Introduction to Trusted Computing

Goals of this lab:

- Get hands-on experience working with hardware for trusted computing
- Understand basic principles for implementing security solutions based on the TPM hardware

Table of Contents

MAIN LAB

In this lab you will implement a simple login token for an e-banking service, using a (simulated) *Trusted Platform Module* (TPM). Both the simulated bank token and the bank server will run as Apptainer containers locally on your SU-room computer. It is also possible to do the lab on ThinLinc.

Solutions to the assignments in the lab will be implemented using command-line tools from the tpm2-tools package[i]. You shouldn't need to resort to any "real" shell-script programming (conditional statements, loops, etc.). Simply stringing together inputs and outputs from different commands using intermediate files is sufficient to complete the lab. However, it is expected that you know how to use a Unix-like operating system from the command line.

Common TPM commands

Here we summarize a few commonly-used tpm2-tools commands.

Key creation
`tpm2_createprimary` – create a primary key
`tpm2_create` – create a child key
`tpm2_import` – import an existing key

Key storage
`tpm2_load` – load key into volatile RAM
`tpm2_evictcontrol` – persist key to NVRAM

Reading PCRs
`tpm2_pcrread` – read contents of PCRs
`tpm2_quote` – create signed digest of PCRs

Authorization policy management
`tpm2_startauthsession` – start an authorization session
`tpm2_policypcr` – record/specify PCR values for an authorization session
`tpm2_flushcontext` – remove/unload objects or sessions from TPM memory

Common crypto operations
`tpm2_encryptdecrypt` – symmetric encrypt/decrypt
`tpm2_rsaencrypt` – RSA encrypt
`tpm2_rsadecrypt` – RSA decrypt
`tpm2_hash` – compute hash
`tpm2_hmac` – compute HMAC
`tpm2_sign` – sign a message/digest
`tpm2_verifysignature` – verify a signature

[i] https://github.com/tpm2-software/tpm2-tools

Part 1: The lab environment

Setting up the lab directory structure

To set up the files needed for the lab, simply run the following command in a terminal:
/courses/TDDE62/lab/tc/setup_tc_lab.sh

This will create the lab directory in your home folder: /home/<*liu-id*>/tdde62/tc

This directory in turn contains the following two directories:

- **server** contains files needed for the bank login server. You don't need to edit these files. They are just provided for reference.
- **token** represents a part of the file system for the simulated bank token. This directory contains the files you will need to edit in order to complete the lab, in the form of two shell scripts. It will also hold files created by the tpm2-tools, and a file representing the contents of the simulated TPM's NVRAM.

Starting the web server

To start the web server for the bank login, run the following script:
/courses/TDDE62/lab/tc/run_server.sh

The server will print the URL for accessing the bank website during startup. It will look something like this: `http://localhost:`*`NNN`*, where *NNN* is a randomly generated port number. Hold **Ctrl** and click the URL to launch a web browser.

The web site implements a simple challenge-response scheme for logging in and checking one's balance. For simplicity's sake, the customer database is implemented as a single JSON-file called **db.json**, located in the **server** directory. Enter a user ID (a four-digit key from the JSON file) to start the login process for the corresponding user. A challenge (an 8-digit number) is presented. This challenge must be fed into the token to get the correct response (also 8 digits) for logging in.

To shut down the web server when you're done working on the lab, use **Ctrl** + **C**

> Note that, in order to keep complexity down, and to make debugging of lab solutions easier, the web server is completely stateless. Moreover, challenges never time out. These would be fatal security flaws in a real-life setting, as they would allow unlimited replay attacks in cases where an attacker can eavesdrop on challenge/response pairs.

Using the token

The token directory contains three shell-script files: **init.sh**, **response.sh**, and **util.sh**. The two former files must be edited in order to complete the lab, while the latter file contains utility functions. When the token has been executed at least once, this directory will also contain a file called **NVChip**, holding the contents of NVRAM for the simulated TPM. In a real-life setting, this data would of course be stored securely inside an actual hardware TPM.

A script is provided for running the simulated token:
/courses/TDDE62/lab/tc/run_token.sh

The script changes the working directory to the **token** directory, starts up the simulated TPM, and then performs one of the following three actions, depending on what command-line arguments are given:

- **run_token.sh --init *S*** is used to initialize the token. In a real-life setting, this script would only be used once by the *bank* before a new token is shipped to the customer. ***S*** here denotes a shared secret used to compute responses. It must be a hex-encoding of a 64-bit binary key. You should simply copy-paste the `shared_secret` field of a user in **db.json** when using the --init mode. This mode will first *reset the TPM state* by deleting the **NVChip** file, and then execute the **init.sh** script.

- **run_token.sh *C*** computes a response for the challenge ***C***. The token must have already been initialized with a shared secret. This mode represents the interface for the *end user* of the token, and executes the script **response.sh**.
- **run_token.sh --shell** launches a command-line shell inside the token container. This is meant to be used as a "playground" for testing tpm2-tools commands. It is recommended that you use this mode to figure out the correct sequence of commands to solve a lab assignment, before putting the commands in the script files.

Hint 1: If you wish to reset the TPM state after having done work in the --shell mode, you can manually delete the **NVChip** file. Note that this will re-generate the owner seed and invalidate all previously generated keys. It is probably also a good idea to clean up old files created by tpm2-tools in the **token** directory before you test your finalized scripts, in order to avoid potential confusion.

Hint 2: At some point, you will probably want to use the utility functions while working in the --shell mode. Execute the command `source util.sh` to import the utility functions into the current shell.

Hint 3: While working on the command-line, you can navigate the command history with the up and down arrows on the keyboard. You can also perform a backwards search in the command history by pressing **Ctrl + R** and typing part of a sought-after command.

Hint 4: You can copy-paste from/to the terminal or other places by simply highlighting text (to copy it implicitly) and then pressing the *middle mouse button*/*scroll wheel* to paste it. Alternatively, you can right click and select Copy, and then right click and select Paste at the destination.

While the token is running, a subdirectory called **scratch** will also be available in the **token** directory. The **scratch** directory will be automatically deleted once the token shuts down, and is supposed to be used for temporary files. (I.e., intermediate files that are only used during a single interaction with the token.)

Part 2: The assignments

Your goal in this lab is to implement the token in such a way that the shared secret installed in the token *cannot be extracted by an attacker*, even if the attacker has physical access to the device or is able to install malicious software on it. (I.e., we want to prevent cloning of tokens.)

Assignment 1: Basic functionality

2.1.1 Inspect the response computation in the file **check_response.php** in the **server** directory. How are responses computed from the shared secret and the challenge? (You don't need to worry about exactly how the raw binary response is transformed to an 8-digit number. This step is already implemented for you in the **utils.sh** file.)

2.1.2 Your first task is to design the procedure for installing and using the shared secret in such a way that it cannot be extracted by attackers. (I.e., it should be protected by the TPM.) Your solution should consider the keys and TPM commands involved. (It is recommended that you work on the *owner hierarchy* only.)

We recommended you to show your design to the lab assistant before you proceed, to make sure that you are on the right track.

2.1.3 Your next task is to implement the installation of the shared secret in the **init.sh** script, and the computation of responses in **response.sh**. Keep files needed for the continuous operation of the token (created during the initialization phase) in the **token** directory, but use the temporary **scratch** directory for intermediate files only used during a single interaction with the token.

2.1.4 Test that your solution works (i.e., that you can successfully "log in" at the bank).

Hint: The tpm2-tools will generally print information about what they are doing. This might be helpful when you're working in the --shell mode, but will clutter the screen when executing the initialization or response scripts. Use the command line option **-Q** to turn off tool printouts. (Error messages will still be printed.)

Report: Describe your design from step 2.1.2, how it satisfies the requirement of disallowing device cloning, and how your implementation in step 2.1.3 relates to it. Also include the contents of the **init.sh** and **response.sh** scripts.

Assignment 2: Ensuring code integrity

2.2.1 Currently, your design should protect against device cloning. However, it would still be possible to eavesdrop on challenge/response pairs, if an attacker is able to install spyware on the device. In this assignment you will extend the implementation to *prevent responses from being computed if the software in the token has been manipulated*.

In a real-life setting, we would be verifying the entire software stack, all the way from the firmware/BIOS up to the OS and application code. For the purpose of this lab, we only consider the integrity of the **response.sh** script. Concretely, the token will extend **PCR 15** of the **SHA256** bank with a digest of **response.sh** during startup. Note that you **don't** need to implement the PCR extend yourselves. (It is already done behind the scenes before the **init.sh**/**response.sh** scripts start to execute.)

2.2.2 Improve your design from step 2.1.2, so that the TPM no longer allows computing responses if PCR 15 of the SHA256 bank has changed since the token was initialized with the --init mode. Again, it is recommended to check your design with the lab assistant before proceeding.

2.2.3 Implement your design from the step above.

2.2.4 Verify that your solution still works as expected, and that it stops working if you make any changes to **response.sh** after initialization. (Just adding a blank line to the script is sufficient for testing this.)

Report: Describe your new design and how you have implemented it. Include the updated contents of **init.sh** and **response.sh** in the report.