# TPM 2.0 Cloudproxy prototype and protocol

manferdelli@, tmroeder@

## Introduction

This document describes Tpm 2.0  and a series of C++ demonstration programs that show how Tpm 2.0 is used to support Cloudproxy.  For a description of loudproxy, see http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.

Cloudproxy requires a hardware root of trust to measure booted images, store secrets for those images and "attest" or prove the validity and binding of keys to program identity and  associated protection properties for system software.  These capabilities were originally provided in Cloudproxy by Tpm 1.2.  The wide-scale availability of Tpm 2.0 on newer machines along with Intel's deployment of a "soft tpm 2.0" on Haswell chipsets makes it a "must support" for Cloudproxy.

Although Tpm 2.0 provides similar base functionality as Tpm 1.2 provides, and, in the case of Cloudproxy, ultimately identical capabilities, the interface is significantly different.  Differences include:

- Support for additional modern algorithms like ECC and SHA-2.
- A more sophisticated authorization model for features and functions.
- Unified key migration between Tpm's.
- A simplified and more powerful mechanism for attestation and "AIK" certification.

The machine level I/O interface for Tpm 1.2 and Tpm 2.0 is similar: Tpm 2.0 is a character device supporting reads and writes from /dev/tpm0 but the commands are quite different and incompatible.  As with Tpm1.2, our interface uses no external code or libraries and implements commands directly by writing to /dev/tpm0 using our tpm2_lib library.  In Cloudproxy, the base system, either the OS or a hypervisor, "owns" the tpm device.  The Cloudproxy interface virtualizes interactions with the Tpm for all software in the stack.  Note that in Linux, the required Tpm 2.0 driver is in version 4 kernels and later.

The four voume TPM 2.0 specification is available at http://www.trustedcomputinggroup.org/resources/tpm_library_specification.  It is voluminous.  A novel feature of the Tpm 2.0 specification is that the formal behavioral model appears in the form of an executable "C" like language and all required ".h" files can be obtained directly from the specification.  In fact, this specification code can be used to build a Tpm simulator.  The TCG group has such a simulator and makes it available to TCG members; the simulator runs only on Windows but is accessible over a network using a TLS based protocol.

A book entitled "A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security," by Will Arthur, David Challener is also available, although I did not find it that useful.  Trousers does not support Tpm 2.0 but there is open source code, from Intel, at https://github.com/01org/TPM2.0-TSS.  Microsoft also has some nice C++ interface code, written in part by Paul England, at https://tpm2lib.codeplex.com/ but it is currently available under a restricted license.  I am told Microsoft plans to release this code under a UCB license in the future.  One useful piece of documentation that was available for the Tpm 1.2 was a code primer in how to use the Tpm (using the /dev/tpm0 character interface).  This was available in the form of some powerpoint slides presented by David Challener at CMU.  That presentation also has a simplified "theory of operation." I have not found a correspondingly simple description for TPM 2.0.

Our code for TPM 2.0 is available in the Cloudproxy repository ( http://github.com/jlmucb/cloudproxy ).  The prototype code is in the directory cloudproxy/src/tpm2.

The prototype code runs under Linux *using a version 4 kernel or lat*er.  Instructions for enabling and configuring the TPM are also in the Cloudproxy/src/tpm2 directory.

## Overall Description of Tpm 2.0

Much of Tpm 2.0's architecture is similar to tpm1.2.  Programmatic interfaces are specified in tpm12.h and tpm20.h in the cloudproxy/src/tpm2 directory.

As with Tpm 1.2, all data passed to and retrieved from the Tpm is big endian format and, indeed, much of the interface code involves marshalling between tpm formats and native formats.   Each Tpm object (keys, context, nvram) has an associated handle (a 32 bit identifier).

Commands to the TPM all have the same prefix format naming the authentication method, command-id and size of the command buffer; after the prefix, all commands have per command formats although there are common authentication data formats for commands that require authentication.  Similarly, responses have a common prefix with authentication information, error code and size of response followed by command specific output.  Once an object has an associated open handle, it can be closed using Tpm2_FlushContext on the handle.  The per command data transmitted  to and returned by the Tpm are in specific marshallable structures (like, TPM2B_DIGEST, TPM2B_NAME, …) described in tpm20.h.

Tpm 2.0 has several standard owner handles for platform owner,  admin owner and endorsement owner.  Unlike Tpm 1.2, initialization is done entirely by firmware when the Tpm is enabled.  At that time, all the owner authorization is cleared so all access is granted pending changes.  Tpm2_Startup and Tpm2_Shutdown are handled entirely by firmware.  You cannot "reinit" the Tpm from  the OS, the moral equivalent of clearing the Tpm must be done via the firmware interface.

Critical to TPM 2.0 is the key hierarchy. The “endorsement key” is a storage root key. As with other storage roots, this is created by Tpm2_CreatePrimary with the designated endorsement key handle (TPM_RH_ENDORSEMENT). Unlike other keys the endorsement key is always generated from a (per Tpm but permanent) seed so every time you create the Endorsement Key you get the same values. There are two other kinds of keys under an Endorsement Key parent: signing keys and sealing keys. Quote keys are a type of signing key. These are created with Tpm2_CreateKey and have different values every time they are created. With the exception of the Endorsement Key, keys need to be loaded (using Tpm2_Load) before they can be used. The mechanism for authenticating a quote key for the Tpm (the “AIK” in Tpm 1.2 vernacular) based on the EK has changed. See the discussion below.

Keys, like other TpmPM objects, have authorization information associated with them. With the exception of a sealed object (which requires an authorization session), we always use password protection and currently, the passwords are hard coded in the tests.

Keys can be saved using `Tpm2_SaveContext` and restored using `Tpm_LoadContext`. It is important to close unused handles because the number of allowable open handles is very limited on Tpm 2.0.

There are quite a few new Tpm 2.0 commands but we only need a few others:

> `Tpm2_Unseal` is used to unseal data objects sealed with CreateKey.
>
> `Tpm2_Quote` is used to “quote” data. A quote signs a statement naming some data to be quoted as well as the value of specified Program Configuration Registers.
>
> `Tpm2_DefineNVSpace` defines a name space for NvRam values.
>
> `Tpm2_UndefineNVSpace` clears a name space for NvRam values.
>
> `Tpm2_ReadNvRam` reads an NV register under authorization control.
>
> `Tpm2_WriteNvRam` writes an NV register under authorization control.
>
> `Tpm2_EvictControl` permanently allocates a TPM handle. Unless you call EvictControl, a handle does not survive reboot.
>
> `Tpm2_GetRandom` gets random bits.
>
> `Tpm2_GetCapabilities` retrieves Tpm capabilities and handles. We currently use this to Flushall handles in tpm2_util.

Tpm2_MakeCredential and Tpm2_ActivateCredential are used in the Tpm 2.0 protocol that replaces CertifyAIK in Tpm 1.2. MakeCredential, takes a secret (credential) and encrypts it to a key (typically, the endorsement key), it then associates the protected secret with a non-exportable TPM object; the non-exportable object is identified with a non-spoofable name that is included in the MakeCredential object. MakeCredential unseals the object and if the named associated object is loaded, returns the value. We use this to certify a Program key in the following way: We Make a credential using the Endorsement key that is associated with a non-exportable Quote key. The quote key signs the named public key along with the attendant PCRs. KeyNegoServer verifies the quoted values and using data provided with a request naming the Quote key, quote endorsement key, does an offline MakeCredential on a random 16 byte value (the "protected credential"). It then signs the program key and encrypts the certificate under the protected credential value. It then returns all this to the requester. ActivateCredential (which will only work on the original machine with the specific non-exportable Quote object that quoted the program key request) unseals the credential and the unsealed credential is used to decrypt the signed Program key.

Note that Tpm2_MakeCredential[1] is not actually called in the Cloudproxy protocol described below, KeyNegoServer can carry out the MakeCredential procedure given the Endorsement Key, QuotePublic key and key name, and the quoted values. Of course, before signing the Program key, KeyNegoServer verifies the PCRs correspond to a known, trusted program. The PCR's which should be named are not generally dictated by the spec. As with Tpm 1.2, conventionally, the PCR's KeyNegoServer should check are PCR 17 and 18 and the BIOS PCR's.

tmp2_lib implements several other Tpm 2.0 commands but they are not currently required for Cloudproxy.

## Services provided by tpm2_lib and tpm2_util

All the Tpm supporting code mentioned in the previous section is implemented in tpm2_lib.cc along with some test and additional interface code set forth in Appendix 2. The library contains code that interprets and creates the quote structures as well as the MakeCredential structures so, KeyNegoserver can prepare the values required for ActivateCredential and verify PCRs from a Cloudproxy application requesting a Program Key signature.

tpm2_util.cc implements some additional useful utility functions like GetRandom and is used to call several end-to-end tests involving a specific sequence of Tpm2 calls. These tests include:

    bool Tpm2_SealCombinedTest (..., int pcr_num) which tests a seal-unseal sequence.

[1] First aid: The padding method that should be used by an endorsement key to encrypt the credential is not terribly obvious but is in the sample code in tpm2_lib.

`bool Tpm2_QuoteCombinedTest(..., int pcr_num)` which implements a quote and verify quote sequence.
`bool Tpm2_KeyCombinedTest(..., int pcr_num)` which implements a sign-verify sequence.
`bool Tpm2_NvCombinedTest(...)` which test the NvRam functions.
`bool Tpm2_ContextCombinedTest(...)` which tests SaveContext and LoadContext.
`bool Tpm2_EndorsementCombinedTest(...)` which tests Endorsement Key generation. Note that the generated EK is always the same value. As with Tpm 1.2, vendors don't usually supply an Endorsement Cert (OEM's charge a fee for this for enterprise customers). As with Tpm 1.2, the utilities below allow a cloud provider to sign an endorsement certificate and we continue to believe this will be the primary method of obtaining EK certs in clouds for Cloudproxy

## CloudProxy protocol

The Cloudproxy protocol includes all the steps to provision a Cloudproxy application with a certified Program Key using an attest protocol. In the prototype code, each protocol step is implemented by a command line program for demonstration purposes but intermediate structures are saved in files rather than transmitted over a TCP channel as would be the case in actual use. Command line arguments for each utility are described in the code and sample arguments are provided in the test scripts `testall.sh` and `prototest.sh`.

There is one implemented and one proposed "helper" command utility related to the Cloudproxy protocol.

- The `SigningInstructions` utility specifies policy related to signing certs like endorsement certs and program certs specifying things like CA name and cert durations.
- The (to be written) `PolicyInstructions` utility specifies the conditions under which a Program Key certificate should be signed; in particular, it names the PCRs and their values required by `KeyNegoServer` (the signing authority for an application domain) in order to certify Program Keys.

Preparing the key infrastructure for Cloudproxy for an application domain, happens in three phases[2]:

1. The policy key is generated, self-signed and provisioned to the `KeyNegoServer` along with the per-application-domain policy specifying what Program Key requests should be signed.

[2] Readers should consult the Cloudproxy Tao for Trusted Computing available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf to familiarize themselves with the operation of Cloudproxy and the purposes of each of the keys mentioned in this section.

2. The endorsement key for each physical machine is retrieved and signed by some authority key (we use the Policy Key in our examples) producing the endorsement certificate required for each physical machine. This operation usually happens once as the machine is originally provisioned.
3. Each Cloudproxy program generates a public/private key pair for the *Program Key* and uses the *Program Key Certification Protocol* described below to communicate with `KeyNegoServer`. `KeyNegoServer`, based on information provided by the requesting Cloudproxy application, will sign the public portion of the program key with the Policy Key. Cloudproxy programs can use this certificate to prove identity, isolation and negotiate bi-directional encrypted, integrity protected channels with other Cloudproxy programs.

Steps one and two of the Cloudproxy key provisioning procedure, using Tpm 2.0 are illustrated by series of utilities, they are:

1. Generation and Signing of that Policy Key: There is one policy key for each application. A “self signed” cert naming the public portion of the policy key is embedded in every application program instance and serves as a “root” for all policy decisions enforced in the application domain. The utility `GeneratePolicyKey` generates a 2048-bit RSA signing key and the utility `SelfSignPolicyCert` self signs the generated request.
2. Retrieving the Endorsement Key and Endorsement Certificate: Every every machine running a Cloudproxy program must have an Endorsement Certificate, signed by the Policy Key in our examples,, naming the public endorsement key. It would be nice if every an Endorsement Certificate were provided by the (hardware) platform supplier but it usually isn’t. The utility `GetEndorsementKey` retrieves the public key and machine name and the utility `CloudProxySignEndorsementKey` signs the endorsement key (in our examples, the endorsement key is signed by the Policy Key).
3. The utility `CreateAndSaveCloudProxyKeyHierarchy` generates the key hierarchy for a Cloudproxy program and saves it in a form suitable for reloading by the program at initialization. The utility `RestoreCloudProxyKeyHierarchy` demonstrates reloading the hierarchy. Each Cloudproxy key hierarchy consists of three key contexts:
    a. A primary key, created by `Tpm2_CreatePrimaryKey`, which is the storage root for the program.
    b. A sealing key, created and sealed by `Tpm2_CreateKey` and unsealed by `Tpm2_Unseal`. This key is used to encrypt program secrets.
    c. A quote key, which is used in the Cloudproxy key certification protocol. The quote key is a signing key used in the `Tpm2_Quote` operation.

Step three of the Cloudproxy key provisioning procedure is implemented as an on-line protocol between a Cloudproxy program and a server, `KeyNegoServer`, in a real application. This protocol consists of a three step process.

To illustrate the protocol, we provide three command line utilities which implement the protocol steps. Communication between participants of the Cloudproxy key provisioning procedure takes place using protobufs which are defined in Appendix 1. Our utilities simply store the protobufs in designated files.

The utility `ClientGenerateProgramKeyRequest` implements step 1 of the Cloudproxy protocol which would run in a Cloudproxy application. `ClientGenerateProgramKeyRequest` generates a public/private key pair. The private portion of the key is sealed and stored for later use. It collects the public key along with the machine's endorsement certificate and a Tpm 2.0 quote naming the PCR state and the newly generated program public key as well as the name and qualified name of the quote key along with its public parameters. All this information is packaged in a protobuf (the `program_cert_request_message` message) for transmission to `KeyNegoServer`.

The utility `ServerSignProgramKeyRequest` implements the actions taken by `KeynegoServer` in the "live" protocol. These includes the computations performed by MakeCredential and ActivateCredential above. The responses are packaged in a protobuf (the `program_cert_response_message` message) and transmitted to the requesting application.

The utility `ClientGetProgramKeyCert` implements the final step carried out by the Cloudproxy application upon receipt of a successful response from `Keynegoserver`. This consists of retrieving the protected credential consisting of the encryption/integrity keys used to encrypt the signed Program Certificate using the `Tpm2_ActivateCredential` function and decrypting and storing the unencrypted Program Certificate for later use.

## Source code and tests

You can download the Cloudproxy repository from https://github.com/jlmucb/cloudproxy, the tpm sample code is in cloudproxy/src/tpm2. To make the library and the utilities, type

```
make -f tpm2.mak
```

after setting up the object and binary locations. To run the test scripts, **as root**, type

```
./prototest.sh
./testall.sh
```

in the binary directory.

Tom: Add Cmake directions.

## Acknowledgement

Thanks to Paul England for many helpful discussions.

## Appendix - Protobufs for Cloudproxy Protocol

```
message private_key_blob_message {
  required string key_type                         = 1;
  optional string key_name                         = 2;
  optional bytes blob                              = 3;
}

message rsa_public_key_message {
  optional string key_name                         = 1;
  required int32  bit_modulus_size                 = 2;
  required bytes  exponent                         = 3;
  required bytes  modulus                          = 4;
}

message rsa_private_key_message {
  required rsa_public_key_message public_key  = 1;
  optional bytes  d                                = 2;
  optional bytes  p                                = 3;
  optional bytes  q                                = 4;
  optional bytes  dp                               = 5;
  optional bytes  dq                               = 6;
}

message asymmetric_key_message {
  optional rsa_private_key_message key        = 1;
}

message public_key_message {
  optional string key_type                         = 1;
  optional rsa_public_key_message rsa_key     = 2;
}

message endorsement_key_message {
  optional string machine_identifier          = 1;
  optional bytes tpm2b_blob                        = 2;
  optional bytes tpm2_name                         = 3;
}

message signing_instructions_message {
  optional string issuer                           = 1;
```

```
  optional int64 duration                      = 2;
  optional string purpose                      = 3;
  optional string date                         = 4;
  optional string time                         = 5;
  optional string sign_alg                     = 6;
  optional string hash_alg                     = 7;
  optional bool isCA                           = 8;
  optional bool can_sign                       = 9;
}

message x509_cert_request_parameters_message {
  required string common_name                  = 1;
  optional string country_name                 = 2;
  optional string state_name                   = 3;
  optional string locality_name                = 4;
  optional string organization_name            = 5;
  optional string suborganization_name         = 6;
  optional public_key_message key              = 7;
}

message x509_cert_issuer_parameters_message {
  required string common_name                  = 1;
  optional string country_name                 = 2;
  optional string state_name                   = 3;
  optional string locality_name                = 4;
  optional string organization_name            = 5;
  optional string suborganization_name         = 6;
  optional string purpose                      = 7;
  optional public_key_message key              = 8;
}

message cert_parameters_message {
  optional x509_cert_request_parameters_message request = 1;
  optional x509_cert_issuer_parameters_message signer   = 2;
  optional string not_before                            = 3;
  optional string not_after                             = 4;
}
message credential_info_message {
  // public key parameters of "active-key"
  optional public_key_message public_key       = 1;
  // Tpm2 name (hash) of the "active-key" info
  optional bytes  name                         = 2;
  // objectAttributes of the "active key"
```

```
  optional int32  properties                 = 3;
}

message program_key_parameters {
  optional string program_name               = 1;
  optional string program_key_type           = 2;
  optional int32 program_bit_modulus_size    = 3;
  optional bytes program_key_exponent        = 4;
  optional bytes program_key_modulus         = 5;
};

message program_cert_request_message {
  optional string request_id                 = 1;
  optional bytes endorsement_cert_blob       = 2;
  optional program_key_parameters program_key = 3;
  optional string active_sign_alg            = 4;
  optional int32 active_sign_bit_size        = 5;
  optional string active_sign_hash_alg       = 6;
  optional bytes active_signature            = 7;
  optional credential_info_message cred      = 8;
  optional bytes quoted_blob                 = 9;
}

message program_cert_response_message {
  optional string request_id                 = 1;
  optional string program_name               = 2;
  optional string integrity_alg              = 3;
  // outer HMAC, does not include size in buffer
  // HMAC key is KDFa derived from seed and "INTEGRITY"
  // This is a TPM2B_DIGEST and has a size.
  optional bytes integrityHMAC               = 4;
  // encIdentity, does not include size of encIdentity in buffer.
  // encIdentity should be an encrypted correctly marshalled
  // This is an encrypted TPM2B_DIGEST and has a size.
  // encIdentity is always CFB Aes-128 encrypted
  // with KDFa derived key derived from the "seed," "STORAGE" and
  // the name of the active key.
  optional bytes encIdentity                 = 5;
  // protector-key private-key encrypted seed || "IDENTITY" buffer
  optional bytes secret                      = 6;
  // Signed, der-encoded program cert CTR encrypted with
  // secret in credential buffer.  TODO(jlm): should also
  // contain an HMAC.
```

```
  optional bytes encrypted_cert                = 7;
  optional bytes encrypted_cert_hmac           = 8;
}

message certificate_chain_entry_message {
  optional string subject_key_name             = 1;
  optional string issuer_key_name              = 2;
  optional string cert_type                    = 3;
  optional bytes cert_blob                     = 4;
}

message certificate_chain_message {
  repeated certificate_chain_entry_message entry  = 1;
}

message quote_certification_information {
  optional bytes magic                         = 1;
  optional bytes type                          = 2;
  optional bytes qualifiedsigner               = 3;
  optional bytes extraData                     = 4;
  optional bytes clockinfo                     = 5;
  optional int64 firmwareversion               = 6;
  optional bytes pcr_selection                 = 7;
  optional bytes digest                        = 8;
}
```

# Appendix 2 - tpm2_lib functions

```
int Tpm2_SetCommand(TPM_ST tag, uint32_t cmd, byte* buf, int
size_param, byte* params);
void Tpm2_IntepretResponse(int out_size, byte* out_buf, int16_t* cap,
uint32_t* responseSize, uint32_t* responseCode);
int Tpm2_Set_OwnerAuthHandle(int size, byte* buf);
int Tpm2_Set_OwnerAuthData(int size, byte* buf)
bool Tpm2_Startup(LocalTpm& tpm);
bool Tpm2_Shutdown(LocalTpm& tpm);
bool Tpm2_GetCapability(LocalTpm& tpm, uint32_t cap, int* size, byte*
buf);
bool Tpm2_GetRandom(LocalTpm& tpm, int numBytes, byte* buf);
bool Tpm2_ReadClock(LocalTpm& tpm, uint64_t* current_time, uint64_t*
current_clock);
bool Tpm2_ReadPcrs(LocalTpm& tpm, TPML_PCR_SELECTION pcrSelect,
uint32_t* updateCounter, TPML_PCR_SELECTION* pcrSelectOut,
TPML_DIGEST* values);
bool Tpm2_ReadPcr(LocalTpm& tpm, int pcrNum, uint32_t* updateCounter,
TPML_PCR_SELECTION* pcrSelectOut, TPML_DIGEST* digest);
bool Tpm2_CreatePrimary(LocalTpm& tpm, TPM_HANDLE owner, string&
authString, TPML_PCR_SELECTION& pcr_selection, TPM_ALG_ID enc_alg,
TPM_ALG_ID int_alg, TPMA_OBJECT& flags, TPM_ALG_ID sym_alg,
TPMI_AES_KEY_BITS sym_key_size, TPMI_ALG_SYM_MODE sym_mode,
TPM_ALG_ID sig_scheme, int mod_size, uint32_t exp, TPM_HANDLE*
handle, TPM2B_PUBLIC* pub_out);
bool Tpm2_Load(LocalTpm& tpm, TPM_HANDLE parent_handle, string&
parentAuth, int size_public, byte* inPublic, int size_private, byte*
inPrivate, TPM_HANDLE* new_handle, TPM2B_NAME* name);
bool Tpm2_PolicyPassword(LocalTpm& tpm, TPM_HANDLE handle);
bool Tpm2_PCR_Event(LocalTpm& tpm, int pcr_num, uint16_t size, byte*
eventData);
bool Tpm2_PolicyGetDigest(LocalTpm& tpm, TPM_HANDLE handle,
TPM2B_DIGEST* digest_out);
bool Tpm2_StartAuthSession(LocalTpm& tpm, TPM_RH tpm_obj, TPM_RH
bind_obj, TPM2B_NONCE& initial_nonce, TPM2B_ENCRYPTED_SECRET& salt,
TPM_SE session_type, TPMT_SYM_DEF& symmetric, TPMI_ALG_HASH hash_alg,
TPM_HANDLE* session_handle, TPM2B_NONCE* nonce_obj);
bool Tpm2_PolicyPcr(LocalTpm& tpm, TPM_HANDLE session_handle,
TPM2B_DIGEST& expected_digest, TPML_PCR_SELECTION& pcr);
bool Tpm2_PolicySecret(LocalTpm& tpm, TPM_HANDLE handle,TPM2B_DIGEST*
policy_digest, TPM2B_TIMEOUT* timeout, TPMT_TK_AUTH* ticket);
```

```
bool Tpm2_CreateSealed(LocalTpm& tpm, TPM_HANDLE parent_handle,
int size_policy_digest, byte* policy_digest, string& parentAuth, int
size_to_seal, byte* to_seal, TPML_PCR_SELECTION& pcr_selection,
TPM_ALG_ID int_alg, TPMA_OBJECT& flags, TPM_ALG_ID sym_alg,
TPMI_AES_KEY_BITS sym_key_size, TPMI_ALG_SYM_MODE sym_mode,
TPM_ALG_ID sig_scheme, int mod_size, uint32_t exp, int* size_public,
byte* out_public, int* size_private, byte* out_private,
TPM2B_CREATION_DATA* creation_out, TPM2B_DIGEST* digest_out,
TPMT_TK_CREATION* creation_ticket);
bool Tpm2_CreateKey(LocalTpm& tpm, TPM_HANDLE parent_handle,
string& parentAuth, string& authString, TPML_PCR_SELECTION&
pcr_selection, TPM_ALG_ID enc_alg, TPM_ALG_ID int_alg, TPMA_OBJECT&
flags, TPM_ALG_ID sym_alg,TPMI_AES_KEY_BITS sym_key_size,
TPMI_ALG_SYM_MODE sym_mode, TPM_ALG_ID sig_scheme, int mod_size,
uint32_t exp,int* size_public, byte* out_public, int* size_private,
byte* out_private, TPM2B_CREATION_DATA* creation_out,  TPM2B_DIGEST*
digest_out, TPMT_TK_CREATION* creation_ticket);

bool Tpm2_Unseal(LocalTpm& tpm, TPM_HANDLE item_handle, string&
parentAuth, TPM_HANDLE session_handle, TPM2B_NONCE& nonce, byte
session_attributes, TPM2B_DIGEST& hmac_digest, int* out_size, byte*
sealed);
bool Tpm2_Quote(LocalTpm& tpm, TPM_HANDLE signingHandle, string&
parentAuth, int quote_size, byte* toQuote, TPMT_SIG_SCHEME scheme,
TPML_PCR_SELECTION& pcr_selection, TPM_ALG_ID sig_alg, TPM_ALG_ID
hash_alg, int* attest_size, byte* attest, int* sig_size, byte* sig);
bool Tpm2_LoadContext(LocalTpm& tpm, int size, byte* saveArea,
TPM_HANDLE* handle);
bool Tpm2_SaveContext(LocalTpm& tpm, TPM_HANDLE handle, int* size,
byte* saveArea);
bool Tpm2_FlushContext(LocalTpm& tpm, TPM_HANDLE handle);

bool Tpm2_ReadNv(LocalTpm& tpm, TPMI_RH_NV_INDEX index, string&
authString, uint16_t size, byte* data);
bool Tpm2_WriteNv(LocalTpm& tpm, TPMI_RH_NV_INDEX index, string&
authString,uint16_t size, byte* data);
bool Tpm2_DefineSpace(LocalTpm& tpm, TPM_HANDLE owner,
TPMI_RH_NV_INDEX index, string& authString, uint16_t size_data);
bool Tpm2_UndefineSpace(LocalTpm& tpm, TPM_HANDLE owner,
TPMI_RH_NV_INDEX index);
bool Tpm2_Flushall(LocalTpm& tpm);
```

```
bool Tpm2_MakeCredential(LocalTpm& tpm, TPM_HANDLE keyHandle,
TPM2B_DIGEST& credential, TPM2B_NAME& objectName, TPM2B_ID_OBJECT*
credentialBlob, TPM2B_ENCRYPTED_SECRET* secret);
bool Tpm2_ActivateCredential(LocalTpm& tpm, TPM_HANDLE activeHandle,
TPM_HANDLE keyHandle, string& activeAuth, string& keyAuth,
TPM2B_ID_OBJECT& credentialBlob, TPM2B_ENCRYPTED_SECRET& secret,
TPM2B_DIGEST* certInfo);
bool Tpm2_Certify(LocalTpm& tpm, TPM_HANDLE signedKey, TPM_HANDLE
signingKey, string& auth_signed_key, string& auth_signing_key,
TPM2B_DATA& qualifyingData, TPM2B_ATTEST* attest, TPMT_SIGNATURE*
sig);
bool Tpm2_ReadPublic(LocalTpm& tpm, TPM_HANDLE handle, uint16_t*
pub_blob_size, byte* pub_blob, TPM2B_PUBLIC& outPublic, TPM2B_NAME&
name, TPM2B_NAME& qualifiedName);
bool Tpm2_Rsa_Encrypt(LocalTpm& tpm, TPM_HANDLE handle, string&
authString, TPM2B_PUBLIC_KEY_RSA& in, TPMT_RSA_DECRYPT& scheme,
TPM2B_DATA& label, TPM2B_PUBLIC_KEY_RSA* out);
bool Tpm2_EvictControl(LocalTpm& tpm, TPMI_RH_PROVISION owner,
TPM_HANDLE handle, string& authString,  TPMI_DH_PERSISTENT*
persistantHandle);
bool Tpm2_DictionaryAttackLockReset(LocalTpm& tpm);
```