JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

11100 Johns Hopkins Road • Laurel, Maryland 20723-6099

Using the IBM TSS Utilities as a Scripting Language

Release 1.1

David Challener, Johns Hopkins University Applied Physics Laboratory

1. INTRODUCTION

The command line utilities that come with the IBM TPM Software Stack (TSS) 2.0 package provide a way of exercising most of the utility of the Trusted Platform Module (TPM) 2.0. As a result, they compose a nearly complete scripting language (when combined with cmd.com in windows or bash in Linux) for the TPM 2.0, making use of the TPM very easy. They also provide (in their source code) a good source of examples of how one can use the IBM TSS libraries to talk directly with the TPM 2.0. In order for these to be used securely as a scripting language, there must be a way for a user to securely provide passwords to the TPM. That capability was added as of version 974 of the code. This document is based on version 1470 of the code. Future versions are not guaranteed to be compatible with older versions, as new capabilities will be incorporated, which may change default values.

This paper reviews various things one might want to do with a TPM and how to accomplish them using the utility functions to interface with the TPM. It assumes familiarity with the TPM 2.0 specification [TPM 2.0]. Subsequent sections cover the following information:

- Section 2: Notation
- Section 3: An overview of using the basic TPM subsystems
- Section 4: Examples of usage of all the utilities
- Section 5: Interacting with OpenSSL
- Section 6: Running the regression tests
- Section 7: Using scripts securely

The appendices contain examples of using the scripting language to do practical things:

- Appendix A: How to setup the software and hook it up to physical or emulated TPM 2.0s
- Appendix B: How to initialize a TPM using the utilities
- Appendix C: Examples of using the scripting language

2. NOTATION

In this document, files with an extension of .bin are binary files. Files with an extension of .hex are stored in hex-ascii. The convention used by IBM's TSS files is followed, where h8xxxxxxx.bin will refer to the public name of keys or non-volatile memory (NV) indices that have been created by the TPM. All lowercase is used for filenames and executables. Examples are given for Linux, but if you download the Windows 32-bit equivalent of the UNIX *cp* and *cat* command (available at http://gnuwin32.sourceforge.net/) those scripts need not change when run in Windows.

3. SOME SHORTCUTS IN USING THE UTILITIES

Sometimes, after creating a key, it is necessary to know either its public portion or name. It is possible to compute the name from the public data, or load the key and ask the TPM for its public portion. However, as a user creates keys, the underlying IBM TSS library transparently calculates and stores this information in files (and deletes them when they are no longer relevant).
For a key:

- **hpHANDLE.bin is the public portion of the key. (e.g., hp80000001.bin)**
- **hHANDLE.bin is the name (e.g., h80000001.bin)** For an NV index:

- **hINDEX.bin is the name (e.g., h01987654.bin)**
- **nvpINDEX.bin is the marshaled TPMS_NV_Public of the index (e.g., nvp01987654.bin)**

To calculate the Name of an entity, see the **publicname** utility.

4. BASE FUNCTIONS

For a scripting language to be effective for using the TPM, it should cover the major uses of the TPM. Examples in this section use utilities to exercise TPM base functions. Note that these utilities provide coverage of the NIST-compliant functions, and thus (together with Bash or cmd.com) form a complete scripting language. Examples in this section provide a command line perspective of the usage of the base functions, including handling of non-volatile memory (NV), keys, Platform Configuration Registers (PCRs), usage of policies, and TPM management.

4.1. NV Utilities

Create an NV index

Command: nvdefinespace

There are many types of NV indices. The most common ones follow.

User-Created Ordinary NV Index Controlled by a Password

- Index is 01xxxxxx (e.g., 01987654)
- Password is: NV_password
- Size is Bytes
- Owner hierarchy password is OwnerPassword

User-Created Ordinary NV Index, only Writeable by Owner, Readable by Anyone

Note: This example assumes you have changed the owner authorization to OwnerPassword.

```
$ ./nvdefinespace -ha 01987654 -hi o -pwdp OwnerPassword -sz 32 +at ow +at ar
```

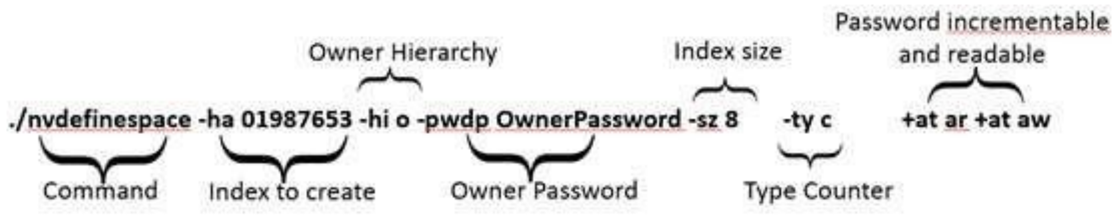
- Index is 01xxxxxx (-ha 01987654)
- Size is NumberOfBytes (-sz 32)
- Owner hierarchy password is OwnerPassword (-pwdp OwnerPassword) used for authorizing creation of the NV index
- Hierarchy chosen is the owner hierarchy (-hi o)
- Authorization to write is set to Owner writeable (+at ow)
- Authorization to read is set to Password readable (+at ar)
  - Since no password is set, the index has no restrictions on readability.

Create an NV counter

Owner-created NV counter at index 01987653, incrementable and readable with a null password.

Note: This example assumes you have changed the owner authorization to OwnerPassword.

```
./ nvdefinespace -ha 01987653 -hi o pwdp OwnerPassword -sz 8 -ty c +at ar +at aw
```


Owner Hierarchy
Index size
Password incrementable and readable
./nvdefinespace -ha 01987653 -hi o -pwdp OwnerPassword -sz 8 -ty c +at ar +at aw
Command
Index to create
Owner Password
Type Counter

- Index is 01xxxxxx (-ha 01987653)
- Size is NumberOfBytes (-sz 8)
- Owner hierarchy password is OwnerPassword (-pwdp OwnerPassword) used for authorizing creation of the NV index
- Hierarchy chosen is the owner hierarchy (-hi o)
- Authorization to increment is set to password writeable (+at ow)
  - Since no password is set, this is generally readable ● Authorization to

read is set to Password readable (+at ar)

○ Since no password is set, the index is generally readable.

Create an NVPIN PASS

When creating a PASS type NVPIN, it is necessary to change the type to 'p' for PASS. It is also necessary to set a password for the index. NVPIN PASS and FAIL are always 8 bytes, so there is no point setting the size. When there's no requirement for an immutable password, it's a good idea to create a policy that provides a means of changing that password, in case the password is compromised. This is demonstrated under ***Creating Policies***.

- **$ ./nvdefinespace -ha 01987654 -hi o -pwdp OWNERAUTH -nalg sha256 -ty p -pwdn NVPASSWORD**
- Note- see above for details, except *nalg* is the encryption algorithm, *pwdn* is the NV Index authorization password, and *–ty p* is for type PIN Pass

Create a NVPIN FAIL

Here, change the type to 'f' for FAIL and set a password for the index. NVPIN PASS and FAIL are always 8 bytes, so there is no point setting the size. Usually it is also a good idea to create a policy that provides a means of changing that password. Otherwise, if a password is compromised, it cannot be changed. This is demonstrated under ***Creating Policies***.

- **$ ./nvdefinespace -ha 01987654 -hi o -pwdp OWNERAUTH -nalg sha256 -ty f -pwdn NVPASSWORD**
- Note- see above for details, except *–ty f* is for type PIN Fail

Write an NV Index

Typically, an NV index is written by the owner authorization. That is done as follows.

Normal Data NV Index, Written by Owner

- **$ ./nvwrite -ha INDEX -hia o –pwdn OWNERAUTH   -ic DATA –off offset**

Changing Value in an NVPIN

- **$ ./nvwrite -ha INDEX -hia o –pwdn OWNERAUTH   -id pinLimit -if NEWLIMIT**
- **$ ./nvwrite -ha INDEX -hia o –pwdn OWNERAUTH   -id pinPass -if 0**

Read an NV Index

Normal data NV index, authorization not needed for the read operation.

Example:

- **$ ./nvread -ha INDEX -sz 32 -off 0**

Delete an NV Index

- **$ ./nvundefinespace –hi o –ha INDEX –pwdp OWNERAUTH**

Increment an NV Counter

Increments an NV counter.

Example:

- **$ ./nvincrement -ha 01987654**

Utilities

In ekutils.c, there is a utility called **getIndexSize**, which takes an index value and returns the size of the index. It is used by the utility getIndexContents that takes an index value and returns the contents (of course, authorization needs to be provided). Returning the size of an index hasn't (yet) been put into a separate utility but this capability could be added to the TSS package in an upcoming release, or the user can link this function into their own command-line utility.

4.2. Creating and Managing Keys

Creating Primary Keys

- **$ ./createprimary**

By itself, this creates a primary RSA storage key in the null hierarchy with a null password, with an empty unique field file and doesn't save the public key file name.

Create an ECC NISTP256 SRK Key

- **$ ./createprimary -hi o -pwdp OwnerAuth -opu SRKpub.bin -ecc nistp256 -st -halg sha256**

Create an HMAC key, SHA256 with a NULL Password, with no Password, with an empty Unique Field

- **$ ./createprimary -hi o -pwdp OwnerAuth -kh -nalg sha256 -halg sha256**

Create a Primary Restricted Signing Key (AIK like) Using ECC P256 and SHA-256

- **$ ./createprimary -hi o -pwdp OwnerAuth -opu SRKpub.bin -ecc nistp256 -sir -halg sha256**

Create a Primary Unrestricted Signing Key (for PolicySigned), RSA 2048, SHA-1

- **$ ./createprimary -hi o -pwdp OwnerAuth -si -halg sha256**

HMAC with a Key

This takes the HMAC key stored at the permanent handle 81000001 and uses it to HMAC the file text.txt and outputs the result into the file output.hmac

- **$ ./hmac –hk 81000001 -if test.txt -of output.hmac**

This new version of the **hmac** command also includes an option to feed in a text file and have it return the HMAC of that file.

Create a Non-Primary Key

The command **create** is used to create keys that are underneath storage keys. Not only can it be used to create keys with any supported algorithms (defaults are RSA 2048, with SHA256 hash algorithm) which are duplicable without DA protection, without a policy, and a NULL password, it can also create such a key, and not save either the public key or output private key anywhere. (One has to be careful not to do this, as the key cannot actually be used if it is created and then discarded.)

Create a ECC P384 SHA384 Restricted Signing Key under the SRK

Note: the SRK must be in the TPM at handle at 81000001 for this to work.

- **$ ./create -hp 81000001 -sir -ecc nistp384 -halg sha384 -opu RSK.pub -opr RSK.blob**

Create a Non-Restricted NIST P384 ECC Signing Key for Signing with Password PW

- **$ ./create -hp 81000001 -si -ecc nistp384 -halg sha384 -pwdk PW –opu SK.pub -opr SK.blob**

Create a SHA-384, HMAC Key Under the SRK that is Duplicable

- **$ ./create -hp 80000000 -sir -ecc nistp384 -kh -nalg sha384 -halg sha384 -opu HMAC.pub -opr HMAC.blob -pol PolicyAuthorize.policy**

Create a Duplicable AES 256-bit Symmetric Key Under the SRK that is Duplicable

- **$ ./create -hp 80000000 -sir -ecc nistp384 -kh -opu AES.pub -opr AES.blob -pol PolicyAuthorize.policy**

Making a Key Persistent

Here, a newly-generated key is made persistent.

- **$ ./evictcontrol -hi o -pwdp OwnerAuth -ho 80000001 -hp 81000001**

1. If auth is TPM_RH_OWNER, then persistentHandle shall be in the inclusive range of 81 00 00 0016 to 81 7F FF FF16.
2. If auth is TPM_RH_PLATFORM, then persistentHandle shall be in the inclusive range of 81 80 00 0016 to 81 FF FF FF16.

Clearing Keys

Clearing a Persistent Key

Clearing a primary key is done using the **evictcontrol** command.

- **$ ./evictcontrol -hi o –pwdp OwnerAuth -hp 81000002 -ho 81000002**

Clearing a Non-Primary Key from Dynamic Memory

Evicting a non-primary key is done with the **flushcontext** command.

- **$ ./flushcontext -ha 80000002**

4.3. PCRs

Read a PCR

- **$ ./pcrread -ha 16**

Extend a PCR

Note that this only takes a file that is smaller than or equal to the size of the HASH.

- **$ ./pcrextend -ha 16 -if data.file**

Reset a PCR

- **$ ./pcrreset -ha 16**

4.4. Using a Policy to Authorize a Command

Changing the NV index password using the NV index password

The NV index passwords can only be changed by using a policy. Earlier an NV index was created that contained **PolicyAuthValue.policy** that allows changing the password for that NV index if the user knows the current password. Given that index, the password is changed from “User1Password” to “User2Password” as follows.

Start a Policy Session

- **$ ./startauthsession -se p**

This returns a session number; call it SN (e.g., 03000000.)

Execute policycommandcode Command, using policyauthorize as the command being authorized

- **policycommandcode -ha SN -cc 0x0000016B**

Use that Session and Execute the policyauthvalue Command

- **$ ./policyauthvalue -ha SN**

Read out the Contents of the Digest Associated with that Session

- **$ ./policygetdigest –ha SN –of digest.bin**

End the Session

- **$ ./flushcontext -ha SN**

Using a Key That Can Be Used by Two Different Users to Authorize Use

First create a policy, called PolicyOrSK1SK2, which is a policy that can be satisfied by User1, by providing proof of control of Signing Key1, or by User2, by providing proof of control of Signing Key2.

Create a Key with the PolicyOrSK1SK2.policy Policy

- **$ ./createprimary -hi o –pwdp OwnerAuth -kh -nalg sha256 -halg sha256 -pol PolicyOrSK1SK2.policy**

Note: This will return a handle; call it 80000002.

Create a Session

Start a Policy Session

- **$ ./startauthsession -se p -on TPM.nonce**

Note: This returns a session number; call it SN (e.g., 03000000.).

Create a signature of the aHash using SK1

- **$ ./load -hp 81000001 -ipu SK1.pub -ipr SK1.priv -se SN** This returns the handle of SK1; call it hSK1.

Now sign the concatenation of nonceTPM, expiration, cpHashA, and policyRef buffers; the expiration is always 32 bits (4 bytes). In this case, it is 8 bytes of zeros. cpHashA and policyRef in this example are both NULL. This can be done on any TPM which has the private SK1 loaded. Call the handle of the private signing key hSK1.

- **cat nonceTPM zeroExpire > aHashInternal**
- **$ ./sign -hk hSK1 -pwdk User1Password -if aHashInternal -os Signature.bin**

Satisfy the Policy

Satisfying the policy can be done using the **policysigned** command as follows:

- **policysigned -hk 80000002 -ha SN -halg sha256 -sig Signature.bin**

Here the SigFile is a standard 256 byte RSA signature file (ECC signatures are not yet supported by the utilities).

Run PolicyOr

- **$ ./policyor -ha SN -if PolicySignedSK1.policy -if PolicySignedSK2.policy**

At this point, the session digest has the correct value in it to allow the key hKey to be used to sign something. This is done as follows:

First, create a file with text to sign:

- **$ ./echo "text" >text.tst** Now sign it.

- **$ ./sign -se SN -hk 80000002 -if text.tst -ipu h80000002.pub -os SignatureOut.bin**

Here, input the public portion of the key so that it will also verify the signature.

4.5. Finding Out Capabilities of the TPM

You can use the command line utilities to find out information about your TPM using the **getcapability** command.

List the transient keys

- **$ ./getcapability -cap 1 -pr 80000000**

List the permanent keys

- **$ ./getcapability -cap 1 -pr 81000000**

List the currently used indexes

- **$ ./getcapability -cap 1 -pr 01000000**

5. USING THE TPM WITH OPENSSL

The TPM format for keys is different from that of OpenSSL, but it is likely that it will be necessary to use OpenSSL keys for policysigned or use the public key that corresponds to a TPM key in OpenSSL. For this reason, Ken Goldman created a file called **tpm2pem**.

The command line **create family** and **readpublic** utilities also have a -opem option to save PEM format output at time of creation

5.1. To Create an OpenSSL PEM File

- **$ ./tpm2pem -ipu hp80000002.bin -opem hp80000002.pem**

This takes a TPM public data, automatically stored by TSS in hp80000002, and converts it into OpenSSL ***.pem*** format. In the prior command, the auto-created public data that corresponds to the key at handle 80000002 is used.

It should be noted that utilities that need public keys will accept a ***.pem*** formatted file directly so the reverse conversion is not necessary.

6. RUNNING THE REGRESSION TEST

There are two regression tests included in the utils subdirectory; one is for Linux and one is for Windows. In Linux, it is as easy as running reg.sh -a.

In Windows, however, first copy the *diff* and *touch* files from https://sourceforge.net/projects/unxutils/?source=typ_redirect into the subdirectory before running reg.bat, as Windows doesn't have those two utilities, and the regression test requires them. Once that is done, the reg.bat file should run to conclusion without an error.

It should be noted that the regression tests give a very thorough set of examples of using the utilities, and there are many examples available beyond those contained in this document.

7. USING SESSIONS CONFIDENTIALLY

The TSS documentation section on **Command Line Utilities** explains how to handle session state when using the utilities.

8. REFERENCES

[TPM 2.0] Trusted Platform Module 2.0 Specification, Parts 1-3. Online: https://trustedcomputinggroup.org/tpm-library-specification

[Cha17] David Challener, "Memo on Creating Policies," the Johns Hopkins University Applied Physics Laboratory, AOS-L-17-2414, 2017

9. APPENDIX A: SETTING UP A SYSTEM TO USE THE UTILITIES

In order to make creation of policies easy, it is necessary to have a program that does the extend function. Ken Goldman of IBM research has written one called ***policymaker***. See *Memo on Creating Policies* [Cha17] for more details.

Run ***SetUpCommandCodes.scr*** script (in Linux) or the **SetUpCommandCodes.bat** (in Windows). The script generates helper files that will be useful for creating policies using policymaker.

Download and Compile

The TSS, command line utilities, sample C code, sample scripts, sample policies, and EK root certificates are in the package at: **https://sourceforge.net/projects/ibmtpm20tss/.** The documentation includes build instructions.

The companion SW TPM is at **https://sourceforge.net/projects/ibmswtpm2**/ and includes build instructions.

Utilities and Library

You can now download precompiled Windows versions of the utilities from SourceForge. If you wish, you can compile them on your own. It is suggested that you read the directions that come with the downloaded source code before building. The following directions are from the documentation of an older version.

Using a Physical TPM

The TSS documentation section on **Optional Customization** explains the three methods of switching between a physical TPM and a software TPM.

Startup

For either Linux or Windows, starting up the emulator and using it is done with three commands:

- **tpm_server**
- **powerup**
- **startup**

10. APPENDIX B INITIALIZATION OF A TPM USING THE UTILITIES

When setting up a TPM 2.0 for the first time, there are a number of things that one might want to do including:

- Clearing the TPM
- Determining the EK certificate
- Regenerating the EK on the TPM

- Generating an SRK for the TPM and making it permanently resident
- Obtaining the public portion of the SRK key
- Setting the Owner password
- Setting the Privacy password (aka the EK hierarchy password)
- Setting the Dictionary attack password
- Creating a restricted signing key for the TPM and making it permanently resident
- Setting up an NV index that is owner writeable, but generally readable These

may all be accomplished using the utilities.

10.1. Clearing the TPM

For this, either use the BIOS or the command **clear**.

```
$ ./clear –hi l
```

Note that after the dictionary attack authorization is changed to **NewLockoutAuth,** it is necessary to execute:

- **$ ./clear -hi l NewLockoutAuth**

10.2. EK Certificate

The **createekcert** utility provisions EK certificates. Use -h for usage help.

The certificate can be read in text format using either **createk** or **nvread**. Use -h for usage help.

10.3. Regenerating the EK on the TPM

Here the **createek** command is used to do various things.

Re-Creating the EK

- **$ ./createek -cp -alg rsa -noflush**

Note that this does not require inputting the Endorsement Hierarchy password. It is assumed to be NULL.

Printing the EK Certificate

- **$ ./createek –ce**
  Note: this won't work if you are talking to the emulator, because it won't have a certificate

10.4. Change the DA Parameters

Change the Password

- **$ ./hierachychangeauth –hi l –pwdn NewLockoutAuth**

Change the New Max Tries to 50, Recovery Time to an Hour (3600 seconds)

- **$ ./dictionaryattackparameters -nmt 50 -nrt 3600**

10.5. Reset the DA Counter

- **$ ./dictionaryattacklockreset -pwd NewLockoutAuth**

10.6. Changing a Hierarchy's Password

The TPM starts with a NULL password. This example changes the owner password to "OwnerAuth."

Note, the possible hierarchies are:

l lockout
e endorsement
o owner
p platform

- **$ ./hierarchychangeauth -hi o -pwdn OwnerAuth**

11. APPENDIX C SOME EXAMPLES

11.1. Example 1: Handling the DA counter reset

It is possible to reset the DA counter, which is triggered if too many attempts are made to (incorrectly) guess a password, in two ways. The first is with a password, and the second is with a policy. The password itself will only allow a single attempt for reset. As a result, an attacker attempting to cause a denial of service may deliberately trigger the DA counter and also make a wrong attempt on the DA reset password to put the machine in lockdown until the DA reset counter has been cleared.

Such an attack does not work if a DA policy is used to reset the DA counter. However, it may not be advisable to allow just anyone to have the full power of the DA policy. Consequently, change the policy for the DA counter so that, with a policyRef of *Reset*, it resets the counter, with a policyRef of *Clear*, it clears the owner hierarchy and, with a policy of *param*, allows changing the parameters. The following provides an example where:

1. A signing key is created that uses the owner password for signing. (The owner password is not DA protected.)
2. That signing key is placed at a particular handle value.
3. A policy is created that allows use of that Signing key to reset the DA password.
4. The DA policy is set to use the newly created policy.
5. The DA password is set to a random value (to prevent it being guessed).
6. An example is shown of resetting the DA counter using the signing key.

Creating a Signing Key and Placing It at a Particular Handle, Using Owner Authorization for Authorization (This requires giving it a policy, which is PolicySigned pointing to OwnerAuth)

- The policy pointing to OwnerAuth comes from the table in the Memo on Creating Policies [Cha17]: The policy number is as follows:

0d84f55daf6e43ac97966e62c9bb989d3397777d25c5f749868055d65394f952

If this is done before the OwnerAuth is set, the " -pwdp OwnerAuth" can be omitted.

- **$createprimary -hi o -pwdp OwnerAuth -opu SRKpub.bin -ecc nistp256 -sir -halg sha256 -pol 0d84f55daf6e43ac97966e62c9bb989d3397777d25c5f749868055d65394f952** This returns a key handle. (I assume here it is 80000001.)

- **$evictcontrol -hi o -pwdp OwnerAuth -ho 80000001 -hp 81000001**

Creating a Policy Using PolicySigned (with a policyref) and PolicyCommandCode

Create a Policy that Requires Use of the Private Key and the Command Code Reset to Reset the Lockout Counter

1. First, write out the public key in hex, to be used in the policy.
   - **$ ./bin2hex h81000001**
2. **N**ext, write out the policyRef in hex. The MkCmd utility can do this.
   - **$ ./MkCmd Reset > Reset.bin**
3. Then, make a policy that requires policysigned using that public key with that policyref.
   - **$ ./cat policysigned.cmd h81000001.hex lf.bin Reset.bin > policysigned.Reset**
4. Add a requirement that it use the reset command.
   - **$ ./cat policysigned.Reset lf.bin policycommandcode.cmd dareset.cmd > policysignedReset.policy**
5. Finally, make it into a policy.
   - **$ ./policymaker -if policysignedReset.policy -of policysignedReset.bin**

Create a Policy that Requires Use of the Private Key and the Command Code Clear to Clear the Owner Hierarchy

1. First, write out the public key in hex, to be used in the policy.
   - **$ ./bin2hex h81000001**
2. Next, write out the policyRef in hex. The MkCmd utility can do this.
   - **$ ./MkCmd Clear > Clear.bin**
3. Then, make a policy that requires policysigned using that public key with that policyref.
   - **$ ./cat policysigned.cmd h81000001.hex lf.bin Clear.bin > policysigned.Clear**
4. Add in a requirement that it use the reset command.

- **$ ./cat policysigned.Clear lf.bin policycommandcode.cmd clear.cmd > policysignedClear.policy**

5. Finally, make it into a policy.
   - **$ ./policymaker -if policysignedClear.policy -of policysignedClear.bin**

Create a Policy that Requires Use of the Private Key and the Command Code that Allows Changing the Dictionary Attack Parameters

1. First, write out the public key in hex, to be used in the policy.
   - **$ ./bin2hex h81000001**
2. Next, write out the policyRef in hex. The MkCmd utility can do this.
   - **$ ./MkCmd Param > Param.bin**
3. Then, make a policy that requires policysigned using that public key with that policyref.
   - **$ ./cat policysigned.cmd h81000001.hex lf.bin Param.bin > policysigned.Param**
4. Add in a requirement that it use the reset command.
   - **$ ./cat policysigned.Param lf.bin policycommandcode.cmd daparameters.cmd > policysignedParam.policy**
5. Finally, make it into a policy.
   - **$ ./policymaker -if policysignedParam.policy -of policysignedParam.bin**

ORing the Policies Together

- **$ ./bin2hex policysignedClear.bin**
- **$ ./bin2hex policysignedReset.bin**
- **$ ./bin2hex policysignedParam.bin**
- **$ ./cat policysignedReset.hex policysignedClear.hex policysignParam.hex > Lockout.policy**
- **$ ./policymaker -if Lockout.policy -of Lockout.bin**

Setting the DA Policy to be this Newly Created Policy

- **$ ./setprimarypolicy -hi l -pol Lockout.bin -halg sha256**

Creating a Random Value and Setting the DA Password to this Random Value

- **$ ./DAPass=”$(getrandom -by 16)”**
- **$ ./setprimarypolicy -hi l -pwda $DAPass**

Showing How to Reset the DA Counter Using this Signing Key

Start a Policy Session

- **$ ./TPM_SESSION_ENCKEY=”$(getrandom -by 16 -ns)”**
- **$ ./startauthsession -se p -on -on nonceTPM**

Note: This returns a session number; call it SN (e.g., 03000000).

Execute policysigned Using 81000001

- **policysigned -ha SN -hk h81000001 -pwdk OwnerPassword -in nonceTPM -pref Reset.bin**
- **policycommandcode -ha SN -cc reset.cmd**

Execute the Command

- **clear -hi l -se1 SN**

11.2. Example 2: Protecting large password databases

Large databases of passwords are periodically exposed on the web. They are usually salted and hashed, but the hash tries and salt value are typically also published with the database. At this point, renting a large number of virtual machines makes it rather easy to crack many of the passwords in the table using a simple dictionary attack.

This could all be avoided if, instead of using a salted hash to obscure the passwords, an HMAC of the userid and password were stored instead, with the HMAC key being one generated on the TPM and stored in it persistently. (If the key were duplicable, it could also be sorted on another system for a backup.)

This example shows how one could create such a key, make it persistent in a known area, and allow it to be duplicated using a policy. Such a key could either be generally usable (TPMs aren't fast enough to be used in brute-force hammering of passwords) or could use a system known key, released during the boot cycle into the kernel's key store. For this example, a password is stored locked to a set of PCRs, which are assumed to only be available to the kernel early in the boot cycle. (This is done for the HMAC key used in EVM, for example, so this is well known.)

Generate a random password

$ ./HMACPass="$(getrandom -by 16)"

Create a policy locked to PCRs which represent the current state of the system (early in boot)

- **$ ./createMask policyPCR1234567891011.policy 1 2 3 4 5 6 7 8 9 10 11**
- **$ ./policymaker if= policyPCR1234567891011.policy of=policyNV.bin**

Note: This needs to be run early in the boot cycle, when the kernel will later retrieve the password.

Create an NV index locked to that policy, but writeable with OwnerAuth.

- **$ ./nvdefinespace -ha 01987654 -hi o -pwdp OwnerPassword -sz 16 +at ow -pol policyNV.bin**

Note that when the PCRs are in the correct state, the index can be either read or written, but it can be written anytime by the owner.

Write the password in the NV index

- **$ ./nvwrite -hia o -ha 01987654 -pwdn OwnerPassword -ic $HMACpass**

Create a policy that allows for duplication using a specified key (81000001)

1. First, write out the public key in hex, to be used in the policy.
   - **$ ./bin2hex h81000001**
2. Next, write out the policyRef in hex. The MkCmd utility can do this.
   - **$ ./MkCmd Dup > Dup.bin**
3. Then, make a policy that requires policysigned using that public key with that policyref.
   - **$ ./cat policysigned.cmd h81000001.hex lf.bin Dup.bin > policysigned.Dup**
4. Add a requirement that it use the reset command.
   - **$ ./cat policysigned.Dup lf.bin policycommandcode.cmd duplicate.cmd > policysignedDup.policy**
   - **$ ./policymaker -if policysignedDup.policy -of HMACpolicy.bin**

Create a persistent HMAC key with that policy

- **$ ./createprimary -hi o –pwdp OwnerAuth -pol HMACpolicy.bin -kh -nalg sha256 -halg sha256**

This returns a handle, say 80000002.

It is then made persistent:

- **$ ./evictcontrol -hi o –pwdp OwnerAuth -ho 80000002 -hp 81000002**

A utility for a PAM file to use when a password comes in

HmacUserAndPassword.scr

This utility will take a userid and a password and then HMAC them with the HMAC key stored by the TPM in persistent storage.

- **$ ./read USERID PASSWORD**
- **$ ./hmac -hk81000002 -ic $USERID$PASSWORD –os output.hmac**

Output.hmac contains the HMACed concatenated userid and password, which can be either stored as a new password or compared to a stored HMAC in the PAM table. If this table of HMACed passwords is stolen, it does not provide any way for the thief to determine the passwords used to create that table.

Example: A Test Script

This Windows batch file executes a lot of commands, loading keys, flushing them, creating indexes, writing and reading them, etc.

```
REM  in a separate command prompt>>

powerup startup
```

REM clear the tpm clear

-hi l

REM change the owner authorization from NULL to OwnerAuth hierarchychangeauth

-hi o -pwdn OwnerAuth

REM checking key generation REM

create an srk (primary ECC key)

createprimary -hi o -pwdp OwnerAuth -opu SRKpub.bin -ecc nistp256 -st -halg sha256 REM

make the srk persistent

evictcontrol -hi o -pwda OwnerAuth -ho 80000000 -hp 81000001

REM flush the non-persistent copy flushcontext -ha 80000000

REM create a restricted signing key under the SRK

create -hp 81000001 -sir -ecc nistp384 -halg sha384 -opu RSK.pub -opr RSK.blob

REM load key just created under SRK load -hp 81000001 -ipu RSK.pub -ipr

RSK.blob

REM (returns handle 80000002) flush the key flushcontext

-ha 80000001

REM create a regular signing key under the SRK

create -hp 81000001 -si -ecc nistp384 -halg sha384 -pwdk PW -opu SK.pub -opr SK.blob

REM get its name hash -if

SK.blob -oh SK.name

REM load the signing key (it will have a handle of 80000002) load

-hp 81000001 -ipu SK.pub -ipr SK.blob

REM (returns handle 80000001) flush the key flushcontext

-ha 80000001

REM create an RSA 2048 signing key

create -hp 81000001 -si -rsa -halg sha384 -pwdk PW -opu SK.pub -opr SK.blob

REM convert the public part of the RSA key to pem format tpm2pem

-ipu SK.pub -opem SK.pem

REM load the RSA signing key load -hp

81000001 -ipu SK.pub -ipr SK.blob

REM REM sign with the just created key

REM create a file to sign echo "To

Sign"> toSign.txt

sign -hk 80000001 -pwdk PW -if toSign.txt -os Signature.bin

REM verify the signature

verifysignature -if toSign.txt -is Signature.bin -hk 80000001

REM evict the key flushcontext -ha 80000001

REM create an hmac key under the SRK

create -hp 81000001 -kh -halg sha256 -pwdk PW -opu HK.pub -opr HK.blob

REM load the hmac key under the SRK load -hp 81000001 -ipu HK.pub -

ipr HK.blob REM hmac the toSign.txt file

hmac -hk 80000001 -pwdk PW -if toSign.txt -os HMACed.bin

REM evict the key flushcontext -ha 80000001

REM Seal data under the SRK

```
REM Make a policy for sealing to pcrs 0-13

REM Read the PCRs 0-13 pcrread -ha 0 -ns

> 0.pcr pcrread -ha 1 -ns > 1.pcr

pcrread -ha 2 -ns > 2.pcr pcrread

-ha 3 -ns > 3.pcr pcrread -ha 4 -

ns > 4.pcr pcrread -ha 5 -ns >

5.pcr pcrread -ha 6 -ns > 6.pcr

pcrread -ha 7 -ns > 7.pcr

pcrread -ha 8 -ns > 8.pcr

pcrread -ha 9 -ns > 9.pcr

pcrread -ha 10 -ns > 10.pcr

pcrread -ha 11 -ns > 11.pcr

pcrread -ha 12 -ns > 12.pcr

pcrread -ha 13 -ns > 13.pcr


REM put them all in one file erase

combine.txt

copy 0.pcr + 1.pcr + 2.pcr + 3.pcr + 4.pcr + 5.pcr + 6.pcr + 7.pcr + 8.pcr + 9.pcr + 10.pcr + 11.pcr +
12.pcr + 13.pcr combine.txt

REM pause 5

REM create a policy with them (assumes sha256) policymakerpcr

-bm 3fff -if combine.txt -of policyPCRs.txt policymaker -if

policyPCRs.txt -of policyPCRs.bin


echo sealed >toSeal.txt

create -hp 81000001 -bl -if toSeal.txt -opu pubSealed.dat -opr privSealed.dat -pol policyPCRs.bin


REM load sealed data

load -hp 81000001 -ipu pubSealed.dat -ipr privSealed.dat
```

REM Start a policysession

startauthsession -se p policypcr -

ha 03000000 -bm 3fff REM

unseal

unseal -ha 80000001 -of unsealed.txt flushcontext

-ha 80000001

REM create a primary restricted signing key

createprimary -hi o -pwdp OwnerAuth -ecc nistp256 -sir -halg sha256

REM evict the key flushcontext -ha 80000000

REM create a primary regular RSA signing key

createprimary -hi o -rsa -pwdp OwnerAuth -si -halg

sha256 REM evict the key flushcontext -ha 80000000

REM create an hmac primary key

createprimary -hi o -pwdp OwnerAuth -kh -nalg sha256 -halg sha256

REM flush the primary key flushcontext -ha 80000000

REM REM Checking the NV commands REM

create a regular NV index

nvdefinespace -ha 01987654 -hi o -pwdp OwnerAuth -sz 32 +at ow +at ar

REM write the index

nvwrite -ha 01987654 -hia o -pwdn OwnerAuth -ic DATA

REM read the index (should result in 44 41 54 41 which is ascii for DATA) nvread

-ha 01987654 -off 0 -sz 4

REM delete the index

nvundefinespace -hi o -ha 01987654 -pwdp OwnerAuth

REM create a counter index (writeable by Owner, writeable, readable by user) nvdefinespace

-ha 01987654 -hi o -ty c -pwdp OwnerAuth -sz 32 +at ow +at ar +at aw

REM increment the counter (this can only be done by the NV index password) nvincrement

-ha 01987654

REM read the index (should result in an incremented value) nvread

-ha 01987654

REM delete the index

nvundefinespace -hi o -ha 01987654 -pwdp OwnerAuth

Intentionally Blank Page