TSS.Net – A TPM2.0 Access Library

*Ronald Aigner, Paul England, Andrey Marochko, Dennis Mattoon, Microsoft Corporation*

# Introduction

TSS.Net is a library for accessing version 2.0 of the Trusted Platform Module (TPM). It supports the TPM reference implementation through a standard TCP/IP socket connection as well as the TPM Base Services (TBS) interface available in Windows. TSS.Net is written in C# and the resulting assembly can be accessed from any .NET language.

The philosophy of TSS.Net is to be a complete but very thin wrapper around the capabilities of the TPM. In most cases TPM functions are accessible through TSS.Net with (essentially) the same input and return parameters that are defined in the TPM specification. Similarly, all data structures defined in the specification have been automatically translated into corresponding .NET types (classes and enumerations).

Here is a simple but complete sample that demonstrates obtaining random numbers from the TPM.

using System;

using Tpm2Lib;

namespace TpmSamples

{

class Program

{

// Simplest program to access the TPM simulator

static void Main()

{

// Connect to the simulator on this machine

ITpm2Device device = new TcpTpmDevice("localhost", 2321);

device.Connect();

Tpm2 tpm = new Tpm2(device);

// If we are running agains the simulator we need to do some of

// the things that the BIOS would normally do

tpm.\_GetUnderlyingDevice().PowerCycle();

tpm.Startup(Su.Clear);

// Ask the TPM for some random data

byte[] rand = tpm.GetRandom(16);

Console.Write("Random data:" + BitConverter.ToString(rand));

// And close the connection to the TPM

tpm.Dispose();

}

}

}

Note: all samples in this document connect to a TPM simulator exporting a standard TCP/IP interface. Examples in the accompanying source distribution also demonstrate access to a real TPM running on Windows.

In some cases TSS.Net provides higher-level abstractions to simplify programming, but in most cases there is a simple mapping between C# methods and the underlying TPM commands. This design is intentional: we hope that learning the library will help the programmer learn the TPM, and *vice-versa*.

(That said we do not consider TSS.Net to be the interface chosen for the majority of application developers: rather, we expect TSS.Net to be used to develop middleware providing higher-level abstractions, and to permit TPM experimentation).

This document is an overview of TSS.Net. It describes how the library can be used to access the TPM and how the library is derived from the TPM specification. All code snippets are copied from the sample projects in the source distribution. With the exception of the complete sample above we omit repeated scaffolding code like establishing the TPM connection (see the sample projects for the working examples themselves).

This document (or more particularly the accompanying samples) also attempts to illustrate the use of the TPM to support common scenarios.

TSS.Net - and indeed the TPM - goes far beyond the examples in this document.

## Naming Conventions

In this section we describe TSS.Net’s type-naming convention. The TPM specification uses c-style names for types. For example, the TPM structure that describes the results of a command audit is called *TPMS\_COMMAND\_AUDIT\_INFO*. TSS.Net uses names more familiar to Java or C# programmers: e.g. the structure above is translated into a .NET class named *CommandAuditInfo*. The translation is mechanical, with a few exceptions to improve readability or avoid name-collisions with frequently used types in the class libraries.

The full set of translation rules is provided elsewhere, but to improve readability the following general principles will be useful:

**Function Names**

TPM function names have TPM2\_ omitted. For example, *TPM2\_GetTestResult* becomes the member function *GetTestResult* in the Tpm2 class.

**Structures**

Structures (*TPMS\_*, *TPMT\_*) have the initial identifier and any contained underscores removed, and then the resulting word collection is camel-cased. For example, *TPMS\_ASYM\_PARMS* becomes the class *AsymParms*.

Similar rules are applied for enumerations and bit fields.

**Helper or Internal Function Names**

Helper or internal function names are prefixed with an underscore. For example *\_GetLastResponseCode*.

# Using the Library

In this section we illustrate the use of TSS.Net (and the underlying TPM) by means of a series of examples. The samples themselves are in the sample projects, and it is assumed that the “tpm” object has been created and connected (as illustrated above).

## Simple Commands

The following sample uses the TPM to calculate the SHA256 hash a 3-byte array. It illustrates the use of enumerated types (*TpmAlgId.Sha256*) and .NET types derived from the specification (*TkHashCheck*, and the prototype for the TPM Hash function itself).

// This sample illustrates simple command execution

// We ask the TPM to calculate the hash of a 3-byte

// array.

TkHashcheck validation;

byte[] hashData = tpm.Hash(

new byte[] {1, 2, 3}, // data to hash

TpmAlgId.Sha256, // hash algorithm

TpmHandle.RhOwner, // hierarchy for ticket (not used here)

out validation); // ticket (not used in this example)

Console.WriteLine("Hashed data:" + BitConverter.ToString(hashData));

Also demonstrated in this sample is the handling of TPM input and output parameters. All commands in the TPM specification start with a boilerplate header. The header contains the command code, the length of the marshaled command string, and a flag indicating whether the command has associated authorization sessions (next section). TPM responses have a similar standard header. The library automatically creates, consumes, and validates these headers.

Other TPM input parameters are directly transliterated from the specification. The first output parameter is translated as the output of the TPM function call (the hash result in this case). Any remaining output parameters must be passed as “out” references).

If the TPM returns an error then the default behavior of TSS.Net is to convert the error into an exception. Ways to override this behavior are described later.

# Handles and Command Authorization

The following example illustrates the use of TPM handles and how commands are authorized.

// Create an auth-value to control later access to the hash object.

// The AuthValue class is a TSS.Net-provided helper object for managing

// authorization data. Here we ask the library to create a 10-byte

// random array

AuthValue authVal = AuthValue.CreateRandom(10);

// Create a SHA-1 hash sequence-object with the auth-value provided.

// This command returns a uint identifier that we call a handle.

TpmHandle hashHandle = tpm.HashSequenceStart(authVal, TpmAlgId.Sha1);

// Hash some data using the hash sequence object just created.

// It is normally the case that the use of TPM internal objects

// must be "authorized" by proof-of-knowledge of the authorization

// value that was set when the object was created.

// Authorization is communicated using a TPM construct called a

// "session." Every handle that requires authorization requires

// a session that conveys knowledge of the auth-value (or other

// authorization).

// in plaintext.

// The simplest type of a session used in the examples below is a

// "password authorization session” (PWAP session), where the password

// is communicated in plaintext. It is suitable when interacting with a

// localTPM device.

// When working with a remote TPM device a more complex HMAC session

// is required to prevent man-in-the-middle style attacks.

// A preferred way to specify sessions in a TPM command is to allow TSS.Net

// to do it. The library will automatically generate a session (or sessions)

// type in accordance with the type of the target TPM device.

tpm.SequenceUpdate(hashHandle, new byte[] { 0, 1 });

// Several styles of explicit session specification are available as well.

// Style 1. Indexing operator.

// In the command below the [authValue] construct is NOT an array element

// accessor. Instead it is a shorthand associating a list of authorization

// sessions with the next TPM command.

// Here that here TSS.Net automatically converts authVal into a PWAP session.

tpm[authVal].SequenceUpdate(hashHandle, new byte[] { 2, 3 });

// Style 2. Standalone method \_SetSessions().

// Effect of a call to \_SetSessions() extends to the next TPM command

// invocation only.

// As in the previous code line TSS.Net automatically converts authVal

// into a PWAP session.

tpm.\_SetSessions(authVal);

tpm.SequenceUpdate(hashHandle, new byte[] { 4, 5 });

// Style 3. Chained method \_SetSessions()

// A call to \_SetSessions() returns "this" so the two lines above can be merged.

tpm.\_SetSessions(authVal).SequenceUpdate(hashHandle, new byte[] { 6, 7 });

// add the final data block

TkHashcheck validation;

byte[] hashedData = tpm[authVal].SequenceComplete(

hashHandle,

new byte[] {4, 5},

TpmHandle.RhOwner,

out validation);

Console.WriteLine("Hashed data: " + BitConverter.ToString(hashedData));

We note the following features:

### Handles

User-created TPM objects (keys, hash-sequence objects, etc.) are assigned a 32-bit unique identifier by the TPM at creation. This identifier is then used for subsequent operations on the key or other entity. This identifier is referred to as a handleand is encapsulated by the TSS.Net *TpmHandle* class*.* Handles are also used to identify other TPM entities like PCRs, NV-slots, and hierarchy identifiers.

This example demonstrates dynamically created TPM objects. A *TpmHandle* is initialized from the TPM handle identifier returned from *HashSequenceStart*, and is then used to reference the hash sequence object created earlier.

TSS.Net does *not* manage the life of the TPM object. In the example above, the TPM object is deleted as a side-effect of the *SequenceComplete* call.

If TSS.Net is used on top of the OS-provided TBS then handle-management is not particularly critical (because the OS resource manager will swap-out TPM entities if internal TPM resources are exhausted). However, if you are programming against a “bare” TPM, like the simulator, then it will have a limited number of slots and the burden of slot-management falls on the developer.

To simplify programming the bare TPM, TSS.Net also includes a built-in resource manager that provides similar capabilities to Windows (this is described later).

### Authorization

Use of most TPM resources (keys, ownership, privacy related actions, etc.) must be authorized. The TPM provides two forms of authorization: (1) proof of knowledge of an “authorization value” that was established when the entity was created or the TPM is initialized, and (2) satisfaction of a security policy identified by a hash (described later).

Authorization value based entity authorization can be further subdivided into the use of plaintext authorization (called password authorization, or PWAP), and HMAC-based proof-of-knowledge of the authorization value.

Authorization to use entities is communicated to the TPM through a construct called a session. There is one session object for each handle in the command that needs authorization. Although other styles of communicating authorization are illustrated in the sample, it is most convenient to convey authorization information to the TSS.Net through the *[ ]* operator, as illustrated in the example above. (Note that *[ ]* is **not** used as an array accessor in this case. The operator has been overloaded). The authorization value is only associated with the TPM object until the next TPM command returns. That is why the authorization value is provided for each TPM command invocation in the example above.

More complex forms of authorization are illustrated in later examples.

## Union Types and Error Handling

The following example demonstrates creation of an RSA primary signing key, using the TPM to sign data, and then using the TSS.Net library and the TPM itself to validate the signature.

The sample also demonstrates some strategies for handling expected (or possible) errors.

// The TPM needs a template that describes the parameters of the key

// or other object to be created. The template below instructs the TPM

// to create a new 2048-bit non-migratable signing key.

TpmPublic keyTemplate = new TpmPublic(

TpmAlgId.Sha1, // name algorithm

ObjectAttr.Sign | // signing key

ObjectAttr.FixedParent | ObjectAttr.FixedTPM | // non-migratable

ObjectAttr.UserWithAuth | ObjectAttr.SensitiveDataOrigin,

new byte[0], // no policy

new RsaParms( // 2048 bit RSA

new SymDefObject(), // with scheme SSA

new SchemeRsassa(TpmAlgId.Sha1),

2048, 0),

new Tpm2bPublicKeyRsa());

// Authorization for the key we are about to create.

byte[] keyAuth = new byte[] {1, 2, 3};

TpmPublic keyPublic;

CreationData creationData;

TkCreation creationTicket;

byte[] creationHash;

// Ask the TPM to create a new primary RSA signing key.

TpmHandle keyHandle = tpm[OwnerAuth].CreatePrimary(

TpmHandle.RhOwner, // in the owner-hierarchy

new SensitiveCreate(keyAuth, new byte[0]), // with this aurh-value

keyTemplate, // describes key

new byte[0], // for creation ticket

new PcrSelection[0], // for creation ticket

out keyPublic, // out pubKey and attrs

out creationData, out creationHash, // not used here

out creationTicket);

// Print out text-versions of the public key just created.

Console.WriteLine("New public key\n" + keyPublic.ToString());

// Use the key to sign some data.

TpmHash dataToSign = TpmHash.FromHashOfString(TpmAlgId.Sha1, "ABC");

ISignatureUnion sig = tpm[keyAuth].Sign(

keyHandle, // handle of signing key

dataToSign.HashData, // data to sign

new NullSigScheme(), // use key’s scheme

TpmHashCheck.NullHashCheck());

// Print the signature. A different structure is returned for each

// signing scheme, so cast the interface to our signature type.

SignatureRsassa actualSig = (SignatureRsassa)sig;

Console.WriteLine("Signature: " + BitConverter.ToString(actualSig.sig));

// Use the TPM library to validate the signature.

bool sigOk = keyPublic.VerifySignatureOverHash(dataToSign, sig);

Debug.Assert(sigOk);

// Load the public key into another slot in the TPM and then

// use the TPM to validate the signature.

TpmHandle pubHandle = tpm.LoadExternal(null, keyPublic, TpmHandle.RhOwner);

tpm.VerifySignature(pubHandle, dataToSign.HashData, sig);

// The default behavior of TSS.Net is to create an exception if the   
 // signature does not validate. If an error is expected the library can

// be notified of this, or the exception can be turned into a value that

// can be later queried. The following are examples of this.

actualSig.sig[0] ^= (byte)1;

tpm.\_ExpectError(TpmRc.Signature).VerifySignature(pubHandle,

dataToSign.HashData, sig);

tpm.\_AllowErrors().VerifySignature(pubHandle, dataToSign.HashData, sig);

Debug.Assert(tpm.\_GetLastResponseCode() == TpmRc.Signature);

// Clean up

tpm.FlushContext(keyHandle);

tpm.FlushContext(pubHandle);

// Demonstrate the use of XML persistence by saving keyPublic to

// a file and making a copy by reading it back into a new object.

string fileName = "sample.xml";

string xmlVersionOfObject = keyPublic.GetXml();

keyPublic.XmlSerializeToFile(fileName);

TpmPublic copyOfPublic =

TpmStructureBase.XmlDeserializeFromFile<TpmPublic>(fileName);

// Demonstrate TSS.Net support of TPM-structure equality operators.

if (copyOfPublic != keyPublic)

Console.WriteLine("Library bug persisting data.");

File.Delete(fileName);

This sample also illustrates strategies for handling errors. The default behavior of TSS.Net is to translate a TPM error into a *TpmException* containing information about the error returned from the TPM (for example, the error type, and possibly the parameter that was discovered to be in error).

Methods of the *Tpm2* class that do not result in a command being issued to the underlying TPM are conventionally prepended with an underscore. E.g., these methods are used in this example:

|  |  |
| --- | --- |
| *Tpm.\_ExpectError(TpmRc expectedErr)* | Tells TSS.Net to create an exception if the TPM does not return the error value indicated |
| *Tpm.\_AllowErrors()* | A Tpm2 member variable is set to *TpmRc.Success* or any error returned by the TPM |
| *Tpm.\_GetLastResponseCode()* | Returns the last response code |

Note that these methods set an attribute or variable in the *Tpm2* instance, and this attribute is only cleared when a TPM command is executed.

This sample also illustrates how the library exposes TPM union types. The TPM typically uses union types to support algorithm agility. For example, the *TPMU\_PUBLIC\_PARMS* union describes the parameters of any supported algorithm. This union has members that describe the specific algorithms supported in the TPM: for example *TPMS\_ECC\_PARMS* for ECC, and *TPMS\_RSA\_PARMS* for RSA. Unions are always used in an enveloping structure with another member that is the “selector” for the union. TSS.Net defines an interface for each defined union type. In this case it is called *IPublicParmsUnion*. Every structure that is included as part of the union in the specification implements this interface (and must implement the member function *GetUnionSelector()*). Practically, if a union interface is required in a structure or command, you may use any class that implements the union. If an interface union is returned from the TPM you may cast it to the underlying class (perhaps using “*is*” or *GetUnionSelector()* to determine the proper cast).

Note also the use of *ToString()* to render a human-readable version of any TPM data structure. For instance the public key printed above

Tpm2Lib.TpmPublic

TpmAlgId type = Rsa

TpmAlgId nameAlg = Sha1

ObjectAttr objectAttributes = FixedTPM| FixedParent| SensitiveDataOrigin|

UserWithAuth| Sign  
 ushort authPolicySize = 0x0 (0)

byte authPolicy[0] = 0x

RsaParms parameters (IPublicParmsUnion)

SymDefObject symmetric

TpmAlgId algorithm = Null

ushort keyBits = 0x0 (0)

TpmAlgId mode = Error

TpmAlgId schemeScheme = Rsassa

SchemeRsassa scheme (IAsymSchemeUnion)

TpmAlgId hashAlg = Sha1

ushort keyBits = 0x800 (2048)

uint exponent = 0x0 (0)

Tpm2bPublicKeyRsa unique (IPublicIdUnion)

ushort size = 0x100 (256)

byte buffer[256] = 0xb8518adb 12750b50 ... 1a73a6fd c8fb182d

Finally in this example we demonstrate TSS.Net support for XML and persistence. The method *GetXml()* returns an XML representation of the object. The methods *XmlSerializeToFile()* and *XmlDeserializeFromFile()* allow any TPM structure to be saved and restored in a machine and TPM-independent form.

## HMAC Sessions

This sample illustrates how TSS.Net may be used to provide an HMAC session for authorization. As the comments in the code describe, many options are supported by the TPM and the library, but this sample demonstrates the unbound and unseeded form.

In this sample an *AuthSession* object (encapsulating an HMAC session) is used to convey the authorization session for the *SequenceUpdate* command.

// This sample shows the use of HMAC sessions to authorize TPM actions.

// HMAC sessions may be bound/unbound and seeded/unseeded. This sample

// illustrates an unseed and unbound session.

// Create a hash-sequence with a random authorization value.

AuthValue authVal = AuthValue.RandomValue(8);

TpmHandle hashHandle = tpm.HashSequenceStart(authVal, TpmAlgId.Sha256);

// Commands with the Ex modifier are library-provided wrappers

// around TPM functions to make programming easier. This version

// of StartAuthSessionEx calls StartAuthSession configured to

// create an unbound and unseeded auth session.

// The auth-value will be automatically added by TSS.Net when the session

// will be used to authorize a particular handle.

AuthSession s0 = tpm.StartAuthSessionEx(TpmSe.Hmac, TpmAlgId.Sha256);

TkHashcheck validate;

// The following calls show the use of the HMAC session in authorization.

// The session to use is communicated as a parameter in the [] overloaded

// function and the auth-value is that set during HMAC session creation.

tpm[s0].SequenceUpdate(hashHandle, new byte[] { 0, 2, 1 });

byte[] hashedData = tpm[s0].SequenceComplete(hashHandle,

new byte[] {2, 3, 4},

TpmHandle.RhOwner, out validate);

tpm.FlushContext(s0.Handle);

This sample also contains an example of a TPM command that is derived from, but not exactly identical to an underlying TPM command. The TPM command *StartAuthSession* contains many parameters that are absent (for common use cases). The command *StartAuthSessionEx* is a simple wrapper around *StartAuthSession* that is easier to use. TSS.Net contains a few more such functions.

## A Simple Policy Sessions

A Policy Session is a TPM construct that provides sophisticated logic for command authorization. The TPM defines about a dozen elementary authorizations – for instance PCR-matching, or proof-of-knowledge of a secret through signing a nonce. These elementary authorizations can be combined with AND & OR operations. TSS.Net provides support for an interoperable scheme for creating, using, and sharing these policies.

The first example illustrates the creation of a policy for a sealed object that can only be accessed if certain PCR are set correctly AND the command is issued at locality zero.

// This sample illustrates the use of a simple TPM policy

// session. The policy demands PCR 1, 2, 3 set to current

// values, and the command be issues at locality zero

// First read the PCR values.

uint[] pcrs = new uint[] { 1, 2, 3};

PcrSelection sel = new PcrSelection(TpmAlgId.Sha, pcrs);

PcrSelection[] selOut;

Tpm2bDigest[] pcrValues;

uint pcrGeneration = tpm.PCR\_Read(new PcrSelection[] { sel },

out selOut, out pcrValues);

// Save the current PCR values in a convenient data structure.

PcrValueCollection expectedPcrVals =

new PcrValueCollection(selOut, pcrValues);

// TSS.Net encapsulates policy sessions with the TpmPolicySession class.

TpmPolicySession policySession = new TpmPolicySession(TpmAlgId.Sha256);

// Set the policy: Locality AND PolicyPcr.

policySession.SetPolicyRoot(

new TpmPolicyLocality(LocalityAttr.TpmLocZero)).And(

new TpmPolicyPcr(expectedPcrVals));

// Ask TSS.Net for the expected policy-hash for this policy.

TpmHash expectedPolicyHash = policySession.ComputePolicyHash();

// Create a sealed primary object with the policy-hash we just calculated.

byte[] dataToSeal = new byte[] {1, 2, 3, 4, 5, 4, 3, 2, 1};

TpmHandle primHandle = CreateSealedPrimaryObject(dataToSeal, null,

expectedPolicyHash.HashData);

// Create an actual TPM policy session to evaluate the policy.

AuthSession sess = tpm.StartAuthSessionEx(TpmSe.Policy, TpmAlgId.Sha256);

// Run the policy on the TPM.

policySession.RunPolicy(tpm, sess, "leaf");

// And unseal the object.

byte[] unsealedData = tpm[sess].Unseal(primHandle);

Console.WriteLine("Unsealed data: " + BitConverter.ToString(unsealedData));

// Change a PCR and make sure that the policy no longer works.

tpm[NullAuth].PCR\_Event(TpmHandle.Pcr(3), new byte[] { 1, 2, 3 });

tpm.PolicyRestart(sess.Handle);

// Run the policy again - an error will be returned.

TpmRc policyError = policySession.RunPolicy(tpm, sess, null, true);

// And the session will be unusable.

unsealedData = tpm[sess].\_ExpectError(TpmRc.PolicyFail).Unseal(primHandle);

// Clean up.

tpm.FlushContext(sess.Handle);

tpm.FlushContext(primHandle);

## A Policy Session with OR-Branches

This example extends the previous example to the policy above OR proof-of-knowledge of a password using two policy branches and a *TpmPolicyOr*.

(Some of the test code is omitted here since it is identical to the sample above)

[[ detail omitted – see the sample ]]

// This sample illustrates the use of PolicyOr

TpmPolicyOr orAce = (TpmPolicyOr)

policySession.SetPolicyRoot(new TpmPolicyOr());

// Set the two branches

orAce.AddPolicyBranch(

new TpmPolicyLocality(LocalityAttr.TpmLocZero)).And(

new TpmPolicyPcr(expectedPcrVals, "branch\_1"));

orAce.AddPolicyBranch(new TpmPolicyAuthValue("branch\_2"));

// Ask TSS.Net for the expected policy-hash for this policy

TpmHash expectedPolicyHash = policySession.ComputePolicyHash();

// Create a sealed primary object with the policy-hash we just calculated

byte[] dataToSeal = new byte[] { 1, 2, 3, 4, 5, 4, 3, 2, 1 };

byte[] authVal = new byte[] {1, 2};

TpmHandle primHandle = CreateSealedPrimaryObject(dataToSeal, authVal,

expectedPolicyHash.HashData);

// Create an actual TPM policy session to evaluate the policy

AuthSession sess = tpm.StartAuthSessionEx(TpmSe.Policy, TpmAlgId.Sha256);

// Run the policy on the TPM

policySession.RunPolicy(tpm, sess, "branch\_1");

// And unseal the object

byte[] unsealedData = tpm[sess].Unseal(primHandle);

Console.WriteLine("Unsealed data: " + BitConverter.ToString(unsealedData));

// Now use the other branch

tpm.PolicyRestart(sess.Handle);

sess.SetAuthVal(authVal);

policySession.RunPolicy(tpm, sess, "branch\_2");

unsealedData = tpm[sess].Unseal(primHandle);

Console.WriteLine("Unsealed data: " + BitConverter.ToString(unsealedData));

[[ clean-up omitted ]]

## Policy Serialization and Policies with Callbacks

The sample Policy (*SamplePolicySerializationAndCallbacks*) demonstrates how policies can be persisted to XML in a standard form (this is described in more detail elsewhere).

Additionally, certain policies need the calling program or user to prove knowledge of a password or private key (obviously such secrets should not be persisted in the XML policy). TSS.Net provides callbacks to allow the calling program to provide necessary authorization (e.g. prompting the user to insert a smart card or type in a PIN).

The application program can install callbacks for:

1. Signing with a private key (*TpmPolicySigned*)
2. Managing NV-slots (*TpmPolicyNV*)
3. Providing session-based authentication (*TpmPolicySecret*), and
4. A dummy *TpmPolicyAction* callback which does not execute on the TPM but calls back into the main program to perform actions out of the scope of the library (for instance incrementing a monotonic counter).

We do not include the sample in this document because it is quite long.

## Using the Built-In Resource Manager

TSS.Net has a built-in resource manager called *Tbs* (not to be confused with the *TbsDevice* used to communicate with the Windows TBS API). This is not designed to replace a production-grade resource-manager such as that built into the Windows TPM driver, but makes it easier to write TPM applications that can then be easily ported to Windows.

The example below creates and uses 32 authorization sessions. A typical TPM only has 3 slots so this program running against a bare TPM would fail. In contrast the *Tbs* resource management, which behaves in most respects like the raw TPM device already discussed, provides a virtually unlimited number of key slots.

*Tbs* also automatically cleans up TPM resources when the object is disposed.

// This sample illustrates the use of the resource manager built into

// TSS.Net using the resource manager relieves the programmer of the

// (sometimes burdensome) chore of juggling a small number of TPM slots.

// Tbs is the built -in resource manager. You attach an instance to the

// underlying TPM device, which has been created before.

Tbs tbs = new Tbs(tpm.\_GetUnderlyingDevice(), false);

Tpm2 tbsTpm = new Tpm2(tbs.CreateTbsContext());

// Make more sessions than the TPM has room for.

int count = 32;

AuthSession[] sessions = new AuthSession[count];

for (int j = 0; j < count; j++)

{

sessions[j] = tbsTpm.StartAuthSessionEx(TpmSe.Policy, TpmAlgId.Sha1);

}

// And now use them. The resource manager will use ContextLoad and

// ContextSave to bring them into the TPM.

for (int j = 0; j < count; j++)

{

tbsTpm.PolicyAuthValue(sessions[j].Handle);

}

// And now clean up.

for (int j = 0; j < count; j++)

{

tbsTpm.FlushContext(sessions[j].Handle);

}

tbsTpm.Dispose();

## Other features

TSS.Net was built primarily as a library to support the testing we did during the development and stabilization of TPM 2.0. As such it has features to support all TPM commands and scenarios, as well as algorithms support for expected cipher-suites (e.g. RSA, ECC, SHA up to SHA512, AES, etc.).

TSS.Net also supports some test-specific features, like various types of pre- and post-command callbacks. This document does not attempt to describe all TSS.Net capabilities.

# Appendix A – Other Use Samples

The samples in this appendix do not introduce any new TSS.Net features, but illustrate how the TPM can be used to perform various functions.

## Creating a Primary Storage Key

This subroutine illustrates the creation of a primary storage key and returning the key handle to the caller. It is used in later samples.

The storage key created is a 2048-bit RSA paired with a 128 bit AES/CFB key. The caller can provide use-authorization data, but this routine sets the policy to null. The storage key is not duplicable.

TpmHandle CreateRsaPrimaryStorageKey(

Tpm2 theTpm,

byte[] auth,

out TpmPublic newKeyPub)

{

// Creation parameters (no external data for TPM-created objects)

SensitiveCreate sensCreate = new SensitiveCreate(

auth, // auth-data provided by the caller

new byte[0]); // no external data (the TPM will create the

// new key.

TpmPublic parms = new TpmPublic(

TpmAlgId.Sha256,

ObjectAttr.Restricted | ObjectAttr.Decrypt // storage key

| ObjectAttr.FixedParent | ObjectAttr.FixedTPM // non-duplicatable

| ObjectAttr.UserWithAuth | // requires user-auth

ObjectAttr.SensitiveDataOrigin, // TPM should create

new byte[0], // no policy

new RsaParms( // storage key should

// be RSA + AES128

new SymDefObject(TpmAlgId.Aes, 128, TpmAlgId.Cfb),

new NullAsymScheme(),

2048,

0),

new Tpm2bPublicKeyRsa());

// The following are returned by the TPM in CreatePrimary (and Create)

// they are not used in this sample.

CreationData creationData;

TkCreation creationTicket;

byte[] creationHash;

TpmHandle primHandle = tpm[OwnerAuth].CreatePrimary(

TpmHandle.RhOwner, // in storage hierarchy

sensCreate, // userAuth

parms, // creation parms set above

// the following parameters influence the creation of the

// creation-ticket. They are not used in this sample

new byte[0], // null outsideInfo

new PcrSelection[0], // do not record PCR-state

out newKeyPub,

out creationData, out creationHash, out creationTicket);

return primHandle;

}

## Creating and Using a Restricted Signing Key

This sample illustrates the creation and use of a signing key that is a child of a storage key (such as that created in the previous example). The signing key in this case is “restricted.” Restricted signing keys are typically used to “quote” the properties of other loaded keys or other TPM state. In this sample we show how to instruct the TPM to *Quote* the values of the SHA1 PCR 1, 2 and 3.

All data structures that are quoted by the TPM start with the value *TPM\_GENERATED*. Restricted signing keys can also be used to sign external data, but only if the TPM can ascertain that the data does not start with this value. This sample demonstrates the failure that will occur if the TPM is asked to sign a piece of data that it did not hash (and therefore cannot be sure that it does not start with *TPM\_GENERATED*). The sample also demonstrates how to use a hash sequence to have the TPM hash the data and generate a ticket that can be used to tell the TPM later that the data is indeed safe.

        internal void Sample9()

        {

            // This sample illustrates the creation and use of an RSA signing key to

            // "quote" PCR state

            // First use a library routine to create an RSA/AES primary storage key

            // with null user-auth.

            TpmPublic rsaPrimaryPublic;

            byte[] primaryAuth = new byte[0];

            TpmHandle primHandle = CreateRsaPrimaryStorageKey(tpm, primaryAuth,

                out rsaPrimaryPublic);

            // template for a signing key.  We will make the key restricted so that we

            // can quote with it too.

            TpmPublic signKeyPubTemplate = new TpmPublic(

                TpmAlgId.Sha1,

                ObjectAttr.Sign | ObjectAttr.Restricted|// a "quoting" key

                ObjectAttr.FixedParent |                // non-duplicatable

                ObjectAttr.UserWithAuth |               // authorize with auth-data

                ObjectAttr.SensitiveDataOrigin,         // TPM will create a new key

                new byte[0],

                new RsaParms(                           // key type and sig scheme

                    new SymDefObject(),

                    new SchemeRsassa(TpmAlgId.Sha1), 2048, 0),

               new Tpm2bPublicKeyRsa());

            // auth-data for new key

            byte[] userAuth = new byte[] { 1, 2, 3, 4 };

            SensitiveCreate sensCreate = new SensitiveCreate(

                userAuth,                   // userAuth

                new byte[0]);               // data

            // Creation data (not used in this sample)

            CreationData childCreationData;

            TkCreation creationTicket;

            byte[] creationHash;

            // Create the key

            TpmPublic keyPub;

            TpmPrivate keyPriv = tpm[primaryAuth].Create(

                primHandle,                     // child of primary key created above

                sensCreate,                     // auth-data

                signKeyPubTemplate,             // template created above

                new byte[0],                    // other parms are not used here

                new PcrSelection[0] ,

                out keyPub,

                out childCreationData, out creationHash, out creationTicket);

            // Load the key as a child of the primary that it

            // was created under.

            TpmHandle signHandle = tpm[primaryAuth].Load(primHandle, keyPriv, keyPub);

            // Note that Load returns the "name" of the key and this is automatically

            // associated with the handle.

            Console.WriteLine("Name of key:" +

                BitConverter.ToString(signHandle.Name));

            // some data to quote

            TpmHash hashToSign = TpmHash.FromHashOfData(TpmAlgId.Sha1,

                new byte[] {4, 3, 2, 1});

            // PCRs to quote.  SHA-1 bank, pcr-indices 1, 2, and 3

            PcrSelection[] pcrsToQuote = new PcrSelection[]

            {

                new PcrSelection(TpmAlgId.Sha, new uint[] { 1, 2, 3 })

            };

            // Ask the TPM to quote the PCR (and the nonce).  The TPM

            // returns the quote-signature and the data that was signed

            ISignatureUnion quoteSig;

            Attest quotedInfo = tpm[userAuth].Quote(

                signHandle,

                hashToSign.HashData,

                new SchemeRsassa(TpmAlgId.Sha1),

                pcrsToQuote,

                out quoteSig);

            // print out what was quoted

            QuoteInfo info = (QuoteInfo) quotedInfo.attested;

            Console.WriteLine("PCR that were quoted: " +

                info.pcrSelect[0].ToString() +

                "\nHash of PCR-array: " +

                BitConverter.ToString(info.pcrDigest));

            // read the PCR to check the quoted value

            PcrSelection[] outSelection;

            Tpm2bDigest[] outValues;

            tpm.PCR\_Read(

                new PcrSelection[]  // PCR to read

                {

                    new PcrSelection(TpmAlgId.Sha, new uint[] { 1, 2, 3 })

                }

                , out outSelection, out outValues);

            // use the TSS.Net library to validate the quote against the

            // values just read.

            bool quoteOk = keyPub.VerifyQuote(

                TpmAlgId.Sha1,

                outSelection,

                outValues,

                hashToSign.HashData,

                quotedInfo,

                quoteSig);

            if (!quoteOk) { Console.WriteLine("Quote did not validate"); }

            // Test other uses of the signing key.  A restricted key can only

            // sign data that the TPM knows does not start with a magic

            // number (that identifies TPM internal data).  So this does not

            // work

            TkHashcheck nullProof = new TkHashcheck(TpmHandle.RhNull,

                new byte[0]);

            tpm[userAuth].\_ExpectError(TpmRc.Ticket).Sign(

                signHandle,

                hashToSign.HashData,

                new SchemeRsassa(TpmAlgId.Sha1),

                nullProof);

            // but if we as the TPM to hash the same data and then sign it

            // then the TPM can be sure that the data is safe, so it will

            // sign it.

            TkHashcheck safeHashTicket;

            TpmHandle hashHandle = tpm.HashSequenceStart(

                NullAuth,

                TpmAlgId.Sha1);

            // the ticket is only generated if the data is "safe."

            tpm[NullAuth].SequenceComplete(hashHandle,

                new byte[] { 4, 3, 2, 1 },

                TpmHandle.RhOwner,

                out safeHashTicket);

            // this will now work because we the ticket proves to the

            // TPM that the data it is about to sign does not start with

            // TPM\_GENERATED

            ISignatureUnion sig = tpm[userAuth].Sign(

                signHandle,

                hashToSign.HashData,

                new SchemeRsassa(TpmAlgId.Sha1),

                safeHashTicket);

            // and we can verify the signature

            bool sigOk = keyPub.VerifySignatureOverData(

                new byte[] { 4, 3, 2, 1 }, sig);

            if (!sigOk) Console.WriteLine("Signature did not verify");

            tpm.FlushContext(primHandle);

            tpm.FlushContext(signHandle);

# Appendix B – Using TSS.Net With Mono

The Mono project is an open source .NET implementation that allows developers to create cross platform applications using C# and the Common Language Runtime. From the Mono Project:

**Mono** is a software platform designed to allow developers to easily create cross platform applications. Sponsored by [Xamarin](http://www.xamarin.com/), Mono is an open source implementation of Microsoft's .NET Framework based on the [ECMA](http://www.mono-project.com/ECMA) standards for [C#](http://www.mono-project.com/CSharp_Compiler) and the [Common Language Runtime](http://www.mono-project.com/Mono:Runtime). A growing family of solutions and an active and enthusiastic contributing community is helping position Mono to become the leading choice for development of Linux applications.

Linux is not the only platform for which Mono support exists. Both Windows and Mac OS X, as well as mobile platforms, are also supported. TSS.Net, and its associated sample code, can be built and run under Mono. With the following caveats.

## TSS.Net Testing and Mono Versions

The TSS.Net library and sample code have only been tested against Mono C# compiler (gmcs) version 3.0.6.0 and version 3.0.3.2 of the MonoDevelop development environment. Also note, the process of accessing and installing Mono-project binaries may vary depending on your particular platform.

As of this writing (using the previously-stated version of Mono), there are a few issues of which users need to be aware. First is a bug in MonoDevelop. In order to open TSS.Net solution files using the MonoDevelop IDE, the user must edit the .sln and change the Visual Studio File Format Version. To open a solution in MonoDevelop, change this version number to 11.00. As in the following example (from the GetCapabilities sample):

Microsoft Visual Studio Solution File, Format Version 11.00  
# Visual Studio 2012  
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "GetCapabilities", "GetCapabilities.csproj", "{A704A8E  
EndProject  
Global  
 GlobalSection(SolutionConfigurationPlatforms) = preSolution  
 Debug|Any CPU = Debug|Any CPU  
 Release|Any CPU = Release|Any CPU  
 EndGlobalSection  
 GlobalSection(ProjectConfigurationPlatforms) = postSolution  
 {A704A8EE-003D-4B88-8AEF-4B0AC9A3D237}.Debug|Any CPU.ActiveCfg = Debug|Any CPU  
 {A704A8EE-003D-4B88-8AEF-4B0AC9A3D237}.Debug|Any CPU.Build.0 = Debug|Any CPU  
 {A704A8EE-003D-4B88-8AEF-4B0AC9A3D237}.Release|Any CPU.ActiveCfg = Release|Any CPU  
 {A704A8EE-003D-4B88-8AEF-4B0AC9A3D237}.Release|Any CPU.Build.0 = Release|Any CPU  
 EndGlobalSection  
 GlobalSection(SolutionProperties) = preSolution  
 HideSolutionNode = FALSE  
 EndGlobalSection  
EndGlobal

## Mono and ECC

Second, Mono does not currently support ECC. Specifically, the Mono runtime does not implement *System.Security.Cryptography.ECDsaCng* or *System.Security.Cryptography.ECDiffieHellmanCng*. A quick search on the identifier ‘\_\_MonoCS\_\_’ in the TSS.Net sources will provide the user a summary of the effected code paths.

## Other Issues

Third, there is a related issue with RSA signing. The Authorization sample provides a good example of this issue. Mono reports an exception indicating that SHA1 is an unsupported hash algorithm for RSA signing. In our samples, this exception has only been reported on calls to the VerifyHash method of the RSA provider (and similar). Since it is only on a verify call, and Mono support is not currently a priority for TSS.Net, the issue has not been investigated or addressed in any way.

Finally, support for XML serialization in Mono seems to be lacking compared to the native Microsoft .Net CLR. Similar to the RSA signature verification issue, XML serialization support for Mono has not been addressed and, at this time, there are no plans to do so in the future.

As always, we welcome feedback from the community on all aspects of TSS.Net. If Mono support is a priority for you and you’d like to provide feedback on the above mentioned issues, we’d be happy to hear it.