Programming With TrouSerS

David Challener
Johns Hopkins University
Applied Physics Laboratory

June 13, 2011

Contents

Chapter 1

Introduction

The Trusted Computing Group (TCG) Software Stack (TSS) provides a common applications programming interface (API) for software that interfaces with the Trusted Platform Module (TPM). TrouSerS is an open source implementation of the API. For those who want to use the TPM and TrouSerS, reading the TSS specification is a daunting task. In actuality, C programming with TrouSerS is easy, and this paper provides a means to get started.

1.1 First Steps

The first step to programming the TPM is to make sure that your system can talk to the TPM. This involves turning the TPM on. The TPM ships in the off state. You must go into the BIOS of your system and turn the TPM on. Turning the TPM on usually requires activating the TPM in a section of the BIOS devoted to security. In some systems this also involves rebooting the system several times. Activating the TPM really isn't that hard, but it can be onerous. After the TPM is turned on, you need to make sure you have the correct driver installed, and that a daemon called tcsd is running. In Fedora core 13, the correct driver is the default driver. For Fedora core 12 or Ubuntu, follow the steps below.

- Main install (Fedora 12 Linux w/ gcc)
  - yum install trousers
  - yum install tpm-tools
  - yum install trousers-devel
  - yum install gcc

Ubuntu Linux w/ gcc
sudo apt-get install trousers
sudo apt-get install tpm-tools
sudo apt-get install libtspi-dev
sudo apt-get install gcc

- Turn on the TPM
  - Go to BIOS and make sure the TPM is on
    - (if it is and you don't know owner auth, you may want to clear it and start over).
    - The procedures differ from PC to PC unfortunately

- Start up tcsd (sudo tcsd start)
  - Make sure you can run the TPM tools (use tpm_getpubek)

- Take ownership using tpm_takeownership –z
  - (The –z sets the SRK password to all zeros, the default "well known secret")
  - Use 123 for the owner_auth for this class

Note: If your machine doesn't have the TPM listed in its ACPI table, you can still get the device driver to use it
- In that case you must use:
  - sudo modprobe tpm_tis force=1 interrupts=0
  - sudo tcsd start

TrouSerS has been ported to other operating systems:

- Windows,see ***security.polito.it/tc/trouserswin/***
- MAC, see ***http://www.osxbook.com/book/bonus/chapter10/tpm/***
- BSD, see ***http://www.listware.net/201008/openbsd-tech/2241-driver-tpm4-and-third-party-packages-for-trusted-platform-modules.html***
- Solaris, see ***http://blogs.sun.com/wyllys/entry/tpm_support_in_solaris***

Chapter 2

Sample Source code

For those who can't wait, working code that exercises all the APIs can be found at: ***http://sourceforge.net/projects/trousers/files/***

Chapter 3

Includes

The first thing in a c file is the set of header files that are included. This list should be sufficient for anything you are doing. It may include things you don't need, but that should be OK.

Basic includes look like this:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

#include <tss/tss_error.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>
```

Chapter 4

Error messages

TrouSerS includes some basic interpretation for error messages. This is wonderful when something doesn't quite work in your program. It is a good idea to start out with a macro that describes the result of using one of the TrouSerS' APIs in English. This is why the

 # include <trousers/trousers.h>

statement is in our includes file.

Our macro is

#define DBG(message, tResult) printf("Line%d, %s)
%s returned 0x%08x. %s.\n", __LINE__,__func__,
message, tResult,(char *)Trspi_Error_String(tResult))

with an example usage of:

DBG("I just created a signing key", result);

Chapter 5

Preamble

Now that you have the basic includes, you must start with a preamble. The preamble is code that typically is inserted at the beginning of every program that you write. The preamble establishes a context for your program and connects the program with the TPM. It also gets a handle for the TPM, so APIs that need that handle can talk to it. Lastly, the preamble gets a handle for the Storage Root Key (SRK) inside the TPM, and tells the context the authorization of that key, so that whenever the SRK is used, the context can provide authorization without further effort on the part of the programmer.

5.1 First Steps

```
int main(int argc,char **argv)
  {
    TSS_HCONTEXT    hContext;
    TSS_HTPM        hTPM;
    TSS_RESULT      result;
    TSS_HKEY        hSRK=0;
    TSS_HPOLICY     hSRKPolicy=0;
    TSS_UUID        SRK_UUID=TSS_UUID_SRK;
    BYTE            wks[2]; //For the well known secret

  // Set wks to the well known secret: 20 bytes of all 0's

memset(wks,0,20);
```

```
//Pick the TPM you are talking to.

// In this case, it is the system TPM (indicated with NULL).

  result = Tspi_Context_Create(&hContext);
  DBG("Create Context",result);

   result = Tspi_Context_Connect(hContext, NULL);
  DBG("Context Connect",result);

// Get the TPM handle
  result=Tspi_Context_GetTpmObject(hContext,
                                   &hTPM);
        DBG("Get TPM Handle",result);


// Get the SRK handle
  result=Tspi_Context_LoadKeyByUUID(hContext,
                                    TSS_PS_TYPE_SYSTEM,
                                    SRK_UUID,
                                   &hSRK);
        DBG("Got the SRK handle", result);

//Get the SRK policy

result = Tspi_GetPolicyObject(hSRK,
                                     TSS_POLICY_USAGE,
                                    &hSRKPolicy);

        DBG("Got the SRK policy",result);

//Then set the SRK policy to be the well known secret

   result=Tspi_Policy_SetSecret(hSRKPolicy,
                        TSS_SECRET_MODE_SHA1,
                        20,
                        wks);
```

//Note: TSS_SECRET_MODE_SHA1 says "Don't hash this.

// Use the 20 bytes as they are.

DBG("Set the SRK secret in its policy",result);

The first command, memset(wks, 0, 20), sets the variable wks to be the well known secret of 20 bytes of zeros.

The next two commands create a context and connect the context to the local TPM (NULL represents the local TPM). If you want to use a TPM other than the local one, check out the sample code at
http://sourceforge.net/projects/trousers/files/ .

Chapter 6

Postlude

At the end of every program you need to clean up. Some APIs automatically create memory for the returned variables. These are owned by the context and you have to tell the context to clean them up. Any objects you create also need to be similarly released.

6.1 Postlude Code

```
// Clean up
    Tspi_Context_Close(hobjects you have created);

    Tspi_Context_FreeMemory(hContext,NULL);
//  This frees up memory automatically allocated for you.

    Tspi_Context_Close(hContext);
    return 0;
}
```

If you put these things together, you have a compilable program. The program doesn't do much, but it does set up a context, connect it to a TPM, get a handle for both the TPM and the SRK, set the SRK authorization to be the well known secret (all zeros), then clean up and close.

You can compile it with gcc as follows:

6.2 Compile Command

gcc file -o file.exe -ltspi -Wall

-ltspi tells it where to find the tspi library of apis, and -Wall tells it to report all warnings (there should not be any).

Chapter 7

Comments on memory handling

As mentioned in the last section, some APIs automatically assign memory for you. You can find out which APIs do so by reading the APIs, but there is a simpler way. If the API has a variable listed as a BYTE **, then you may assume that it is assigning the memory for you. Since it is doing the memory assignment, you should not!

Instead you should create a variable

BYTE *variableName;

and pass into the API as &variableName. This way the context can create the memory and assign it to your pointer.

WARNING
If you do something different, such as creating the variable as
BYTE variableName[256];
then you have assigned the memory and the API does it as well, and weird things can happen!

Prototype:
```
TSS_Result      Tspi_Hash_Sign
{
    TSS_HHASH       hHash,                  // in
    TSS_HKEY        hKey,                   // in
    UINT32*         pulSignatureLength,     // out
```

```
    BYTE**          prgbSignature          // out
};
```
Code:
```
    UINT32          SignatureLength;
    BYTE           *rgbSignature;
    TSS_RESULT      result;
    TSS_HHASH       hHash;
    TSS_HKEY        hKey;

    result=Tspi_Hash_Sign(hHash,hKey,
                          &SignatureLength,
                          &rgbSignature);
```

Chapter 8

Comments on authorization

Many objects that a programmer will use, such as keys, the TPM, and NVRAM, have an associated authorization. Fortunately TrouSerS has been set up so that the context will keep track of authorizations associated with particular objects. This means that (one time only) the context has to be told the authorization used by objects.

Each object will have an associate Policy object, which will hold the authorization. Some objects (like the TPM and the SRK) will already have an associated policy object. For these objects, one only needs to ask the object for its policy using the Tspi_GetPolicyObject command. When creating an object, however, the new object will not have a policy object associated with it, so it needs to have a policy object created, using Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_POLICY, TSS_POLICY_USAGE, &hNewPolicyName). This policy object then needs to be associated with the newly created object via

Tspi_Policy_AssignToObject(nNewPolicyName,hObject).

To let the TSS know the authorization for a particular object, such as a key or the TPM itself, you must:

- Define a policy object handle

  TSS_HPOLICY myPolicyHandle;

- Associate the Policy handle with a Policy Object
  - Get an existing Policy (e.g. for the TPM)

result=Tspi_GetPolicyObject(hTPM,
TSS_POLICY_USAGE,
&hTPMPOlicy)

– Create a policy(and later associate it with the object)
result=Tspi_Context_CreateObject(hContext,
TSS_OBJECT_TYPE_POLICY,
TSS_POLICY_USAGE
&hNewPolicy);

- Fill in the authorization value into the Policy object
Tspi_Policy_SetSecret(hNewPolicy,
TSS_SECRET_MODE_SHA1,
lengthOfPassword,*newPassword);

- Associate the Policy object with the appropriate object (if you didn't get an existing policy from the object to begin with)
Tspi_Policy_AssignToObject(hNewPolicy,hObject);

Once you have given the context the policy once, it remembers it. Consider the following code:

```
// Getting the TPM's policy object
TSS_HPOLICY hTPMPolicy;
result=Tspi_GetPolicyObject(hTPM,
                            TSS_POLICY_USAGE,
                           &hTPMPolicy);
   DBG("Getting the TPM policy", result);

// Then we set the default owner's authorization as its secret
result = Tspi_Policy_SetSecret(hOldTPMPolicy,
                    TSS_SECRET_MODE_SHA1,
                    3,"123");
```

```
    DBG("Setting the policy secret to 123",result);

//Create a new policy object and put a new password in it
result=Tspi_Context_CreateObject(hContext,
                                 TSS_OBJECT_TYPE_POLICY
                                 TSS_POLICY_USAGE,
                                 &NewPolicy);
    DBG("Create a new policy", result):

result = Tspi_Policy_SetSecret(hOldTPMPolicy,
                                 TSS_SECRET_MODE_SHA1,
                                 4,"abcd");
    DBG("Set new secret to abc",result);

// Change the password to the one in the new policy
result=Tspi_ChangeAuth(hTPM,0x0000000,
                               hNewPolicy);

//Note: 0x0000000 is the parent of the TPM
//(which has no parent)
    DBG("ChangeAuthOfTPM", result);
```

Tell the context the TPMs old password, which is "123". Then create a new policy with a new password, stored in the variable newPassword, and 20 bytes long. (perhaps it is "12345678901234567890"). Next, change the TPMs owner authorization from "123" to "12345678901234567890" by using the Tspi_ChangeAuth command. Normally in a change authorization command, you must first provide the current authorization. But in this case, the context has already been told that the current authorization is "123".

Chapter 9

Creating Keys

There are several types of keys you may want to create, each with a number of different attributes. Key types include

1. Attestation Identity Keys (AIKs), used to quote the current values of PCRs contained in a TPM's Platform Configuration Registers, TickCount, and to certify other non-migratable keys

2. Signing Keys, which are used to sign data. There are 4 signature schemes used with signing keys. TSS_SS_RSASSAPKCS11V15_INFO is one of the most interesting, as it only signs tagged structures. As such, it can be used in place of an AIK without worry about quote spoofing.

3. Storage keys, which are used as key encrypting keys, either to store other asymmetric keys or to "seal" data. Storage keys are always 2048 bit RSA keys.

4. Binding keys are a type of key encrypting key that only stores small amounts of data. Typically these are used to store AES keys.

5. Legacy keys can be used to both encrypt and sign data. They are allowed only to enable legacy software to still use keys stored in hardware, but use of legacy keys is not recommended.

Keys are a type of object in TrouSerS, and the creation of keys follows a standard procedure:

- Create the object

- Set the general attributes of the object
- Create a policy object for the object
- Associate the policy object with the object
- Set any needed attributes of the object
- Instantiate the object in silicon (let the TPM fill in the public/private key)

In the case of a key object, typically the user will establish its type at object creation, create a policy object to be associated with the key object, set the password of the policy object, associate the policy object with the key object, and then send the object into the TPM, where the TPM will create the actual asymmetric key, wrap the key with its parents public key, and export the resultant key blob to the user.

Note that this is the only time the user can get the key blob out of the TPM, so if the key blob is not saved (either to a file or registered to the Trusted Core Services (TCS) daemon), once the key handle is lost, the key is lost forever as well. Typically at this point the key is registered.

Keys can be registered either to a system pool or to a user pool of keys. They are registered with a UUID of the form {0,0,0,0,0,{0,0,0,0,x,y}}. Here x and y are bytes (as are all the zeros), but if x is a zero, then the value of y is reserved for pre-registered UUIDs found in the header files or the TSS spec. If x has a value of 1, then the UUIDs are reserved for "owner evict" keys - keys that are stored permanently inside the TPM by the owner, and which can only be evicted by the owner, or by clearing the owner. This leaves a LOT of UUIDs for you to use, though.

Creating a key is pretty easy, as shown in the next two examples:

9.1 Sample Code: Creating a Bind Key

```
#define BACKUP_KEY_UUID {0,0,0,0,0,{0,0,0,2,10}}
TSS_HKEY        hBackup_Key;
TSS_UUID        MY_UUID=BACKUP_KEY_UUID;
TSS_HPOLICY     hBackup_Policy;
TSS_FLAG        initFlags;
```

```
    BYTE            *pubKey;
    UINT32           pubKeySize;
    FILE            *fout;
// Create a policy for the new key. Set its password to "123"
    result=Tspi_Context_CreateObject(hContext,
                                     TSS_OBJECT_TYPE_POLICY
                                    0, &hBackup_Policy);
        DBG("Create a backup policy object",result);

    result=Tspi_Policy_SetSecret(hBackup_Policy,
                                 TSS_SECRET_MODE_PLAIN,
                                 3,"123");
          DBG("Set backup policy object secret",result);

    initFlags = TSS_KEY_TYPE_BIND |
                TSS_KEY_SIZE_2048 |
                TSS_KEY_AUTHORIZATION |
                TSS_KEY_NOT_MIGRATABLE;

    result=Tspi_Context_CreateObject( hContext,
                                 TSS_OBJECT_TYPE_RSAKEY,
                                 initFlags, &hESS_Bind_Key );
          DBG("Create the key object", result);

    // Set the padding type
    result = Tspi_SetAttribUint32(hESS_Bind_Key,
             TSS_TSPATTRIB_KEY_INFO,
             TSS_TSPATTRIB_KEYINFO_ENCSCHEME,
             TSS_ES_RSAESPKCSV15);
       DBG("Set the key's padding type", result);

    // Assign the key's policy to the key object
    result=Tspi_Policy_AssignToObject( hESS_Policy,hESS_Bind_Key);
       DBG("Assign the key's policy to the key", result);

    // Create the key, with the SRK as its parent.
    printf("Creating the key could take a while\n");
    result=Tspi_Key_CreateKey(hESS_Bind_Key,
```

```
                                    hSRKey, 0);
   DBG("Asking TPM to create the key", result);

// Once created, I register the key blob so I can retrieve it later
result=Tspi_Context_RegisterKey(hContext,
                                     hESS_Bind_Key,
                                    TSS_PS_TYPE_SYSTEM,
                                    ESS_BIND_UUID,
                                    TSS_PS_TYPE_SYSTEM,
                                    SRK_UUID);
  DBG("Register the key for later retrieval", result);

printf("Registering key blob for later retrieval\r\n");

// Now that the key is registered, I also want to store the public
// portion of the key in a file for distribution
// This is done in two parts:
// 1) Get the public key and stuff it into pubKey

result=Tspi_Key_LoadKey(hESS_Bind_Key,hSRKey);
      DBG("Load key in TPM", result);
result=Tspi_Key_GetPubKey(hESS_Bind_Key,
                                &pubKeySize, &pubKey);
      DBG("Get public portion of key", result);

// 2) Save it in a file. The file name will be
//    "BackupESSBindKey.pub"

fout=fopen( "BackupESSBindKey.pub", "wb");
if( fout != NULL)
{
   write(fileno(fout),pubKey,pubKeySize);
     printf("Finished writing BackupESSBindKey.pub\n");
   fclose(fout);
}
else
{
  printf("Error opening BackupESSBindKey.pub \r\n");
```

```
}

// CLEAN UP
Tspi_Policy_FlushSecret(hESS_Policy);
```

9.2 Sample Code: Creating a Signing Key

```
#define SIGN_KEY_UUID {0,0,0,0,0,{0,0,0,2,11}}
TSS_HKEY        hSigning_Key;
TSS_UUID        MY_UUID=SIGN_KEY_UUID;
TSS_FLAG        initFlags;
BYTE            *pubKey;
UINT32          pubKeySize;
FILE            *fout;


initFlags = TSS_KEY_TYPE_SIGN |
            TSS_KEY_SIZE_2048 |
            TSS_KEY_NO_AUTHORIZATION |
            TSS_KEY_NOT_MIGRATABLE;
result=Tspi_Context_CreateObject( hContext,
              TSS_OBJECT_TYPE_RSAKEY,
              initFlags, &hSigning_Key );
       DBG("Create the key object", result);

// Set the padding type
result = Tspi_SetAttribUint32(hSigning_Key,
         TSS_TSPATTRIB_KEY_INFO,
         TSS_TSPATTRIB_KEYINFO_ENCSCHEME,
         TSS_SS_RSAESPKCSV15_INFO);
   DBG("Set the key's padding type", result);

// Create the key, with the SRK as its parent.
printf("Creating the key could take a while\n");
result=Tspi_Key_CreateKey(hSigning_Key,
                          hSRKey, 0);
```

```
    DBG("Asking TPM to create the key", result);

// Once created, I register the key blob so I can retrieve it later
result=Tspi_Context_RegisterKey(hContext,
                                hESS_Bind_Key,
                                TSS_PS_TYPE_SYSTEM,
                                MY_UUID,
                                TSS_PS_TYPE_SYSTEM,
                                SRK_UUID);
DBG("Register the key for later retrieval", result);

printf("Registering key blob for later retrieval\r\n");

// Now that the key is registered, I also want to store the public
// portion of the key in a file for distribution
// This is done in two parts:
// 1) Get the public key and stuff it into pubKey

result=Tspi_Key_LoadKey(hSigning_Key,hSRKey);
    DBG("Load key in TPM", result);

result=Tspi_Key_GetPubKey(hSigning_Key,
                          &pubKeySize,
                          &pubKey);
    DBG("Get public portion of key", result);

// 2) Save it in a file. The file name will be
//    "SigningKey.pub"
fout=fopen( "SigningKey.pub", "wb");
if( fout != NULL)
{
    write(fileno(fout),pubKey,pubKeySize);
    printf("Finished writing SigningKey.pub\n");
  fclose(fout);
}
else
{
   printf("Error opening SigningKey.pub \r\n");
```

```
}

// CLEAN UP
Tspi_Policy_FlushSecret(hSigning_Key_Policy);
```

You will note that in both cases there is code to extract the public key from the key object. This is very important, as in most cases it is important to send the public key to another party, and it is important to certify the public key. You will also note that the pubKey in this case is returned as a BYTE **. This means that the context allocated the memory for the public key and returned the size and a pointer to the memory location. In this case, pubKey is just a pointer, that gets passed into the routine and is associated with the allocated memory.

9.3 Sample Code: Unregistering a key

After a key is registered, it may be necessary to unregister it. This is very easy.

```
result=Tspi_Context_GetKeyByUUID(hContext,
                                 TSS_PS_TYPE_SYSTEM,
                                 ESS_BIND_UUID,
                                &hESS_Bind_Key);
      DBG("Get key handle", result);

printf("Unregistering key\r\n");
result=Tspi_Context_UnregisterKey(hContext,
                                  TSS_PS_TYPE_SYSTEM,
                                  ESS_BIND_UUID,
                                 &hESS_Bind_Key);
    DBG("Unregister key",result);
```

Chapter 10

Binding Data

You must use a "bind" key to bind data. Binding data is a standard encryption with a public key. Since binding data only utilizes the public key, it is not necessary to involve the TPM in this operation. Indeed, most TPMs do not implement a means of binding data.

Its counterpart, Unbind, is a private key operation and can only be done by utilizing something that has access to the private key, usually the TPM.

Binding data usually consists of the following operations:

1. Create a bind key object and setting its public key
2. Create a bind data object
3. Bind cleartext into the object, using the binding key
4. Read out the encypted data from the bindobject

10.1 Code Binding Data

```
UINT32  ulDataLength;
BYTE    *rbgBoundData;

// Retrieve the public key
fin =fopen(Bind.pub", "r");
```

```
    read(fileno(fin),newPubKey,284);
fclose(fin);

// Create a key object
result=Tspi_Context_CreateObject( hContext,
                        TSS_OBJECT_TYPE_RSAKEY,
                        initFlags, &hESS_Bind_Key );
    DBG("Tspi_Context_CreateObject BindKey",result);

// Feed the key object with the public key read from the file
result=Tspi_SetAttribData(hESS_Bind_Key,
          TSS_TSPATTRIB_KEY_BLOB,
          TSS_TSPATTRIB_KEYBLOB_PUBLIC_KEY,
          284, newPubKey);
  DBG("Set Public key into new key object", result);

// Read in the data to be encrypted
fin=fopen(AES.key","rb");
    read(fileno(fin),encData,7);
fclose(fin);

// Create a data object , fill it with clear text and then bind it.
result=Tspi_Context_CreateObject(hContext,
                              TSS_OBJECT_TYPE_ENCDATA,
                              TSS_ENCDATA_BIND,
                             &hEncData);
       DBG("Create Data object",result);

result=Tspi_Data_Bind( hEncData,
                             hESS_Bind_Key,
                             7,encData);
       DBG("Bind data",result);

// Get the encrypted data out of the data object
result=Tspi_GetAttribData( hEncData,
         TSS_TSPATTRIB_ENCDATA_BLOB,
         TSS_TSPATTRIB_ENCDATABLOB_BLOB,
      &ulDataLength,&rgbBoundData);
```

DBG("Get encrypted data", result);

// Write the encrypted data out to a file called Bound.data
fout=fopen("Bound.data", "wb");
write(fileno(fout),rgbBoundData,ulDataLength);
fclose(fout);

Chapter 11

Unbinding data

Of course if all we could do was bind data, and never decrypt it, it would be as useful as a write only hard drive.

Reading data that has been bound is almost the same as creating bound data. We use the UUID that we registered the binding key with to retrieve its key handle. Then we load it into the TPM. Next we get its policy, and set its password. We create a bindData object, set the encrypted data as its bound data attribute, and then perform an Tspi_UnBind command referencing the loaded key. This is shown in the next example.

Unbinding data usually consists of the following operations

1. Get the handle to a bind key, load it, and set its policy
2. Create a bind data object and set the encrypted data into it
3. Unbind the bound object using the private portion of the binding key
4. Write the cleartext

11.1 Example Unbind Data

```
UINT32   encLen=256;
BYTE     encryptedData[256];
BYTE     *rgbDataunBound;
UINT32   ulDataLength;
```

```
FILE        fin;
FILE        fout;

//Read in the encrypted data from the file
     fin=fopen("Bound.data","rb");
       read fileno(fin),encryptedData,ulDataLength);
     fclose(fin);

//Create a new data object
   result=Tspi_Context_CreateObject(hContext,
                        TSS_OBJECT_TYPE_ENCDATA,
                        TSS_ENCDATA_BIND, &hEncData);
      DBG("Created Data object", result);
result = Tspi_Context_GetKeyByUUID(hContext,
                                   TSS_PS_TYPE_SYSTEM,
                                  BIND_UUID,
                                 &hUnBind_Key);
      DBG("Get unbinding key",result);

result = Tspi_Key_LoadKey(hRecovered_UnBind_Key,
                                    hSRKey);
      DBG("Loaded Key",result);

// Use the Unbinding key to decrypt the encrypted data
// in the BindData object, and return it
result=Tspi_Data_UnBind(hNewEncData,
                                     hRecoveredUnbindKey,
                                   &ulDataLength,
                                   &rgbDataUnBound);
      DBG("Unbind",result);
```

Chapter 12

Sealing data

Any non-migratable storage key, including the SRK, can be used to seal data. Sealing data differs from binding data in a number of important ways. First, the encryption must be done by the TPM. Secondly, the decryption depends on not just a password (the Policy of the decrypting key), but also on certain TPM Platform Configuration Register (PCR) values being in the correct state, as chosen during the encryption phase. Thirdly, during encryption, the current values of PCRs are encrypted as a "history" attribute of the data object. This history is securely stored with the data object, and is the reason that only non-migratable keys are allowed to do a SEAL command. NOTE: If you don't care about this history file, you can accomplish all the other purposes of a seal command (i.e. encrypting data so that it can only be decrypted when the system is in a particular state) by locking the BindKey (during creation) so that it can only be used when selected PCRs are in a prescribed state.

12.1 Example code: Sealing data

```
char TypePass[12]="My Password";

    result=Tspi_Context_CreateObject(hContext,
                                     TSS_OBJECT_TYPE_PCRS,
                                     0,
                                    &hPCRs);
```

```
    DBG("Creating a PCR object",result);

result=Tspi_TPM_PcrRead(hTPM,8,
                        &ulPcrLen,
                        &rbgPcrValue);
  DBG("Read the current value for PCR 8",result);

result=Tspi_PcrComposite_SetPcrValue(hPcrs,8,
                                     20,
                                     rbgPcrValue);
    DBG("Set the value read from PCR 8
         as the sealing value of PCR8",result);

result=Tspi_TPM_PcrRead(hTPM,9,
                        &ulPCcrLen,
                        &rbgPcrValue);
    DBG("Read the current value of PCR 9",result);

result=Tspi_PcrComposite_SetPcrValue(hPcrs,9,20,
                                     rbgPcrValue);
    DBG("Set the value read from PCR 9
         to be the sealing value of PCR9",result);

// Create an data object for sealing
result=Tspi_Context_CreateObject(hContext,
                        TSS_OBJECT_TYPE_ENCDATA,
                        TSS_ENCDATA_SEAL,
                       &hEncData);
    DBG("Create data object for sealing", result);

result=Tspi_Data_Seal(hEncData,
                      hSRKey,
                      strlen(TypePass),
                      TypePass,hPcrs);
DBG("Sealed my password, using the SRK key,
       to PCR 8",result);

// Now we get the encrypted data out of the data object
```

```
// and write it to a file
result= Tspi_GetAttribData( hEncData,
                    TSS_TSPATTRIB_ENCDATA_BLOB,
                    TSS_TSPATTRIB_ENCDATABLOB_BLOB,
                  &encDataSize,
                  &encData);
    DBG("Get the encrypted (sealed data", result);

// write the encrypted (sealed) data out to the file name
fout=fopen("owner_auth.pass", "wb");
if(fout != NULL)
{
    write(fileno(fout),encData,encDataSize);
    fclose(fout);
}
else
{
    printf("Error opening Owner_Auth.pass.");
}
```

Of course once you have sealed data, you will also want to unseal the data. This can be easily accomplished by (almost) reversing the steps used to seal the data.

12.2 Example code: Unsealing Data

```
UINT32      outlength;
BYTE       *outstring;
BYTE        EncryptedData[312];

memset(EncryptedData,0,312);

//Read in the sealed data
    fin=fopen("owner_auth.pass","rb");
          read(fileno(fin), EncryptedData,312);
    fclose(fin);
```

```
//Create a data object to unseal with
result=Tspi_Context_CreateObject(hContext,
                                 TSS_OBJECT_TYPE_ENCDATA,
                                 TSS_ENCDATA_SEAL,
                                &hRetrieveData);
   DBG("Create data object for unsealing", result);

result=Tspi_SetAttribData(hRetrieveData,
                          TSS_TSPATTRIB_ENCDATA_BLOB,
                          TSS_TSPATTRIB_ENCDATA_BLOB,
                          312,EncryptedData);

result=Tspi_Data_Unseal(hRetrieveData,
                           hSRK,
                          &outlength,
                          &outstring);

//Now outstring has the decrypted data.
```

Chapter 13

Signing

Identity keys cannot be used for general purpose signing - they can only sign three structures - a quote structure, a key certificate structure and a tickstamp structure. As a result there is another kind of key, a signing key, which can be used to sign arbitrary data.

The signing key can be created to use one of 4 different signing schemas:

1. None
2. TSS_SS_RSASSAPKCS15V2_SHA1
3. TSS_SS_RSASSAPKCS15V2_DER
4. TSS_SS_RSASSAPKCS15V2_INFO

The last of these will only allow the signing of structures, but when used in any of the commands that already sign structures, the signing key behaves the same way that an identity key behaves. In the case of the Tspi_Hash_Sign command, this signature scheme causes the function to first creates a SIGN_INFO structure from the presented data before signing it. This prevents the signing key from ever being used to spoof a structured signature.

Signing is generally as easy as sealing data. Load a key to sign with, get its policy and load its password, so that the context knows the password associated with the key. Next create a hash object, and load the object with the data to be hashed and signed. Lastly hash and sign the data.

13.1 Example follows: Signing Data

```
//Get the Signing key handle from its UUID, and then load it
   result = Tspi_Context_GetKeyByUUID(hContext,
                                      TSS_PS_TYPE_SYSTEM,
                                      SIGNING_UUID,
                                     &hSigning_Key);
      DBG("Get Key By UUID", result);

   //Load the private key into the TPM using its handle
   result=Tspi_KEY_LoadKey(hSigning_Key,hSRKey);
      DBG("Load Key",result);

   //Create a hash object
   result=Tspi_Context_CreateObject(hContext,
                                    TSS_OBJECT_TYPE_HASH,
                                    TSS_HASH_SHA1,
                                   &hHashToSign);
      DBG("Create Hash object", result);

   // Read in a file to hash
   pubKeyLength=filelength("file.data");
   fin=fopen("file.dat","rb");
      read(fileno(fin),pPubKey,pubKeyLength);
   fclose(fin);

   //Hash the data using SHA1
   result=Tspi_Hash_UpdateHashValue(hHashToSign,
                                    pubKeyLength,pPubKey);
      DBG("Hash the public key",result);

   //Sign the resultant hash object
   result=Tspi_Hash_Sign(hHashToSign,hSigning_Key,
                         &ulSignatureLength,
                         &rgbSignature);
      DBG("Sign",result);

   //Write the resultant signature to a file called Signature.dat
```

```
fout=fopen("Signature.dat","wb");
   write(fileno(fout),rgbSignature,ulSignatureLength);
fclose(fout);
```

13.2 Example Code: Verifying a Signature

Now that we have created a signature, it is likely that someone will want to verify that the signature is correct. This is a public key operation, and there are two ways of doing it with TrouSerS. TrouSerS can use a purely software verification of the signature, or it can use the TPM (with owner authorization) to perform a signature verification.

In both cases, one needs to have a signature, and the file that was signed. The software then hashes the file that was signed, and provides TrouSerS with the hash of the file, the signature, and the public key which corresponds to the key used to sign the file.

```
// Create a Hash Object so as to have something to sign
// Create a generic Hash object
result=Tspi_Context_CreateObject(hContext,
                                 TSS_OBJECT_TYPE_HASH,
                                 TSS_HASH_SHA1,
                                &hHashToSign);
DBG("Create Hash object", result);

pubKeyLength=filelength(filename);
fin=fopen(filename,"rb");
if(fin != NULL){
   read(fileno(fin),pPubKey,pubKeyLength);
fclose(fin);
}
else {
printf("ERROR opening file for reading\r\n");
}
// Hash the data using SHA1//
result=Tspi_Hash_UpdateHashValue(hHashToSign,
                                                pubKeyLength,
                                                pPubKey);
```

```
    DBG("Hash in the public key", result);

// We are going to create a Verify key
// Here I determine the key will be a Signing key of 2048 bits,
// non-migratable, with no authorization.
fin = fopen("GH.lis", "rb");
    read(fileno(fin),pubVerifyKey,284);
// Read the Second key in my file
    read(fileno(fin),pubVerifyKey,284);
fclose(fin);
fin = fopen(filenameOut, "rb");
    read(fileno(fin), Signature, 256);
fclose(fin);

initFlags = TSS_KEY_TYPE_SIGNING |
            TSS_KEY_SIZE_2048 |
            TSS_KEY_NO_AUTHORIZATION |
            TSS_KEY_MIGRATABLE

// Create the key object
result=Tspi_Context_CreateObject( hContext,
                        TSS_OBJECT_TYPE_RSAKEY,
                        initFlags,
                      &hVerify_Key );
    DBG("Tspi_Context_CreateObject Verify_Key",result);

result=Tspi_SetAttribData(hVerify_Key,
            TSS_TSPATTRIB_KEY_BLOB,
                TSS_TSPATTRIB_KEYBLOB_PUBLIC_KEY,
            pubSignKeyLength,pubVerifyKey);
    DBG("SetPubKey in Verify_Key", result);

result=Tspi_Hash_VerifySignature(hHashToSign,
                                 hVerify_Key,
                                 256,
                                 Signature);
    DBG("Verify", result);
```

```
if(result==0)
    printf("Signature verified using TSS software\n");
// Now again using the TPM for Verification of the Signature
// First get the TPM owner auth
     //Get TPM Policy Object
result = Tspi_GetPolicyObject(hTPM,
                                   TSS_POLICY_USAGE,
                                  &hTPM_Policy);
    DBG("Tspi_GetPolicyObject", result);

     //Set Secret
result = Tspi_Policy_SetSecret(hTPM_Policy,
                               TSS_SECRET_MODE_PLAIN,
                              3,(BYTE *)"123");
    DBG("Tspi_Policy_SetSecret", result);

// Now we create the MigObject to hold the signature and digest
// for verification
Tspi_Context_CreateObject(hContext,
                          TSS_OBJECT_TYPE_MIGDATA,
                          0, hMigData);
// Put in new hash value here and see what happens
// Hash the data using SHA1//
result=Tspi_Hash_GetHashValue(hHashToSign,
                                   &hashSize,
                                   &myHash);
  DBG("Set initial hash value to all zeros", result);

Tspi_SetAttribData( hMigData,
            TSS_MIGATTRIB_TICKET_DATA,
            TSS_MIGATTRIB_TICKET_SIG_DIGEST,
            20,
            myHash);
Tspi_SetAttribData( hMigData,
           TSS_MIGATTRIB_TICKET_DATA,
           TSS_MIGATTRIB_TICKET_SIG_VALUE,
           256,
           Signature);
```

```
//Create Ticket
result = Tspi_TPM_CMKCreateTicket(hTPM,
                                  hVerify_Key,
                                  hMigData);
   DBG("Verify using CMKCreateTicket", result);

if(result==0)
     printf("Signature verified using the TPM\n");
```

Chapter 14

NVRAM

TPMs have a small amount of non volatile random access memory (NVRAM). This NVRAM can have the same controls for reading and writing as other TPM objects. In particular, the NVRAM can be set so that it can only be written to when one set of PCRs is in a particular state, only read from when another set of PCRs is in a particular state, can have passwords set for read or write, and can be set so that read or write permissions are under TPM owner control. In order to create a region of the TPM non volatile memory for usage, the context has to know the TPM owner authorization. A non volatile object is created, and it attributes, including index, size, and security, are set. Once created, the non volatile object is sent to the TPM to be instantiated in silicon.

14.1 Example code: Creating NVRAM space

```
TSS_HNVSTORE hNVStore;
  //Create a NVRAM object
  result=Tspi_Context_CreateObject(hContext,
                                   TSS_OBJECT_TYPE_NV,
                                   0, &NVStore);
     DBG("Create NV object", result);

  //Next its arbitrary index will be 0x00011101
  result=Tspi_SetAttributeUint32(hNVStore,
```

```
                                    TSS_TSPATTRIB_NV_INDEX,
                                    0,0x00011101);
    DBG("Set index",result);

    //Next its policy (only writeable by owner, generally readable)
    result=Tspi_SetAttributeUint32(hNVStore,
                        TSS_TSPATTRIB_NV_PERMISSIONS,
                        0,
                        TPM_PER_OWNERWRITE);
    DBG("Set policy",result);

    //Set size as 40 bytes
    result=Tspi_SetAttributeUint32(hNVStore,
                    TSS_TSPATTRIB_NV_DATASIZE,
                    0,40);
    DBG("Set size",result);

    // In order to either instantiate or write to the NVRAM location, we now
    // will need TPM owner authorization.
    // First we will get the TPM policy, and then we will set its secret
    result=Tspi_GetPolicyObject(hTPM,
                                    TSS_Policy_USAG,
                                   &hTPMPolicy);
    DBG("Get TPM Policy",result);

    result= Tspi_Policy_SetSecret(hTPMPolicy,
                        TSS_SECRET_MODE_PLAIN,
                        3, "123");
    DBG("Set secret to 123",result);

    // Finally we can create the NVRAM space in the TPM
    result=Tspi_NV_DefineSpace(hNVStore,0,0);
    DBG("Created NV space",result);
```

Once created, we want to write data into the NVRAM space. We created the NVRAM index to require TPM owner authorization in order to write to the NVRAM. The TPM knows this, but TrouSerS does not. As a result, we have to create a policy object for the NVRAM object, give it the TPM

owner authorization secret, and then associate it with the NVRAM object. Once this is accomplished, we can write into the NVRAM.

14.2 Example code: Writing to NVRAM space

```
TSS_HNVSTORE     hNVStore;
TSS_HPOLICY      hNewPolicy;
char             dataToStore[6]="Data."

//Create a NVRAM object
result=Tspi_Context_CreateObject(hContext,
                                 TSS_Object_TYPE_NV,
                                 0, &NVStore);
   DBG("Create NV object", result);

//Next its arbitrary index will be 0x00011101
result=Tspi_SetAttributeUint32(hNVStore,
                       TSS_TSPATTRIB_NV_INDEX,
                       0,
                       0x00011101);
   DBG("Set index",result);

//Next its policy (only writeable by owner, generally readable)
result=Tspi_SetAttributeUint32(hNVStore,
                       TSS_TSPATTRIB_NV_PERMISSIONS,
                       0,
                       TPM_PER_OWNERWRITE);
   DBG("Set policy",result);

//Set size as 40 bytes
result=Tspi_SetAttributeUint32(hNVStore,
                           TSS_TSPATTRIB_NV_DATASIZE,
                           0,
                           40);
   DBG("Set size",result);
```

```
// In order to write to this NVRAM location, we now
// will need TPM owner authorization.
// First we will get the TPM policy, and then we will set its secret
result=Tspi_Context_CreateObject(hContext,
                                 TSS_OBJECT_TYPE_POLICY,
                                 TSS_POLICY_USAGE,
                                 &hNewPolicy);
   DBG("Get TPM Policy",result);

result= Tspi_Policy_SetSecret(hNewPolicy,
                              TSS_SECRETMODE_PLAIN,
                              3,"123");
   DBG("Set secret to 123",result);

result = Tspi_Policy_AssignToObject(hNewPolicy,
                                    hNVStore);
   DBG("Assigned policy to NV object",result);

// Finally we can write to the NVRAM space
result=Tspi_NV_WriteValue(hNVStore,0,18,
                          dataToStore);
   DBG("Wrote to the NVRAM",result);
```

Now once we have written the data, we will need to read it.

14.3 Example code: Reading from NVRAM space

```
TSS_HNVSTORE    hNVStore;
char            dataStore[19];

memset(dataStore,0,19);

//Create a NVRAM object
result=Tspi_Context_CreateObject(hContext,
                TSS_Object_TYPE_NV,0,NVStore);
```

```
    DBG("Create NV object", result);

  //Next its arbitrary index will be 0x00011101
  result=Tspi_SetAttributeUint32(hNVStore,
                  TSS_TSPATTRIB_NV_INDEX,
                  0,0x00011101);
    DBG("Set index",result);

  //Next its policy (only writeable by owner, generally readable)
  result=Tspi_SetAttributeUint32(hNVStore,
                        TSS_TSPATTRIB_NV_PERMISSIONS,
                        0,
                       TPM_PER_OWNERWRITE);
    DBG("Set policy",result);

  //Set size as 40 bytes
  result=Tspi_SetAttributeUint32(hNVStore,
                  TSS_TSPATTRIB_NV_DATASIZE,
                  0,40);
    DBG("Set size",result);

  // No authorization needed to read this NVRAM location
  // Finally we can read to the NVRAM space
  result=Tspi_NV_ReadValue(hNVStore,0,18,
                           &dataStore[0]);
    DBG("Read value from NV",result);
```

14.4 Destroying NVRAM space

```
// Create a NVRAM object //
    result=Tspi_Context_CreateObject(hContext,
                                TSS_OBJECT_TYPE_NV,
                                0,
                               &hNVStore);
    DBG("Create an NVRAM object", result);

    //Next its arbitrary index will be 0x00011102
```

//(00-FF are taken, along with 00011600).

result=Tspi_SetAttribUint32(hNVStore,
TSS_TSPATTRIB_NV_INDEX,
0, 0x00011101);
DBG("Set NVRAM index", result);

// In order to destroy a NVRAM location, owner_auth is required.
// We need to get the TPM policy object and put it in there.
result=Tspi_GetPolicyObject(hTPM,
TSS_Policy_USAG,
&hTPMPolicy);
DBG("Get TPM Policy",result);

result= Tspi_Policy_SetSecret(hTPMPolicy,
TSS_SECRET-MODE_PLAIN,
3, "123");
DBG("Set the TPM policy secret to 123", result);

// Now I will release NVRAM location
printf("Release NVRAM space\r\n");
result=Tspi_NV_ReleaseSpace(hNVStore);
DBG("Release NV space", result);

Chapter 15

PCRs

There are 24 Platform Configuration Registers (PCRs) in a modern TPM. PCRs 0-6 are used to record the preboot history. PCRs 17,18, and 19 are used to record the Dynamic Root of Trust Measurement chain. PCR 10 is used by the Integrity Measurement Architecture (IMA) to record libraries, configuration files and executables as they are loaded by the kernel.

The following code examples demonstrate how to read the values of PCRs, how to change the values of PCRs, and how to read the log that was used to create the PCR value.

15.1 Example of reading PCR values

```
// MAIN BODY VARIABLES
   BYTE     *rgbPcrValue;
   UINT32   ulPcrValueLength=20;
   int      i;
   UINT32   j;

   // ENTER MAIN BODY
   // STEP 1 //
   //Read and print out current PCR value
   for(j=0;j<24;++j)
   {
     result = Tspi_TPM_PcrRead( hTPM, j,
                                &ulPcrValueLength,
```

```
                                        &rgbPcrValue );
    printf("PCR %02d ",j);
    for(i=0 ; i<19;++i)
    {
        printf("%02x",*(rgbPcrValue+i));
    }
    printf("\n");
}
```

15.2 Example of changing PCR values

```
// MAIN BODY VARIABLES
    UINT32    PCR_result_length;
    BYTE      *Final_PCR_Value;
    BYTE      valueToExtend[250];
    int       count = 0;
    int       pcrToExtend = 0;

    // ENTER MAIN BODY
    memset(valueToExtend,0,250);

    // Display each command line argument.
    printf("\n Command line arguments:\n");

    for (count = 0; count < argc; count++)
    {
        printf(" argv[%d] %s\n", count, argv[count]);
    }

    // Examine command line arguments.

    if (argc >= 3)
    {
        if (strcmp(argv[1],"-p") == 0)
        {
            pcrToExtend = atoi(argv[2]);
```

```
        if(pcrToExtend < 0 || pcrToExtend >23)
        {
            printMenu();
            return 0;
        }
        if (argc == 5)
        {
            if(strcmp(argv[3],"-v") == 0)
            {
                memcpy(valueToExtend,
                        argv[4], strlen(argv[4]));
            }
        }
        else
        {
        // Use default value.
            memcpy(valueToExtend,
                    "abcdefghijklmnopqrst", 20);
        }
}
else
{
    printMenu(); return 0;
}
  //Extend the value
result = Tspi_TPM_PcrExtend(hTPM,
                            pcrToExtend,20,
                            (BYTE *)valueToExtend,
                            NULL,
                           &PCR_result_length,
                           &Final_PCR_Value);
    DBG("Extended the PCR", result);

// Clean up //
Tspi_Context_FreeMemory(hContext, NULL);
Tspi_Context_Close(hContext);
return 0;
}
```

```
void printMenu()
{
   printf("\nChangePCRn Help Menu:\n");
   printf("\t-p PCR register to extend (0-23)\n");
    printf("\t-v value to be extended into PCR(abc..)\n");
  printf("\tNote: -v argument is optional
             and a default value will be used
             if no value is provided\n");
    printf("\tExample: ChangePCRn -p 10 -v abcdef\n");
}
```

15.3 Example of reading PCR value logs

```
UINT32              ulpcrIndex = 9;
UINT32              ulStartNumber=0;
UINT32              ulEventNumber=15;
TSS_PCR_EVENT      *prgbPcrEvents;
char                eventBlank[256];
int                 i;

Tspi_TPM_GetEvents(hTPM,pcrIndex,ulStartNumber,
                   (UINT32 *)&pcrNumber,&prgbPcrEvents);

for(i=0; i<pcrnumber;++i)
{
   memset(eventBlank,0,256);
   memcpy(eventBlank,prgbPcrEvents[i].rgbEvent,
            prgbPcrEvent[i].ulEventlog);
   printf("Event %d, is %s \n", i, eventBlank);
}
```

Chapter 16

RNG

The TPM contains a Random Number Generator (RNG) . The RNG provides entropy that can be used to seed an operating systems entropy store or obtain a random password. Accessing the entropy is easy. Simply ask the TPM for a random number of a particular size. If too big a number is requested, the operation may take a very long time, but eventually a random number will be returned.

16.1 Example of asking for a random number:

```
// MAIN BODY VARIABLES BYTE *randomBytes;
        FILE *fout;
        int      i;
        int      numRandomBytesOut=atoi(argv[1])


   // ENTER MAIN BODY
   if((randomBytes=
                    (BYTE *)malloc(numRandomBytesOut))
                   ==NULL)
        {
          printf("ERROR ALLOCATING randomBytes\n");
          return 1;
```

```
    }

// Ask the TPM for a random number and put it in the random
// Bytes Variable
 result=Tspi_TPM_GetRandom(hTPM,
                           numRandomBytesOut,
                          &randomBytes);
   DBG("Get Random number from TPM", result);

 fout = fopen("random.out","wb");
    write(fileno(fout),
          randomBytes,
          numRandomBytesOut);
 fclose(fout);
```

Chapter 17

HASH

The TPM is not a particularly fast crypto co-processor, so if a lot of hashes are required the TPM may not be the best solution. A crypto library to perform hashes is almost always the best solution. But if only a few hashes are needed, you can use the hash engine in the TPM.

In order to use the TPM hash engine, you must create a hash object, do a "hash extend" into the hash object, and then read the hashed data out of the object. (This is very similar to using the bind or seal commands with a data object.)

17.1 Example code: Hashing

```
// Name: HashThis
   // Parameters: pubKey - public key to hash
   // pubKeysize - size of public key
   // hash - buffer to store hash value
   // Return: NONE
   // Purpose: Use TPM to generate SHA1 hash of public key.

   void HashThis(TSS_HCONTEXT hContext,
                 BYTE *pubKey,
                 UINT32 pubKeysize,
                 BYTE hash[20])
   {
```

```
    TSS_RESULT    result;
    BYTE          *digest;
    UINT32        digestLen;
    TSS_HHASH     hHashOfESSKey;
    BYTE          initialHash[20];

    memset(initialHash,0,20);

// Create a generic Hash object
result=Tspi_Context_CreateObject(hContext,
                                 TSS_OBJECT_TYPE_HASH,
                                 TSS_HASH_SHA1,
                                 &hHashOfESSKey);
    DBG("Create Hash object", result);

// Hash the data using SHA1
result=Tspi_Hash_UpdateHashValue(hHashOfESSKey,
                                 pubKeysize,
                                 pubKey);
      DBG("Hash in the public key", result);

   result=Tspi_Hash_GetHashValue(hHashOfESSKey,
                                 &digestLen,
                                 &digest);
       DBG("Get the hashed result", result);

// so now the digest has the hash of the ESS public key in it
   memcpy(hash,digest,20);
}
```

Chapter 18

Owner Evict Keys

Owner Evict keys are keys that remain resident on a TPM even when the power is off. They reside in TPM NVRAM, unlike most keys which reside encrypted on the hard disk. In this way, an owner evict key is similar to the Storage Root key (SRK), which also resides on TPM NVRAM.

Owner evict keys are called such because only the TPM owner is able to cause them to be evicted from the TPM, thus allowing the NVRAM to be used for other things. Owner evict keys are typically used for one of two specific purposes - using a key in the absence of access to the hard disk, or using a key without bothering to load it.

To date, we have succeeded in creating owner evict keys, but not in using the keys. The TrouSerS implementation for using owner evict keys doesn't currently work, but hopefully that will change in a near-future release. Owner authorization of the TPM is required to make a key an owner evict key, so you will note that we use "123" as the requisite password. The following example describes how owner evict keys are supposed to work:

18.1 Example code: Owner evict keys

```
// MAIN BODY VARIABLES
      TSS_HKEY        hMB_AIK_Key=0;
      TSS_FLAG        initFlags;
      TSS_HPOLICY     hMB_AIK_Policy;
      UINT32          blob_length;
```

```
   BYTE              *mb_aik_key_blob;
   BYTE              *pubKey;
   UINT32             pubKeySize;
   FILE              *fout;
   TSS_UUID           pUuidData;
   int                drcheck;
   char               mypass[21];
memset(wks,0,20);
printf("Entering CreateOEAIK_Temporary\r\n");
‘
// Now in order to create an AIK, I need to retrieve the TPM
// owner_auth and put it into the TPM policy, so it is
// available for the TSS to use.
// First we get a TPM policy object and set its secret to "123"

result=Tspi_GetPolicyObject(hTPM,
                                TSS_POLICY_USAGE,
                               &hTPM_Policy);
   DBG("GetTPM policy",result);

result=Tspi_Policy_SetSecret(hTPM_Policy,
                         TSS_SECRET_MODE_PLAIN,
                         3, (BYTE *)"123");
   DBG("Set TPM policy's secret to123", result);

//Create policy object for the key we are about to make
result= Tspi_Context_CreateObject(hContext,
                      TSS_OBJECT_TYPE_POLICY,
                      TSS_POLICY_USAGE,
                     &hMB_AIK_Policy);
DBG("Create a policy object for the key",result);

// Set its password to "123"
result=Tspi_Policy_SetSecret(hMB_AIK_Policy,
                     TSS_SECRET_MODE_PLAIN,
                     3, (BYTE *)"123");
DBG("Set the password to 123", result);
```

```
// Determine the key will be a Non Migratable
// Signing key of 2048 bits, using an authorization.
printf("Selecting the key to be a non-migratable
        signing key of 2048 bits\r\n");

initFlags = TSS_KEY_SIZE_2048 |
            TSS_OBJECT_TYPE_RSAKEY |
            TSS_KEY_TYPE_SIGNING |
            TSS_KEY_AUTHORIZATION |
            TSS_KEY_NOT_MIGRATABLE;
result=Tspi_Context_CreateObject( hContext,
                  TSS_OBJECT_TYPE_RSAKEY,
                  initFlags, &hMB_AIK_Key );
    DBG("Ask the TPM to create the requisite key
        - a non-migratable signing key,
          not an AIK", result);

result=Tspi_Policy_AssignToObject( hMB_AIK_Policy,
                                   hMB_AIK_Key);
    DBG("Assign policy to the key object", result);

// Create the key, with the SRK as its parent.
printf("Asking TPM to create a Non-migratable
        signing key. This may take awhile\n");

result=Tspi_Key_CreateKey(hMB_AIK_Key,
                          hSRKey, 0);
DBG("Create non-migratable signing key.", result);

// Once created, I register the key blob so I can retrieve it later
// Make the key an owner_evict key (owner auth is required
// to evict it - we need to have put the TPM owner_auth into the
// TPM policy first. We already did.
// Make the key an Owner_Evict key
// Load the key into the TPM

printf("Loading the key into the TPM\r\n");
result=Tspi_Key_LoadKey(hMB_AIK_Key, hSRKey);
```

```
    DBG("Load the key into the TPM", result);

// Tell the TPM to not allow anyone but the owner to evict it
printf("Store the key in non volatile storage in the
       TPM, such that only the owner can evict it\n");

result=Tspi_TPM_KeyControlOwner(hTPM,
    hMB_AIK_Key,
     TSS_TSPATTRIB_KEYCONTROL_OWNEREVICT,
    TRUE, &pUuidData);
DBG("Make key an owner evict key", result);

// In order to not fill up the TPM with keys, change the
// key so it is allowed to be evicted
   printf("In order to not fill up the TPM with keys,
        change the key so it is allowed to be evicted\n");

result=Tspi_TPM_KeyControlOwner(hTPM,
       hMB_AIK_Key,
      TSS_TSPATTRIB_KEYCONTROL_OWNEREVICT,
      FALSE, &pUuidData);
DBG("Make the key evictable", result);
// Now that the key is registered, I also want to store the
// public portion of the key in a file for distribution.
// This is done in two parts:
// 1) Get the public key and stuff it into pubKey.
printf("Storing public portion of key in
          MotherboardNMSK.pub\r\n");
result=Tspi_Key_GetPubKey(hMB_AIK_Key,
                          &pubKeySize,
                          &pubKey);
        DBG("Get public portion of key", result);

// 2) Save it in a file.
// The file name will be "MotherboardAIK.pub"
    fflush(stdout);
         fout=fopen( "MotherboardNMSK.pub", "wb");
    if(fout != NULL){
```

```
        write(fileno(fout), pubKey,pubKeySize);
        fclose(fout);
        printf("Finished writing file\n");
    }
    else {
       printf("ERROR opening file for writing\r\n");
    }
    printf("Save the encrypted key blob to
              MotherboardNMSK.blob");

  result=Tspi_GetAttribData(hMB_AIK_Key,
                            TSS_TSPATTRIB_KEY_BLOB,
                            TSS_TSPATTRIB_KEYBLOB_BLOB,
              &blob_length, &mb_aik_key_blob);
DBG("Get key blob, for removable media", result);

    fout=fopen( "MotherboardNMSK.blob", "wb");
    if(fout != NULL)
    {
       write(fileno(fout), mb_aik_key_blob, blob_length);
       fclose(fout);
       printf("Finished writing file\n");
    }
    else {
       printf("ERROR opening file for writing\r\n");
    }
```

Chapter 19

Key Migration

Manageability of keys requires a mechanism to move a key from one system to another, perhaps when upgrading a system or when a motherboard goes bad. Not everyone will take advantage of key migration, but it may be important to establish a key on a remote system that can be used to pass data back and forth.

The TPM is designed to allow for this use case. When a key is created, the creator can designate it as a migratable key. In this case, two passwords are needed for the key - one for key usage, and one for migration of the key. These passwords do not have to be the same.

In the following example, the usage password will be "Usage Password" and the migration password will be "Migration Password". A binding key will be created that is migratable, whose parent is the SRK.

The TPM owner has to "bless" all migration targets, by creating a Migration Ticket. We will provide the code to create this ticket.

We will be given the public portion of a remote TPM's SRK, and will then rewrap the binding key with the remote TPM's public SRK, passing in the pre-approved ticket for the remote TPM's public SRK.

19.1 Example: Making the ticket

```
// Set TPM Owner auth, so that you can make the ticket
   result=Tspi_GetPolicyObject(hTPM,
                               TSS_POLICY_USAGE,
                              &hTPM_Policy);
```

```
    DBG("Getting TPM Policy object", result);

result = Tspi_Policy_SetSecret(hTPM_Policy,
                               TSS_SECRET_MODE_PLAIN,3, 123");
    DBG("Set TPM policy's secret to 123", result);

// Read in the public key you want to "bless"
fin=fopen( "Storage.pub", "rb");
   read(fileno(fout),pubKey,284);
fclose(fin);

// Create the key object
result=Tspi_Context_CreateObject( hContext,
                                  TSS_OBJECT_TYPE_RSAKEY,
                                  initFlags,
                                 &hStorage_Key);
    DBG("Tspi_Context_CreateObject StorageKey",result);

// Set the key object's public key to be the one you just read in.
result=Tspi_SetAttribData(hStorage_Key,
                  TSS_TSPATTRIB_KEY_BLOB,
                  TSS_TSPATTRIB_KEYBLOB_PUBLIC_KEY,
                 &pubKeySize,
                 &pubKey);
     DBG("Set Public key into key object", result);

//Make the ticket
result=Tspi_TPM_AuthorizeMigrationTicket(hTPM,
                                         hStorage_Key,
                                         TSS_MS_REWRAP,
                                        &TicketLength,
                                        &rgbMigTicket);
     DBG("Make Ticket",result);

// Save the ticket to a file for later use
fout=fopen( "Ticket", "wb");
    write(fileno(fout),rgbMigTicket,TicketLength);
fclose(fout);
```

19.2 Sample Code: Use the ticket

```
initFlags = TSS_KEY_TYPE_SIGNING |
            TSS_KEY_SIZE_2048 |
            TSS_KEY_NO_AUTHORIZATION |
            TSS_KEY_MIGRATABLE;

   // Create the key object
   result=Tspi_Context_CreateObject( hContext,
                      TSS_OBJECT_TYPE_RSAKEY,
                      initFlags,&hMigrateStorageKey );
       DBG("Tspi_Context_CreateObject SigningKey",result);

   // I have to assign a migration policy to the key I am creating -
   // which has the handle hMigrateStorageKey
   // Create migration policy
   result = Tspi_Context_CreateObject(hContext,
                            TSS_OBJECT_TYPE_POLICY,
                            TSS_POLICY_MIGRATION,
                           &hMigPolicy);

   //Set Secret
   result = Tspi_Policy_SetSecret(hMigPolicy,
                           TSS_SECRET_MODE_PLAIN,
                           7,(BYTE *)"Migrate");

   //Assign migration policy
   result = Tspi_Policy_AssignToObject(hMigPolicy,
                                       hMigrateStorage_Key);

   // Now I finally create the key, with the SRK as its parent.
   printf("Creating key... this may take some time\n");
   result=Tspi_Key_CreateKey(hMigrateStorage_Key,
                                 hSRKey, 0);
         DBG("Create Key", result);
```

```
//Read in the ticket data
fin=fopen("ticket.dat","rb");
    read(fileno(fin),ticket,ticketLength);
fclose(fin);

// Create the migration blob
result=Tspi_Key_CreateMigrationBlob(
                                    hMigrateStorage_Key,
                                    hSRKey,
                                    ticketLength,
                                    ticket,
                                   &rnLength,
                                   &rn,
                                   &migBlobLength,
                                   &migBlob);
     DBG("Create ReWrapped Key",result);

result=Tspi_GetAttribData(hMigrateStorage_Key,
                    TSS_TSPATTRIB_KEY_BLOB,
                    TSS_TSPATTRIB_KEYBLOB_BLOB,
                   &blob_length,
                   &migrated_blob);
    DBG("Get the migrated blob",result);

fout=fopen("Migrated.blob", "wb");
    write(fileno(fout), migrated_blob, blob_length);
fclose(fout);

printf("Finished writing Migrated.blob\n");
```

This key blob can be loaded the way any other key blob is loaded: by creating a key object, loading the key blob into the object, and then loading the key.

19.3 Example: Load the migrated blob

```
TSS_HKEY hMigratedKey;
FILE        *fin;
char         migrated_blob[1024];
int          blob_length=MIGRATED_KEY_BLOB_SIZE;

memset(migrated_blob,0,1024);

fin=fopen("Migrated.blob", "rb");
   read(fileno(fin), migrated_blob, blob_length);
fclose(fin);

result=Tspi_Context_LoadKeyByBlob(hContext,hSRKey,
                                  blob_length,
                                  rgbBlobData,
                                  hMigratedKey) ;
      DBG("Load the migrated blob",result);
```

Chapter 20

Programming Challenge

1. Create a public / private Binding key, called EnterpriseBackup
2. Register the key
3. Create a file with the public portion (EnterpriseBackup.pub)
4. Create an NVRAM space with 20 bytes, generally readable, but only writeable with the owner authorization at index 0x00011101
5. Use the owner authorization to write a hash of the pub key into the NVRAM space
6. Write a program that has input of 32 random hex bytes
7. Compare the hash value of EnterpriseBackup.pub with the value stored in the NVRAM location: 0x00011101
8. If the comparison matches, encrypt (bind) the 32 hex bytes with the EnterpriseBackup.pub key and store the encrypted data in the file Encrypted.dat. (This will be done on the client computer.)
9. Write another program using the registered key to decrypt the file Encrypted.dat. (This would be done on the server computer.)

Chapter 21

Summary

TrouSerS provides a very easy environment in which to take advantage of the features of the TPM. Its object oriented nature allows a programmer to learn how to use different features incrementally. This paper has shown how to set up that environment, and then has gone through the various objects of the Trusted Computing Group's Software Stack (TSS) which TrouSerS implements and has given working example code which exercises the mostly commonly used features of the TPM. I hope you find it useful.

Bibliography

[1] TSS 1.2 Errata A Specification:
http://www.trustedcomputinggroup.org/resources
/tcg_software_stack_tss_specification

[2] A Practical Guide To Trusted Computing: David Challener, Kent Yoder, Ryan Catherman, David Safford, Leendert Van Doorn