LTH

ADVANCED COMPUTER SECURITY

EITN50

Project 3: TPM

Group [redacted]

Authors:

[redacted]

Date: October 3, 2017

Rapporten är eget orginal arbete och källor av all icke egen material är angiven.

Contents

1 Introduction

The TPM, Trusted Platform Module, is an independent chip whose primary task is to provide cryptographic functions to the computer it is attached to. It implements several security functions that increase the trustworthiness of the (processing) platform, and enables things like verified or measured boot and secure storage of sensitive data.

This chip is usually soldered to the motherboard, and set up by the manufacturer. The reason for this is that the manufacturer, most likely, is a member of the TCP (Trusted Computing Group) and can therefore provide a signature of authenticity. For this project we will emulate a TPM, and will not worry about the authenticity of the root key. The goal here is to get a better understanding of how this platform works, and its limitations.

The project was based on the project description version 2017-09-17[1].

2 Assignments

2.1 Assignment 1: Setting Up the Environment

For this project we utilized the provided virtual machines (TPM1, TPM2 and TSS), and these were placed inside a local folder on our computer. The TSS machine will act as our Trusted Computer with the remote TPM module provided, to begin with, by the TPM1 machine.

We started by booting up the TSS (username: "tss", password: "lab") and the TPM1 (username: "pi", password: "tpm") virtual machines. We made sure the machines could communicate with each other by issuing the ping command.

```
tss@TSS ~ ping 10.0.2.14
PING 10.0.2.14 (10.0.2.14) 56(84) bytes of data.
64 bytes from 10.0.2.14: icmp_seq=1 ttl=64 time=0.486 ms
```

We enabled the TPM service by setting the correct environment variables.

```
pi@TMP1 ~ env | grep TPM
TPM_PATH=/home/pi/tpm/tpm4720/tpmstate
TPM_PORT=6545
TPM_SERVER_NAME=localhost
TPM_SERVER_PORT=6545
```

Similarly we must make sure that the main TSS machine also has the correct configuration.

```
tss@TSS ~ env | grep "TPM\|TCSD"
TCSD_USE_TCP_DEVICE=true
TCSD_TCP_DEVICE_PORT=6545
TCSD_TCP_DEVICE_HOSTNAME=10.0.2.14
TPM_SERVER_NAME=10.0.2.14
TPM_SERVER_PORT=6545
```

With the environments set, both TPM machines should be empty at first, as in they have not been initialized and does not have any keys loaded. However, as a precaution we checked that the folder

```
pi@TPM1 ~/tpm/tpm4720/tpmstate
```

was empty before proceeding. The file "`00.permall`" is what holds the TPM state and we did not need any resets but it could be good to know if any problems arise during the setup. Nevertheless, with the configuration complete we are now ready to start the remote TPM server.

```
pi@TPM1 ~ tpm_server
main: Initializing TPM at Mon Sep 25 14:15:44 2017
main: Compiled for latest revision specLevel 0002 errataRev 03
main: Compiled svn version 4720 revMajor 12 revMinor 70
... a lot more printouts ...
```

This will start a TCL server from which the TSS machine can access the remote TPM data. If any problem arises during the setup steps, make sure the IP addresses on the TSS machine point to TPM1's address.

2.2 Assignment 2: Getting the TPM Ready for Use

To get the TPM working on a normal computer you first need to activate it in BIOS, but on the virtual machines provided this worked slightly differently. Normally the TPM has the EK (Endorsement Key) key pair and a certificate for it. These should be provided by the manufacturer of the trusted hardware, but if they aren't provided they need to be created and certified as the second step in the TPM provisioning. However, for our purpose we ignore the EK certificate.

With the virtual machines we emulate this procedure by first activating the TPM in BIOS, which in our case is only a terminal command which does not provide any output. This does a `TPM_Startup`, to activate the TPM, which resets the PCR registers or takes them from a previously saved state.

```
tss@TSS ~ tpmbios
```

Next we establish a connection from the TSS machine to the TPM emulator machine TPM1. This will start a daemon process and we will continue issuing commands in a separate terminal window until we change to another TPM.

```
tss@TSS ~ sudo -E /usr/local/sbin/tcsd -e -f
TCSD TDDL ioctl: (25) Inappropriate ioctl for device
TCSD TDDL Falling back to Read/Write device support.
TCSD trousers 0.3.13: TCSD up and running.
```

After opening another terminal, we issue the command to actually create the EK key pair (normally provided by the manufacturer).

```
tss@TSS ~ createek
```

Observing the console output on the TPM1 machine you will actually see the public part of the EK key printed there. The private part should never be disclosed by the TPM to preserve its integrity.

```
... EK key printed on pi@TPM1 ...
TPM_IO_Write: length 314
 00 C4 00 00 01 3A 00 00 00 00 00 00 00 01 00 03
 00 01 00 00 00 0C 00 00 08 00 00 00 00 02 00 00
 00 00 00 00 01 00 C7 40 18 41 A1 CF C7 0B 92 7D
 C8 66 14 82 DC 84 A9 09 8A D0 3D 7F 6D AA 51 68
 5E 20 33 56 4B 5E 2A 57 62 24 12 B3 BD BB 12 8B
 24 24 A7 D5 96 AE D2 EB D4 BC A1 E0 3A 91 15 6D
 2C 5A 5C B8 B4 49 89 35 44 DB C0 87 1A 78 82 E6
 3D BD BA 6B 8B B5 B2 63 14 64 FB BD 6B 98 60 85
 D8 6E 7D CD CE D5 F9 FF 10 DC 2A 51 E8 A9 97 B4
 4E 70 47 92 A6 1E 0E 44 00 05 89 61 F9 F3 B3 6E
 02 7D 14 53 5F 9E 9B E8 8A 3B 45 5E 54 BC F9 47
 03 51 B8 B0 F7 56 DD 86 7A 5F 16 09 72 93 68 02
 D8 E0 BA 3B 12 60 C2 42 C7 EA B0 67 46 78 FD 24
 0A 74 C5 D9 A9 77 27 C6 12 6F E0 5D 5B 6A 5E 60
 7B F1 03 64 9A D0 E5 3A B6 47 0A AB 03 2D 4C 32
 B5 0E 4C 3B B8 A6 A1 7C 51 72 90 5A 90 A6 59 A2
 4B 85 EF 46 7A 9A 34 99 84 ED 95 AE FB 96 E4 8B
 A2 3B 3E 10 25 E3 49 CE D3 26 8A 49 82 87 AA AD
 3B 95 5A 17 86 65 7B 6C EB F8 18 2F 2E FC 7F F0
 CE 1E 45 BE C0 22 F7 AB 1E FA
TPM_IO_Connect: Waiting for connections on port 6545
```

The next step is to take ownership of the TPM. It is in this step where the SRK (Storage Root Key) is created. Two passwords have to be set in this step: one for the owner of the TPM and one for the SRK. The owner password is normally needed when, for example, the TPM settings are changed and the SRK password is required when the SRK is used to store other keys.

Note: Memorize both passwords since we will be needing them later. This command gives no output.

```
tss@TSS ~ takeown -pwdo ooo -pwds sss
```

Next we acquire the public key in `SRK.pub` format (not PEM) by issuing the appropriate command. This will place the file "`SRK.pub`" in the current working directory. We suggest that you create a separate folder for TPM1 to wok in since there will be a lot of key pairs created in the future.

```
tss@TSS ~/tpm1/keys ownerreadinternalpub -v -hk 40000000 -pwdo ooo -of SRK.pub
TPM_Send: OIAP
TPM_TransmitSocket: To TPM [OIAP] length=10
00 C1 00 00 00 0A 00 00 00 0A
TPM_ReceiveSocket: From TPM length=34
... a lot data being sent and received ...
```

Below is a hexdump of the SRK.pub key. This was obtained by running the following command.

```
tss@TSS ~/tpm1/keys xxd -g 1 SRK.pub
```

Which gives the following output.

```
00 00 00 01 00 03 00 01 00 00 00 0c 00 00 08 00
00 00 00 02 00 00 00 00 00 00 01 00 c1 5d d0 63
be ad 21 d5 df 0a 2d 19 65 a0 2d 4e 75 5a 7d 6b
42 1b ea 1d 3a 8b 86 07 07 99 7d 23 11 5c fb 6f
3e d5 df 78 f5 f2 74 fd ee 67 1a b4 4b a0 af 06
f8 5d 17 2a 4f 87 e6 e3 bd 98 91 65 b5 da 12 09
6f 96 c7 6b ed 6e eb 2a 9a d2 63 59 03 4b 65 eb
d3 be 80 a2 42 45 57 6c bb ec d8 ab cc 5c cc 77
b3 1b ec 8b 0e d2 a9 e9 58 1f a5 9d 17 d6 41 45
11 ad b8 24 d1 4e 90 b2 60 7e 99 38 a0 a4 61 ba
53 79 e7 22 31 74 a7 c1 0a 40 06 09 30 f6 8c a7
78 9c 4b 3a 86 28 e9 71 fd 45 d7 b7 ee 83 55 42
1e 56 48 5f 74 cb be fc 23 a6 7c 1e 65 ab ac 8d
b4 e2 6a fd 6a 69 d0 9b 6f cc 63 73 e6 96 fd 8f
da 5a 39 ee b0 27 24 b8 5b 4c 18 5d 88 94 2f f8
48 5e 5c a2 e8 1a 3f bd 3c 71 2e 7d 7d 93 04 cc
a1 f8 2b b2 ab 80 44 2a 18 cc df d2 dc e4 23 2b
1d 4d df 2b 0f 46 25 36 75 bd 9b 07
```

2.3 Assignment 3: Key Hierarchy

With the base setup completed we started building our specified key hierarchy. Our first task was to answer some questions.

1. *The identity key is one type of signature key, describe some differences between an identity and a signature key.*
   - An identity key must meet minimum security requirements and needs the SRK as its parent while the signature key does not need any specific security and can be created further down the key hierarchy.[2]
2. *Which keys can be used for file encryption?*
   - All keys can theoretically be used for file encryption but, since we do not want to compromise our key hierarchy and provide adequate security, the best key to use is a storage key whose main purpose is to encrypt the file contents.[2]
3. *There is one type of key that exists, but its use is not recommended. Which key is that, and why does it exist?*
   - Legacy key, which can be used both for signing and encryption although not recommended. They have lower security in order to provide backwards compatibility with older standards. [2]

In order to create keys, the steps from assignment 1 and 2 needs to be completed before proceeding. Our task was then to create keys according to the hierarchy

name	*parent*	*type*
A	`SRK`	non migratable storage key
B	`A`	migratable storage key
C	`B`	a non migratable sign key
D	`B`	a migratable sign key
E	`B`	a migratable bind key
F	`A`	a non migratable sign key
G	`A`	a migratable sign key
H	`SRK`	an identity key

Table 1: Key hierarchy given by specification

presented in table 1. In order to fully understand the key structure we also needed to provide a drawing of the key hierarchy which can be seen in figure 1.

We started with key A and utilized the `createkey` command with the appropriate settings. For A we passed `-kt e` to specify encryption (storage) key, `-pwdk aaa` to set the key password, `-pwdp sss` to unlock the parent key, `-ok A` to give the key a name and `-hp 40000000` to provide the key handle of the parent. With the key created we loaded it into the TPM. We then wanted to list the keys loaded into the TPM but unfortunatly found that the `listkeys` command was not reliable and did not list all the keys currently loaded.

```
# Create A from SRK
tss@TSS ~/keys createkey -kt e -pwdk aaa -pwdp sss -ok A -hp 40000000
# load into TPM
tss@TSS ~/keys loadkey -hp 40000000 -ik A.key -pwdp sss
New Key Handle = EDA20B34
```

Similarily, we then continued by creating the rest of the keys. Note that the creation of key C failed since all children of a migratable key also needs to be migratable [3].

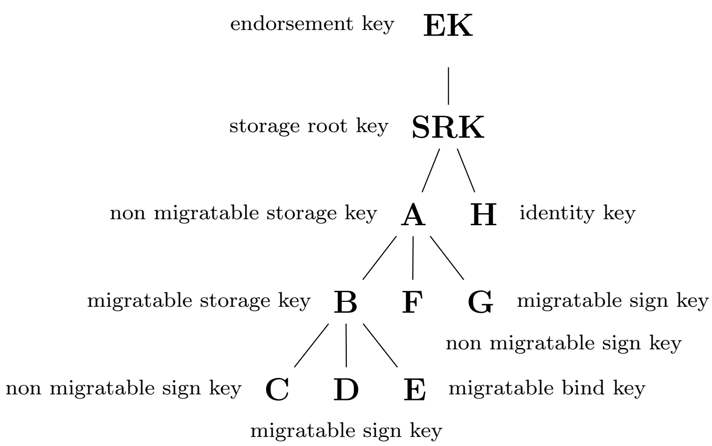

Figure 1: Drawing of the key hierarchy.

```
tss@TSS ~ createkey -kt em -pwdk bbb -pwdp aaa -pwdm mmm -ok B -hp EDA20B34
tss@TSS ~ loadkey -hp EDA20B34 -ik B.key -pwdp aaa
New Key Handle = 7E96F005

tss@TSS ~ createkey -kt s -pwdk ccc -pwdp bbb -ok C -hp 7E96F005
Error Invalid key usage from TPM_CreateWrapKey

tss@TSS ~ createkey -kt sm -pwdk ddd -pwdp bbb -pwdm mmm -ok D -hp 7E96F005
tss@TSS ~ loadkey -hp 7E96F005 -ik D.key -pwdp bbb
New Key Handle = 9B2AAD5C

tss@TSS ~ createkey -kt bm -pwdk eee -pwdp bbb -pwdm mmm -ok E -hp 7E96F005
tss@TSS ~ loadkey -hp 7E96F005 -ik E.key -pwdp bbb
New Key Handle = 4DA3B20E

tss@TSS ~ createkey -kt s -pwdk fff -pwdp aaa -ok F -hp EDA20B34
tss@TSS ~ loadkey -hp EDA20B34 -ik F.key -pwdp aaa
New Key Handle = FBDCE93E

tss@TSS ~ createkey -kt sm -pwdk ggg -pwdp aaa -pwdm mmm -ok G -hp EDA20B34
tss@TSS ~ loadkey -hp EDA20B34 -ik G.key -pwdp aaa
New Key Handle = 4DD31188

tss@TSS ~ identity -pwdk hhh -pwds sss -pwdo ooo -ok H -la idH
tss@TSS ~ loadkey -hp 40000000 -ik H.key -pwdp sss
New Key Handle = 61A7D9DA
```

The key handles of the loaded keys are presented in table 2.

Key	*Handle*
A	`EDA20B34`
B	`7E96F005`
C	`N/A`
D	`9B2AAD5C`
E	`4DA3B20E`
F	`FBDCE93E`
G	`4DD31188`
H	`61A7D9DA`

Table 2: Key handles given by loading into TPM1

2.4 Assignment 4: Key Migration

For our next assignment we also started by answering the given questions.

1. *Is it possible for a migratable key to be the parent of a non-migratable key?*
   - No, since when a migratable parent key is exported, all its children needs to be able to tag along. Having the child non-migratable would then cause a contradiction.[3]
2. *Which command is the first to be executed when performing a key migration?*
   - On the TPM you are about to import into, TPM2 in our case, you should create a new migrateable storage key that can then be used with TPM1 to encrypt keys you want to export with the command `TPM_CreateMigrationBlob`. This way the keys will be protected during transport. [3]
   - On the TPM which you are about to export keys from (TPM1), the first command should be `TPM_CreateWrapKey` to prepare the private part of the key. [4]
3. *Give a short description of the command* `TPM_ConvertMigrationBlob`.
   - This command takes a migration blob and decrypts it to a normal wrapped blob which is then possible load into the TPM using the `TPM_LoadKey` function. Note that the command migrates private keys only. The migration of the associated public keys is not specified by TPM because they are not security sensitive. [5]
4. *Which TPM command loads the migrated keys into the TPM?*
   - After the migration blob has been converted, the command to load the wrapped key into the TPM is `TPM_LoadKey`. [4]
5. *Is it the TPM or the TSS that handles the transfer of the migration blob?*
   - It is the TSS, since TPM:s have no ability to communicate directly with another TPM. [3]

We should now migrate keys from TPM1 into TPM2. Since it is necessary to execute commands on both machines we would like to advice the reader to pay extra attention to on which machine the command is executed.

Begin with creating a migratable storage key on TPM2. This will be used to encrypt the keys from TPM1 during transit.

```
... Connected to TPM2 ...
tss@TSS ~ createkey -kt em -pwdk mig -pwdp sss -pwdm mmm -ok mig -hp 40000000
tss@TSS ~ loadkey -hp 40000000 -ik mig.key -pwdp sss
New Key Handle = D1C50374
```

Reconnect to TPM1 again. Since we are on the same TSS machine we have easy access to the migration key we just created from TPM2. Use this one to encrypt the keys to be exported. This first code block is a sample of a complete migration procedure for one key. Next code block will move multiple keys in a more compact fashion.

```
... Connected to TPM1 ...
tss@TSS ~ migrate -hp EDA20B34 -pwdp aaa -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlob -ik B.key
... Reconnect to TPM2 ...
tss@TSS ~ loadmigrationblob -hp D1C50374 -if migBlob -pwdp mig
Successfully loaded key into TPM.
New Key Handle = 2BD1CF9B
```

The parent key handle in the example above is for the migration key used to encrypt during transport. Imported keys will then branch from that one. The following commands were issued on TPM1.

```
... Connect to TPM1 ...
... Key C not available ..
... Key D ...
tss@TSS ~/tpm2/keys migrate -hp 7E96F005 -pwdp bbb -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlobD -ik ~/tpm1/keys/D.key
Wrote migration blob and associated data to file.
... Success ...
... Key E ...
tss@TSS ~/tpm2/keys migrate -hp 7E96F005 -pwdp bbb -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlobE -ik ~/tpm1/keys/E.key
Wrote migration blob and associated data to file.
... Success ...
... Key F ...
tss@TSS ~/tpm2/keys migrate -hp EDA20B34 -pwdp aaa -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlobF -ik ~/tpm1/keys/F.key
CreateMigrationBlob returned 'Authorization failure for 2nd key' (29).
... Failure (non-migratable) ...
... Key G ...
tss@TSS ~/tpm2/keys migrate -hp EDA20B34 -pwdp aaa -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlobG -ik ~/tpm1/keys/G.key
Wrote migration blob and associated data to file.
... Success ...
```

We then loaded the successfully created migration blobs while reconnected to TPM2.

```
... Reconnect to TPM2 ...
... Key D ...
tss@TSS ~/tpm2/keys loadmigrationblob -hp D1C50374 -if migBlobD -pwdp mig
Successfully loaded key into TPM.
New Key Handle = 8E75BF6E
... Key E ...
tss@TSS ~/tpm2/keys loadmigrationblob -hp D1C50374 -if migBlobE -pwdp mig
Successfully loaded key into TPM.
New Key Handle = 2FF1AE1D
... Key G ...
tss@TSS ~/tpm2/keys loadmigrationblob -hp D1C50374 -if migBlobG -pwdp mig
Successfully loaded key into TPM.
New Key Handle = 34E45AD6
```

From this we get new key handles for the exported keys on TPM2 summerized in table 3.

Key	*Handle*
mig	`D1C50374`
B	`2BD1CF9B`
C	`N/A`
D	`8E75BF6E`
E	`2FF1AE1D`
F	`N/A`
G	`4DD31188`
H	`34E45AD6`

Table 3: Key handles given by loading into TPM2

Lastly we answer the questions related to key migration.

1. *Do the migration and document in your report.*
   - See code blocks above.
2. *There are other ways to migrate keys.*

   When do you use a key of type `TPM_KEY_USAGE = TPM_Migrate`? (Hint: look in [6])
   - `TPM_KEY_USAGE` can have the value `TPM_KEY_MIGRATE`, which is used when there is a need for a migration authority (we did not find any `TPM_migrate` command).[6][7]
3. *What is the rewrap option of the migrate command used for?*
   - This is used when migration authority is needed. [8] (page 4)

2.5 Assignment 5: Extending Values to PCRs

1. *Describe one TPM command that can be used to extend a SHA-1 digest to a PCR.*
   - `TPM_SHA1CompleteExtend`
     "This capability terminates a pending SHA-1 calculation and EXTENDS the result into a Platform Configuration Register using a SHA-1 hash process." [5]
2. *Describe which TPM command that can be used to read a PCR value.*
   - `TPM_PCRRead`
     "The `TPM_PCRRead` operation provides non-cryptographic reporting of the contents of a named PCR." [5]

We shall now calculate the hash of the `tpmbios` binary file, and place it inside one of the registers on the TPM. In the code below, the command `` ` which tpmbios` `` will expand to the full path of the `tpmbios` binary, a SHA-1 computation will be done of it and the result placed in registry 11. The last command will then read back the value of register 11.

```
tss@TSS ~/tpm/tpm4720/tpm sha -if `which tpmbios` -ix 11
SHA1 hash for file '/home/tss/tpm/tpm4720/libtpm/utils/tpmbios':
Hash: 55ac0462404445623f38fdae9adf87d487125874
New value of PCR: dba8c73876627a1e4439627b64c96c8f9c8d404a

tss@TSS ~/tpm/tpm4720/tpm pcrread -ix 11
Current value of PCR 11: dba8c73876627a1e4439627b64c96c8f9c8d404a
```

2.6 Assignment 6: File Encryption

1. *Why is `TSS_Bind` a TSS command, and not a TPM command?*
   - The public part of the binding key pair is used for encrypting data. This should be possible to do from anywhere, so the need to load the private key is unnecessary. Once the data is bound, only the one with control over the private part should be able to decrypt it, which is why the private key needs to be loaded for decryption to be possible. This only requires the unbind operation to be a TPM command. [9]
2. *Give some differences between Data binding and Data sealing.*
   - "Binding": we can encrypt data on another computer, and decryption can only be done on the computer which has the TPM with the private key. [10][9].
   - "Sealing": binds data to a certain value of the PCR and a key that is not migratable. Then only this specific TPM can decrypt (unseal) the data, and even then only if the PCR value is the same as when encryption happened (sealing). [10][9]
3. *Can a key used for data sealing be migrated to another TPM?*
   - No since, the key used for the sealing is required to be non-migratable. [10]

Now we are going to try to bind data using the TPM emulator. We start by creating a migratable binding key with TPM1.

```
tss@TSS ~ createkey -kt bm -pwdk bnd -pwdm mmm -pwdp sss -ok bindMig -hp 40000000
tss@TSS ~ loadkey -hp 40000000 -ik bindMig.key -pwdp sss
New Key Handle = 44632C94
```

To test the binding functionality, we create a simple text file that contains the line "Hello World".

```
tss@TSS ~ echo "Hello World" > test.txt
```

To bind the file, using the newly created key, the following command is used. Take note that it is the public part of the key pair (.pem) that is used for encryption.

```
tss@TSS ~ bindfile -ik bindMig.pem -if test.txt -of test.bound
```

To unbind the file you need the private part of the key, which was loaded into the TPM earlier. Decryption is done on the file, and a `diff` command confirms that they are identical in content.

```
tss@TSS ~ unbindfile -hk 44632C94 -if test.bound -of test.unbound -pwdk bnd
tss@TSS ~ diff test.txt test.unbound
tss@TSS ~ ... same content in both files ....
```

1. *Why does the "private key" have to be loaded before decryption?*

    - Since we not only need secure keys but also a secure platform in order to correctly decrypt the file, i.e. not possible without a authorized TPM. This is implemented in parts by requiring the private key to be descended from SRK which in turn is related to the unique secret of the TPM chip, the decryption needs parts of the secret only known to the chip manufacturer. [9] The key is also encrypted via the need for password previously set by the TPM.

To test if we can unseal once the key is migrated to TPM2, we do the migration procedure and load it into the other TPM.

```
tss@TSS ~ migrate -hp 40000000 -pwdp sss -pwdo ooo -im mig.key \
-pwdk mig -pwdm mmm -ok migBlobBindMig -ik bindMig.key
Wrote migration blob and associated data to file.
tss@TSS ~ loadmigrationblob -hp D1C50374 -if migBlobBindMig -pwdp mig
Successfully loaded key into TPM.
New Key Handle = 79961905
```

Unsealing the file should then not be any problem.

```
tss@TSS ~ unbindfile -hk 79961905 -if ../../tpm1/keys/test.bound \
-of test.unbound -pwdk bnd
tss@TSS ~ cat test.unbound
Hello World
```

As shown above, the correct message can be read form the unsealed file. As stated, the binding key is migratable, however, it requires that the private part of the key pair is loaded into a trusted platform before decryption can take place. This requires authorization to use the key.

Now we return to TPM1. We shall try sealing with a couple of different key types, and we begin by creating them.

```
... Storage Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk seal -ok storSealnonMig -kt e
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik storSealnonMig.key
New Key Handle = C285648D
... Legacy Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk seal -ok legacySeal -kt l
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik legacySeal.key
New Key Handle = F892B803
... Signing Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk seal -ok signSeal -kt s
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik signSeal.key
New Key Handle = 27EB4F3D
... Binding Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk seal -ok bindSeal -kt b
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik bindSeal.key
New Key Handle = 919E7DC3
```

Now we test them by sealing the text file used earlier, and then unsealing it to see if it worked.

```
... Storage Key ...
tss@TSS ~ sealfile -hk C285648D -if test.txt -of test.seal -pwdk seal
tss@TSS ~ unsealfile -hk C285648D -if test.seal -of test.unsealed -pwdk seal
tss@TSS ~ cat test.unsealed
Hello World
... Storage key can seal file ...
... Legacy Key ...
tss@TSS ~ sealfile -hk F892B803 -if test.txt -of test.seal -pwdk seal
Error Invalid key usage from TPM_Seal
... Legacy key can not seal file ...
... Signing Key ...
tss@TSS ~ sealfile -hk 27EB4F3D -if test.txt -of test.seal -pwdk seal
Error Invalid key usage from TPM_Seal
... Signing Key cannot seal file ...
... Binding Key ...
tss@TSS ~ sealfile -hk 919E7DC3 -if test.txt -of test.seal -pwdk seal
Error Invalid key usage from TPM_Seal
... Binding key cannot seal file ...
```

As can be seen, it is only the non-migratable storage key that can be used for sealing. When we to try to make it migratable, the following occurred.

```
... Migratable Storage Key ...
createkey -hp 40000000 -pwdp sss -pwdk seal -pwdm mmm -ok storSeal -kt em
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik storSeal.key
New Key Handle = BF52F964
tss@TSS ~ sealfile -hk BF52F964 -if test.txt -of test.seal -pwdk seal
Error Invalid key usage from TPM_Seal
... Migratable key cannot seal file ...
```

The key needs to be non-migratable in order to be used by the `sealfile` command. It was however possible to use migratable keys when using `bindfile` command, since with bind we only encrypt with the key itself. Contrast this with sealing were we also bind the encryption with the PCR values of the TPM. Therefore it is impossible to seal a file with migratable keys as they would be useless for decryption since some PCR values can't be extracted out of the TPM [9].

A legacy key does not work since it does not meet the required security level. A binding key cannot be used either since we need ways of encrypting the PCR value which the storage key provides. In the same way a signing key can not be used either since it is only secure when used with signing and we need secure encryption. In conclusion, we have no way of migrating keys and decrypting the file with the other tpm once they are sealed.[2] [9] [6]

2.7 Assignment 7: TPM Authentication

Here we test two different ways a TPM can authenticate itself.

1. *In the above, could the `verifyfile` command have been done by another TPM?*
   - Yes, since the signing is done by the private key, the public part can be used for verifying the private signing. And since that part is public, any TPM can use it.[5]
2. *Which TPM command is used to decrypt the file?*
   - "`TPM_UnBind` takes the data blob that is the result of a `Tspi_Data_Bind` command and decrypts it for export to the User." [5]

     The binding operation is undone by unbind. This requires the private part of the keypair and is done inside the TPM.
3. *Can the decryption based authentication be done by using data sealing instead of binding?*
   - Not really. Sealing limits the encryption and decryption to only one specific TPM. No other TPM should be able to encrypt or decrypt it which means the only external party that the TPM can authenticate towards is itself, which is kind of moot.

Now we shall sign a file using a signature key, created by TPM1, and let TPM2 verify it. Then we shall bind a file with a bind key, and later unbind it. Create both keys and load them into TPM1.

```
... Signing Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk sign -ok signer -kt s
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik signer.key
New Key Handle = 0061D9D5
... Binding Key ...
tss@TSS ~ createkey -hp 40000000 -pwdp sss -pwdk bind -ok binder -kt b
tss@TSS ~ loadkey -hp 40000000 -pwdp sss -ik binder.key
New Key Handle = 353456BF
```

Use the private part of the signing key to sign the text file that we have been using previously.

```
tss@TSS ~ signfile -hk 0061D9D5 -if test.txt -os test.signed -pwdk sign
```

Switching over to TPM2, we use the public part of the key pair to verify that it is a valid signature.

```
tss@TSS ~ verifyfile -is tpm1/keys/test.signed -if tpm1/keys/test.txt \
-ik tpm1/keys/signer.pem
... The signature was verified...
```

Binding a file, using the public part of the binding key pair is done like this.

```
... Using TPM2 ...
tss@TSS ~/tpm1/keys bindfile -ik binder.pem -if test.txt -of test.bound
... Using TPM1 ...
tss@TSS ~/tpm1/keys unbindfile -hk 353456BF -if test.bound -of test.unbound -pwdk bind
... Success ...
```

Returning to TPM1 to unbind results in a file we can read.

2.8 Assignment 8: Attestation

Attestation is a way to obtain proof that the correct, untampered, software is the one which is loaded. An example would be a banking app that will not authenticate anything unless it knows the software that is running is a legitimate version. Just like with authentication, attestation can either be signature based or decryption based. We begin by studying the signature aspect. We will create an AIK (Attestation Identity Key) using the command `identity`.

```
tss@TSS ~/tpm1/keys identity -la assign8 -pwdo ooo -pwdk idty -pwds sss -ok identity
tss@TSS ~/tpm1/keys loadkey -hp 40000000 -ik identity.key -pwdp sss
New Key Handle = 7FB4546E
```

To assign the key to a PCR value, the hash from `tmpbios` is calculated again, and assigned to registry 11. This value is then quoted by the AIK.

Note: "`quote`" is a command in bash as well, so write the full path so no there is no ambiguity to which one you use.

```
tss@TSS ~/tpm1/keys sha -if `which tpmbios` -ix 11
SHA1 hash for file '/home/tss/tpm/tpm4720/libtpm/utils/tpmbios':
Hash: 55ac0462404445623f38fdae9adf87d487125874
New value of PCR: dba8c73876627a1e4439627b64c96c8f9c8d404a
tss@TSS ~/tpm1/keys /home/tss/tpm/tpm4720/libtpm/utils/quote -v -hk 7FB4546E \
-bm dba8c73876627a1e4439627b64c96c8f9c8d404a -pwdk idty
```

The `-v` flag is necessary, since otherwise the signature isn't printed. The printouts on TPM1 are shown on the next page.

```
TPM_ReceiveSocket: From TPM length=335
00 C5 00 00 01 4F 00 00 00 00 00 00 01 00 01
00 02 00 00 00 0C 00 00 08 00 00 00 00 02 00 00
00 00 00 00 01 00 AA 12 07 57 F3 8D B4 8C CE 99
B1 C5 09 84 E8 3C 48 DE A6 C1 7D 96 A2 22 7C F4
AF 52 FB F9 A9 C0 CF 43 A3 4C F5 B4 B4 97 DF E2
39 5A A2 B3 63 4D C6 AB 3E D4 53 82 40 87 29 D9
E6 50 1A 15 BC 9F 99 15 36 80 EA BA 52 CE AF E4
EC 9F 0C 54 75 83 4B 08 09 5D D1 D2 7A B8 F4 74
17 D9 6C 3A 60 A2 DE 8D 8C 34 D1 03 B0 62 6E AF
89 8A E6 75 15 E6 4B EB AC 37 C7 FD 1C A9 BF 85
16 15 AB 96 AB DD 14 B3 C4 49 90 6C 2B 80 B3 DB
2E EF 3F F0 FA BD C1 46 F8 43 08 A8 05 BD 5B DF
9D 12 D6 2F 35 4C 5F CC F3 59 2E D9 E3 C3 13 7E
D8 5B 66 6D 4B 33 CA 68 69 86 AD C1 B4 10 5A FB
F4 5E C1 91 90 8E FE 6C A1 DC C7 FC 8E 12 AB 50
06 AA 15 51 40 E7 F3 15 D2 92 B7 E4 4D 42 F2 5A
72 3D 5D 82 3F B9 B8 70 9A 75 0A F8 C7 B8 AB 0F
0F B8 EB F5 77 76 98 73 44 90 89 F8 7A C3 8B DE
AD 19 0C F1 D2 23 CA B4 28 51 74 4E 1A 07 00 04
67 E9 4B D9 C0 A0 16 89 30 DB 00 6B 58 9D F2 26
D7 F8 A9 33 62 89 15 85 A0 B4 E8 E1 65 AC B6
```

Now to test decryption-based attestation. To do this we will extend the hash of out trusted text file into PCR registry 12.

```
tss@TSS ~/tpm1/keys sha -if test.txt -ix 12
SHA1 hash for file 'test.txt':
Hash: 648a6a6ffffdaa0badb23b8baf90b6168dd16b3a
New value of PCR: 4e2a96d44e4bd5f04e54066371a84ec963677755
```

This registry value is then included in the creation of a storage key, which will bind the key to the text file hash.

```
tss@TSS ~/tpm1/keys createkey -hp 40000000 -kt e -pwdp sss -pwdk ix -ok ixStorage \
-ix 12 4e2a96d44e4bd5f04e54066371a84ec963677755
tss@TSS ~/tpm1/keys loadkey -hp 40000000 -ik ixStorage.key -pwdp sss
New Key Handle = 2DBF1EEB
```

Sealing and unsealing of files is done in the same way as before, with the additional PCR registry parameter.

```
tss@TSS ~/tpm1/keys sealfile -hk 2DBF1EEB -if test.txt -of test.seal -pwdk ix
tss@TSS ~/tpm1/keys unsealfile -hk 2DBF1EEB -if test.seal -of test.unsealed -pwdk ix
tss@TSS ~/tpm1/keys cat test.txt
Hello World
tss@TSS ~/tpm1/keys cat test.unsealed
Hello World
```

However, were we to change the original text file, and load its new hash into the PCR registry, unsealing will no longer be possible.

```
tss@TSS ~/tpm1/keys echo "Hello World!" > test.txt
tss@TSS ~/tpm1/keys cat test.txt
Hello World!
tss@TSS ~/tpm1/keys sha -if test.txt -ix 12
SHA1 hash for file 'test.txt':
Hash: a0b65939670bc2c010f4d5d6a0b3e4e4590fb92b
New value of PCR: c2df11d0c106d988764e95f6d098b1b402d1d7ca

tss@TSS ~/tpm1/keys unsealfile -hk 2DBF1EEB -if test.seal -of test.unsealed -pwdk ix
Error PCR mismatch from TPM_Unseal
```

This way you can detect unwanted tampering with files, and hinder decryption of sensitive files on potentially compromised systems.

This brings us to the end of these assignments. To restore the state of the TPM back to the beginning, the following command should be run.

```
tss@TSS ~/tpm1/keys forceclear
tss@TSS ~/tpm1/keys forceclear
TPM_ForceClear returned error 'TPM disabled'.
```

2.9 Assignment 9: Your First TPM Application

We made sure that we had run the `forceclear` command from the last part in the assignment 8. On TPM1; we stopped the `tpm_server` daemon and removed `~/tpm/tpm4720/tpmstate/00.permall` to start afresh. We relaunched the TPM by issuing the `tpm_server` command again.

On the TSS machine we once more put the TPM in a well defined state by running

```
tss@TSS ~ tpmbios
tss@TSS ~ sudo -E /usr/local/sbin/tcsd -e -f
```

We opened a new terminal and created a new EK key pair for our "new" TPM.

```
tss@TSS ~ createek
```

This concluded the brief setup in preparation for developing our own tpm application.

2.9.1 Developing TPM Applications

Our task was to develop a TPM application that communicates with the TPM. We decided to implement a simple random number generator that outputs 8 bytes in base 10 format. This is the result when running our application.

```
tss@TSS ~ ./trousersApp
... Debug information ...
#########################################
###### adsec14 TPM Random Generator #####
#########################################
(Line70, main) Tspi Random Byte
 returned 0x00000000. Success.
Result: |238|119|216|122|40|219|170|241|
#########################################
... Debug information ...
```

During the development we had a lot of code to go on and only needed 4 lines of code in addition to the printouts in order to have a working application. We utilized the `Tspi_TPM_GetRandom` command to retrieve random bytes from the TPM. The full working code is visible below. The application is written in C and can be compiled on the provided TSS virtual machine via the following command where trousersApp.c is the file containing our code.

```
gcc -o trousersApp trousersApp.c -ltspi -Wall
```

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<tss/platform.h>
#include<tss/tss_defines.h>
#include<tss/tss_typedef.h>
#include<tss/tss_structs.h>
#include<tss/tspi.h>
#include<trousers/trousers.h>
#include<tss/tss_error.h>

#define DEBUG 1
// Macro for debug messages
#define DBG(message, tResult) { if(DEBUG)  printf("(Line%d, %s) \
%s returned 0x%08x. %s.\n",__LINE__ ,__func__ , message, \
tResult,(char *)Trspi_Error_String(tResult));}

int main( int argc, char **argv ){

    BYTE *rgbPcrValue, *rgbNumPcrs;
    UINT32 ulPcrValueLength;
    UINT32 exitCode, subCapSize, numPcrs, subCap, i, j;

    TSS_HCONTEXT hContext=0;
    TSS_HTPM hTPM = 0;
    TSS_RESULT result;
    TSS_HKEY hSRK = 0;
    TSS_HPOLICY hSRKPolicy=0;
    TSS_UUID SRK_UUID = TSS_UUID_SRK;
    //By default SRK is 20bytes 0
    //takeownership -z
    BYTE wks[20];
    memset(wks,0,20);

    //At the beginning
    //Create context and get tpm handle
    result =Tspi_Context_Create(&hContext);
    DBG("Create a context\n", result);
    result=Tspi_Context_Connect(hContext, NULL);
    DBG("Connect to TPM\n", result);
    result=Tspi_Context_GetTpmObject(hContext, &hTPM);
    DBG("Get TPM handle\n", result);
    //Get SRK handle
    //This operation need SRK secret when you takeownership
    //if takeownership -z the SRK is wks by default
    result=Tspi_Context_LoadKeyByUUID(
            hContext,
            TSS_PS_TYPE_SYSTEM,
            SRK_UUID,
            &hSRK
            );
    DBG("Get SRK handle\n", result);
```

```
    result=Tspi_GetPolicyObject(hSRK,
            TSS_POLICY_USAGE, &hSRKPolicy);
    DBG("Get SRK Policy\n", result);
    result=Tspi_Policy_SetSecret(hSRKPolicy,
            TSS_SECRET_MODE_SHA1,20, wks);
    DBG("Tspi_Policy_SetSecret\n", result);

    // put your TPM code here
    printf("\n\n##################################\n");
    printf("## adsec14 TPM Random Generator ##\n");
    printf("##################################\n");
    BYTE *randomBytes;
    UINT32 randomSize=8;
    randomBytes = (BYTE *)malloc(randomSize);
    result=Tspi_TPM_GetRandom(hTPM, randomSize, &randomBytes);
    DBG("Tspi Random Byte\n", result);
    printf("Result: |");
    for (i=0;i<randomSize;++i)
    {
      printf("%d|", randomBytes[i]);
    }
    printf("\n");
    printf("##################################\n\n");
    //At the end of program
    //Cleanup some object
    result = Tspi_Context_FreeMemory(hContext, NULL);
    DBG("Tspi Context Free Memory\n", result);
    result = Tspi_Context_Close(hContext);
    DBG("Tspi Context Close\n", result);
    return 0;
}
```

3 Conclusion

This concludes our report of our solutions to the assignments on the Trusted Platform Module (TPM) [1]. This project was not particularly difficult, but demanded a lot of patience since the documentation of the commands used by the TSS were very non-descriptive. A quick glance on a guide to Unix best practice regarding help text would have helped immensely for the usability of the programs. We recommend taking a look at `http://docopt.org/` or similar initiatives.

However, with a lot of consultation with the TPM manual [5], lecture slides and external literature the entire process could be recreated without problems. We believe this exercise further helped grow our understanding of how secure platforms are constructed and helped us correct misunderstandings we had gathered from simply reading the literature.

References

[1] D. of Electrical and L. U. Information Technology, "Project: Tpm," http://www.eit.lth.se/fileadmin/eit/courses/eitn50/Project_TPM/Project_TPM.pdf, 2017, [Online; accessed 25-September-2017].

[2] A. Segall, "TPM Keys — Creating, Certifying, and Using Them," http://opensecuritytraining.info/IntroToTrustedComputing_files/Day1-7-tpm-keys.pdf, 2017, [Online; accessed 26-September-2017].

[3] T. Hardjono, "Overview of the TPM Key Management Standard," https://trustedcomputinggroup.org/wp-content/uploads/Kazmierczak20Greg20-20TPM_Key_Management_KMS2008_v003.pdf, 2008, [Online; accessed 2-October-2017].

[4] J. Mirosavljevic, "Securing telecommunication network node data using TPMs," http://www.eit.lth.se/sprapport.php?uid=792, 2014, [Online; accessed 2-October-2017].

[5] I. Trusted Computing Group, "TPM Main - Part 3 Commands," https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf, 2011, [Online; accessed 2-October-2017].

[6] ——, "TPM Main - Part 2 Structures," https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf, 2011, [Online; accessed 2-October-2017].

[7] M. Ryan, "Introduction to the tpm 1.2," http://courses.cs.vt.edu/cs5204/fall10-kafura-BB/Papers/TPM/Intro-TPM.pdf, 2009, [Online; accessed 1-October-2017].

[8] R.-U. BOCHUM, "TPM Key Management and Key Replication Mechanisms," https://shazkhan.files.wordpress.com/2010/10/http__www-trust-rub-de_media_ei_lehrmaterialien_trusted-computing_keyreplication_.pdf, 2009, [Online; accessed 27-September-2017].

[9] Wikipedia, "Trusted Platform Module — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Trusted%20Platform%20Module&oldid=801030586, 2017, [Online; accessed 25-September-2017].

[10] S. Ben, "Trusted platforms - lecture 6," http://www.eit.lth.se/fileadmin/eit/courses/eitn50/Lectures/Lect5_slides.pdf, 2017, [Online; accessed 1-October-2017].