopub` columns

This structure is a *Keyed Hash* object, which is in theory a way to store a secret key to perform a HMAC-based authentication. In practice, this object is used here to store a *sealed* secret (it cannot be used as a HMAC authentication because `parameters.keyedHashDetail.scheme` is `TPM_ALG_NULL`), in the private parts which are in columns `userpriv` and `sopriv`. In order to *unseal* the secret, the public and private parts need to be loaded in the TPM, and then `tpm2_unseal` can be used with an authorization value derived from the PIN (when using `userpub`/`userpriv`) or the SOPIN (when using `sopub`/`sopriv`). This authorization value only consists in the SHA256 digest of the PIN concatenated with the value in column `userauthsalt` or `soauthsalt`, truncated to 16 bytes (listings 26 and 27):

```
>>> from hashlib import sha256
>>> userpin = b"XXXX"
>>> userauthsalt = bytes.fromhex("""
... 373238326536323465616431646563316137616535393862343630656563663
... 356637366337306465626232346633353766646135313134376539373333653 4
... """)
>>> usersealauth = sha256(userpin + userauthsalt).digest()[:16]
>>> usersealauth.hex()
'c3402eac59ae76a86317d9b34811ca3d'
```

Listing 26. Python session which computes the authorization value of the `userpub` field, from the user PIN and `userauthsalt`

```
$ tpm2_load -c /tmp/context -C 0x81000000 \
    -u userpub -r userpriv
$ tpm2_unseal -c /tmp/context -p c3402eac59ae76a86317d9b34811ca3d
```

```
9d7854e46e3e8816070697c9770fbc2ed4297dea669bebbe2c22a52421df63cd
```

Listing 27. Using the derived authorization value to unseal the `userpriv` field

If the authorization value is wrong or missing, `tpm2_unseal` fails and returns the error message "the authorization HMAC check failed and DA counter incremented". This indicates that the *sealed* secret is protected by the TPM's Dictionary Attack counter which prevents brute-force attacks.

Moreover, repeating the commands with the SOPIN and its associated fields leads to the same secret being *unsealed*. This secret (encoded in hexadecimal) is also displayed by command `tpm2_ptool verify` using the name *wrappingkey*.

4.2 Decrypting an authorization value from the wrapping key

The previous sections presented that:

— the SSH key generated with `tpm2_ptool addkey` was protected by an authorization value,

— and the PIN and or SOPIN of a *token* enabled *unsealing* a *wrapping key* which was not this authorization value.

Something is missing to establish a link between *wrapping key* and the authorization value. Studying how `tpm2-pkcs11` works enabled bridging the gap: the *wrapping key* is an AES key which is used to encrypt the authorization value using AES-GCM. The encrypted value and the parameters of the GCM mode (a nonce and a tag) are stored in attribute `CKA_TPM2_OBJAUTH_ENC` of each private key, in table `tobjects` of the SQLite database.

In the tests, this attribute contains (listing 28):

```
3638333330346435336435306134376466356666383530333a33313632626439
3666393134316564633232656138366636646661373964666653a663036323866
3066353661373637656261316361393631376239396234613162643561653666
3662653863323664623966383932373765666531393137343234
```

Listing 28. Attribute `CKA_TPM2_OBJAUTH_ENC` of the generated SSH key

Here is a Python 3.8 session showing how to decode this data, using the wrapping key unsealed in listing 27 (listing 29):

```
>>> from cryptography.hazmat.primitives.ciphers.aead import AESGCM
>>> objauth_enc_unhex = bytes.fromhex("""
... 3638333330346435336435306134376466356666383530333a33313632626439
... 3666393134316564633232656138366636646661373964666653a663036323866
... 3066353661373637656261316361393631376239396234613162643561653666
... 3662653863323664623966383932373765666531393137343234
```

```
... """)
>>> fields = objauth_enc_unhex.decode("ascii").split(":")
>>> nonce, tag, ciphertext = map(bytes.fromhex, fields)

>>> wrappingkey = bytes.fromhex("""
... 9d7854e46e3e8816070697c9770fbc2ed4297dea669bebbe2c22a52421df63cd
... """)
>>> aesgcm = AESGCM(wrappingkey)
>>> authval = aesgcm.decrypt(nonce, ciphertext + tag, b'')
>>> authval
b'06412b67afc87c07eba26c4e01ae6b50'
```

Listing 29. Python session which computes the authorization value of a key from the wrapping key

This last value is indeed the authorization value embedded in the sensitive structure of the private key which was generated (listing 19).

In short, the PIN and the SOPIN are used by `tpm2-pkcs11` to *unseal* an AES key which is used to decrypt (without using the TPM) the authorization value which is necessary to use keys, for example to sign data or to perform SSH authentication.

5 Conclusion

When using `tpm2-pkcs11` 1.5.0 to generate and handle SSH keys with a TPM 2.0, the software never sees the private keys. They are exported by the TPM in *private blobs* that `tpm2-pkcs11` stores in a database. To use these keys, an authorization value needs to be provided to the TPM and `tpm2-pkcs11` stores an encrypted copy of this value which relies on a complex derivation scheme.

This enables to answer the questions which were asked in the introduction:

- Is stealing the SQLite database enough to impersonate the user? No, as the keys it contains are encrypted using the TPM's SRK.
- Is there any software (which could be compromised) that sees the private key when the user uses it to connect to a server? No.
- How is the TPM actually used to authenticate the user? The key is loaded into the TPM, the software computes an authorization value using the PIN or the SOPIN. It then requests the TPM to use the key to perform a signature operation used in the authentication protocol.

Moreover, even though this article focused on SSH, the software stack provides a compatibility layer with many other protocols through a PKCS#11 interface. For example this enables transposing the questions

and their answers to VPN software (Virtual Private Network), TLS stacks (Transport Layer Security), etc.

Knowing how `tpm2-pkcs11` works in detail would enable assessing its security. This could be done in some future work.

References

1. Jeremy Boone. TPM genie: Attacking the hardware root of trust for less than $50. CanSecWest, 2018. `https://cansecwest.com/csw18archive.html`.
2. Aurélien Bordes. Bitlocker. SSTIC, June 2011. `https://www.sstic.org/2011/presentation/bitlocker/`.
3. James Bottomley. TPM enabling the crypto ecosystem for enhanced security. Kernel Recipes, September 2018. `https://kernel-recipes.org/en/2018/talks/tpm-enabling-the-crypto-ecosystem-for-enhanced-security/`.
4. Andreas Fuchs. Introducing TPM NV storage with E/A policies and TSS-FAPI. Linux Security Europe, November 2020. `https://www.youtube.com/watch?v=JckONn4h6pQ`.
5. Nicolas Iooss. Représenter l'arborescence matérielle. SSTIC Rump Sessions, June 2018. `https://www.sstic.org/2018/presentation/2018_rumps/`.
6. Microsoft. Minimum hardware requirements, section 3.7 trusted platform module (TPM), 2016. `https://docs.microsoft.com/en-us/windows-hardware/design/minimum/minimum-hardware-requirements-overview#37-trusted-platform-module-tpm`.
7. Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL:TPM meets timing and lattice attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073, 2020. `https://tpm.fail`, CVE-2019-11090 and CVE-2019-16863.
8. Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1648, 2017.
9. Henri Nurmi. Sniff, there leaks my bitlocker key, December 2020. `https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/`.
10. Bernard Ourghanlian. L'implémentation des spécifications du TCG au sein de la plateforme windows : un aperçu de bitlocker. SSTIC, June 2006. `https://www.sstic.org/2006/presentation/Les_evolutions_de_l_implementation_des_specifications_du_TCG_au_sein_de_la_plateforme_Windows/`.
11. Christopher Tarnovsky. Deconstructing a 'secure' processor. Black Hat DC, 2010. `https://www.youtube.com/watch?v=62DGIUpscnY`.