CHAPTER 17

Decrypt/Encrypt Sessions

2B or not 2B, that is the question.

Dave Challener,
During TCG TSS Working Group discussion of
decrypt/encrypt sessions

Chapter 13 briefly touched on decrypt and encrypt sessions. As you may remember, these are per-command session modifiers. This chapter describes these two session modifiers in detail: what they do, practical uses of them, some limitations on them, how to set them up, and some code examples.

What Do Encrypt/Decrypt Sessions Do?

In a nutshell, decrypt and encrypt sessions protect secrets being transmitted over an insecure medium. A caller, to protect the confidentiality of data, can encrypt it with a command encryption key known only to the caller and the TPM. The encryption key is determined, in part, by the parameters used to start the session (more on that later). A decrypt session then informs the TPM that the first parameter is encrypted. This means after receiving the parameter, the TPM must decrypt it—hence the name, *decrypt session*. For a response, an encrypt session indicates that the TPM has encrypted the first response parameter before returning it to the caller, which is why it's called an *encrypt session*. After receiving the encrypted response parameter, the caller uses the response decryption key to decrypt the data.

Two different symmetric-key modes can be used for decrypt and encrypt sessions: XOR and CFB. CFB mode offers stronger encryption but requires that the TPM and the caller both have access to a hashing algorithm and an encryption algorithm. XOR requires only a hashing algorithm and is the right choice for very small code size, but it is less secure.

Practical Use Cases

So, what are these symmetric key modes good for? The quick answer is that there are many ways to use them; just look in Part 3 of the TPM 2.0 specification for every command that has a `TPM2B` as a first command and/or first response parameter. All of those parameters are possible candidates to be encrypted.

A small sampling of common use cases are as follows:

- `Tpm2_Create`: The first command parameter to this command, `inSensitive`, is a structure which contains the password (called `userAuth` in the structure description) in one of its fields. This should probably be sent to the TPM encrypted, which would require that the session be set up as a decrypt session.[1]
- Confidential data being written to or read from TPM NV indexes. Suppose you want to use the TPM NV indexes to save password information or personal credit card information. Encrypting this data before sending it to or receiving it from the TPM helps protect it.
- Use of decrypt and encrypt sessions becomes even more important when communicating with a remote TPM over the network. Suppose you want to store keys on a remote server and recover them from client machines. Sending these in the clear over the network is obviously insecure. SSL sessions can remedy the network snooping vulnerability, but the keys are still in the clear in multiple software layers on the client and server machines. Encrypt and decrypt sessions can vastly reduce the attack surface.

Decrypt/Encrypt Limitations

There are some limitations on which parameters can be encrypted and decrypted and the number of encrypt and decrypt sessions per command.

Only the first command parameter can be encrypted and only the first response parameter can be decrypted, and in both cases, only if that first parameter is a `TPM2B` as defined in Chapter 5. Commands that don't have a `TPM2B` as the first command parameter cannot be sent to the TPM using sessions set for decrypt; likewise, if a response's first parameter isn't a `TPM2B`, the response can't be received using an encrypt session.

As you learned in Chapter 13, commands can be sent with up to three sessions. But a maximum of one session per command can be set for decrypt and a maximum of one for encrypt. If a command allows the use of both decrypt and encrypt sessions, the same session can be used to set both attributes or separate sessions can be used, one for each attribute.

So how do you enable decrypt and encrypt sessions?

[1]There's an interesting wrinkle related to the first response parameter from `Tpm2_Create`: even though this parameter is a `TPM2B` and could be encrypted by setting the session as an encrypt session, it's always encrypted by the TPM. Encrypting it again would seem to be of little value.

Decrypt/Encrypt Setup

At first glance, configuring sessions as decrypt and/or encrypt sessions is very easy. For an open session, all you have to do is set either or both of the session attributes bits in the authorization area for the command: `sessionAttributes.decrypt` and/or `sessionAttributes.encrypt`.

Of course, things are rarely that simple, and it's certainly true here. For a decrypt session, the caller has to properly encrypt the first parameter. Likewise for an encrypt session, the caller has to properly decrypt the first response parameter after receiving it from the TPM; otherwise, it will be meaningless gibberish to the caller. Two modes of encryption are used for decrypt and encrypt sessions: XOR and CFB mode. These modes are set when the session is created. Both modes have the property that the plain text and ciphertext are the same length, so the byte stream lengths don't change. Session nonces figure into the encryption, which ensures that the encryption and decryption operations function as one-time pads.

For XOR mode, a *mask* (one-time pad) is generated and XORed with the data to be encrypted or decrypted. The mask is generated by passing the `hashAlg` (`authHash` parameter used when the session was started), the HMAC key, the string "XOR", `nonceNewer`, `nonceOlder`, and the message size to the key derivation function (KDFa). The output is a mask that is as long as the message to be decrypted or encrypted. A simple XOR of the mask with the data completes the encryption or decryption operation.

For CFB mode, the KDFa is used to generate the encryption key and initialization vector (IV). The inputs to the KDFa are `hashAlg` (the `authHash` parameter used when the session was started), `sessionKey`, the "CFB" string, `nonceNewer`, `nonceOlder`, and the number of bits (`bits`) needed for the symmetric key plus the IV. The output is a string of `bits` length, with the key in the upper octets and the IV in the lower octets. The IV size is determined by the block size of the encryption algorithm. The key and IV are used as inputs to the encryption algorithm to perform the required encryption or decryption operation.

For both XOR and CFB modes, `nonceNewer` and `nonceOlder` figure into the encryption. For XOR mode, because the nonces change, a different mask is generated for encryption of command parameters than the one used for response parameters. Likewise, for CFB mode, a different encryption key and IV are generated for commands and responses. In both XOR and CFB modes, because the nonces roll for every usage of the session, encrypt and decrypt sessions act as one-time pads.

Pseudocode Flow

As you may recall from Chapter 13, sessions can be one of three types: HMAC, policy, or trial policy sessions. HMAC and policy sessions can be used as decrypt or encrypt sessions; trial policy sessions cannot.

To keep things very simple, the following example uses an unbound, unsalted policy session that isn't being used for authorization.[2] The only use of this session is for decryption and encryption of command and response parameters. A separate password session is used for authorization. This means the test code doesn't need to calculate HMACs or manage the `policyDigest` for the encrypt and decrypt session.

[2]Unbound and unsalted sessions don't yield strong encryption keys and should not normally be used for decrypt or encrypt sessions. This was done to keep the example as simple as possible.

When a session is started, the TPM generates a session key. To use decrypt and encrypt sessions, the caller needs to independently generate that session key, just as he had to do in order to use HMAC and policy sessions.

To unify all this into a single flow, the steps in decrypt and encrypt session lifecycles are as follows:

1. Start the session using `Tpm2_StartAuthSession`, and set the symmetric parameter to

   - CFB mode:

     ```
     // AES encryption/decryption and CFB mode.
     symmetric.algorithm = TPM_ALG_AES;
     symmetric.keyBits.aes = 128;
     symmetric.mode.aes = TPM_ALG_CFB;
     ```

   - XOR mode:

     ```
     // XOR encryption/decryption.
     symmetric.algorithm = TPM_ALG_XOR;
     symmetric.keyBits.exclusiveOr = TPM_ALG_SHA256;
     ```

2. Generate the session key, and save it.

3. For a command that has a `TPM2B` as the first parameter, if you desire to encrypt that parameter, do the following:

   a. Generate the HMAC key for this use of the session. The session key figures into the generation of this key.

   b. For CFB mode:

      - Generate the encryption key and IV using the session hash algorithm, HMAC key, special label ("CFB"), `nonceNewer`, `nonceOlder`, and the number of bits to be encrypted.

      - Encrypt the first parameter, using the encryption key and IV.

   c. For XOR mode:

      - Generate the mask using the HMAC key, the session hash algorithm, `nonceNewer`, `nonceOlder`, and the number of bytes to be encrypted.

      - XOR the clear text data with the mask to generate the encrypted data.

   d. Set the `sessionAttributes.decrypt` bit.

4. If the first response parameter is a `TPM2B` and you want the TPM to send that parameter in encrypted format, set the `sessionAttributes.encrypt` bit.
5. Send the command to the TPM.
6. Receive the response from the TPM.
7. If the first response parameter is a `TPM2B` and the `sessionAttributes.encrypt` bit is set, do the following:
    a. Generate the HMAC key for this use of the session. The session key figures into the generation of this key.
    b. For CFB mode:
        - Generate the encryption key and IV using the session hash algorithm, HMAC key, special label ("CFB"), `nonceNewer`, `nonceOlder`, and the number of bits to be decrypted.
        - Decrypt the first parameter, using the encryption key and IV.
    c. For XOR mode:
        - Generate the mask using the HMAC key, the session hash algorithm, `nonceNewer`, `nonceOlder`, and the number of bytes to be decrypted.
        - XOR the encrypted data with the mask to generate the clear data.

For details on CFB and XOR decryption/encryption see the "Session-based encryption" section of Part 1 of the TPM 2.0 specification.

Sample Code

This section shows an example of actual working code for doing decrypt and encrypt sessions. First some notes about this code:

- This code does a write of encrypted data to an NV index (`decrypt` session attribute set) followed by two reads from the same NV index: a plain text read (`encrypt` attribute not set) and a ciphertext read (`encrypt` session attribute set). After both reads, the read data is compared to the plain text write data.

■ **Note** The reason for doing the plain text read is to verify that the NV index was written with plain text, not encrypted data. If you didn't set the `decrypt` session attribute, encrypted data would be written to the NV index. But the test would still appear to be working because the encrypted data would be written to the TPM, read back, and decrypted by the calling application, and the test to verify the read and write data would pass. This was actually a mistake that I made on my first pass at writing this code.

To catch this issue, do a plain text read of the NV index and compare this to the unencrypted write data. They should be equal.

- This function tests both CFB and XOR mode encryption. CFB is done on the first pass and XOR on the second pass.
- The code demonstrates some new features of the TSS system API code that couldn't be discussed earlier:[3]
  - *Getting and setting of encrypt and decrypt parameters*: These calls enable the caller to get the plain text unencrypted command parameters (`Tss2_Sys_GetDecryptParam`), encrypt them, and then set the encrypted command parameters in the command byte stream (`Tss2_Sys_SetDecryptParam`) before sending the command. Likewise, `Tss2_Sys_GetEncryptParam` and `Tss2_Sys_SetEncryptParam` enable the caller to properly process response parameters that were encrypted by the TPM.
  - *Asynchronous execution* (`Tss2_Sys_ExecuteAsync` and `Tss2_Sys_ExecuteFinish`): This mode of execution allows the application to send the command (`Tss2_Sys_ExecuteAsync`), do some work while waiting for the response, and then get the response (`Tss2_Sys_ExecuteFinish`) with a configurable timeout.
  - *Synchronous execution calls* (Tss2_Sys_Execute): This function will wait forever for a response, so it assumes that the TPM eventually responds.

[3]For details on these system API calls, review the TSS System Level API and TPM Command Transmission Interface Specification at `www.trustedcomputinggroup.org/developers/software_stack`.

- *Setting command authorizations* (`Tss2_SetCmdAuths`) *and getting response authorizations* (`Tss2_GetRspAuths`): These functions are used to set command authorization area parameters and get response area parameters, including nonces, session attributes, passwords (for password sessions), and command and response HMACs. In this example they will be used for nonces, session attributes, and the password. Access to the command and response HMACs isn't needed in this code since the code doesn't use HMACs.

- This code relies heavily on an application-level structure, `SESSION`, that maintains all session state information including nonces. There are many ways this can be done—this just happens to be the implementation I chose. This structure looks like this:

```
typedef struct {
    // Inputs to StartAuthSession; these need to be saved
    // so that HMACs can be calculated.
    TPMI_DH_OBJECT tpmKey;
    TPMI_DH_ENTITY bind;
    TPM2B_ENCRYPTED_SECRET encryptedSalt;
    TPM2B_MAX_BUFFER salt;
    TPM_SE sessionType;
    TPMT_SYM_DEF symmetric;
    TPMI_ALG_HASH authHash;

    // Outputs from StartAuthSession; these also need
    // to be saved for calculating HMACs and
    // other session related functions.
    TPMI_SH_AUTH_SESSION sessionHandle;
    TPM2B_NONCE nonceTPM;

    // Internal state for the session
    TPM2B_DIGEST sessionKey;
    TPM2B_DIGEST authValueBind; // authValue of bind object
    TPM2B_NONCE nonceNewer;
    TPM2B_NONCE nonceOlder;
    TPM2B_NONCE nonceTpmDecrypt;
    TPM2B_NONCE nonceTpmEncrypt;
    TPM2B_NAME name;    // Name of the object the session handle
                        // points to. Used for computing HMAC for
                        // any HMAC sessions present.
                        //
    void *hmacPtr;      // Pointer to HMAC field in the marshalled
                        // data stream for the session.
                        // This allows the function to calculate
```

```
                        // and fill in the HMAC after marshalling
                        // of all the inputs is done.
                        //
                        // This is only used if the session is an
                        // HMAC session.
                        //
    UINT8 nvNameChanged;// Used for some special case code
                        // dealing with the NV written state.
} SESSION;
```

- The `RollNonces` function does what it says: it copies `nonceNewer` to `nonceOlder` and copies the new nonce to `nonceNewer`. The nonces must be rolled before each command and after each response, as described in Chapter 13. Here's the complete code for this function:

```
void RollNonces( SESSION *session, TPM2B_NONCE *newNonce )
{
    session->nonceOlder = session->nonceNewer;
    session->nonceNewer = *newNonce;
}
```

- The `StartAuthSessionWithParams` function starts the session, saves its state in a `SESSION` structure, and adds the `SESSION` structure to a list of open sessions.
- The `EndAuthSession` function is used to remove the `SESSION` structure from the list of open sessions after the session has ended.
- `EncryptCommandParam` encrypts command parameters, and `DecryptResponseParam` decrypts response parameters. Both functions examine the authorization structures' `TPMA_SESSION` bits to determine if the decrypt and/or encrypt bits are set. This chapter doesn't describe the details of these functions, but they perform the encryption and decryption operations as explained in Part 1 of the TPM 2.0 specification.
- For some of the common routines and data structures that aren't described here, please refer to Chapters 7 and 13 as well as the TSS System API specification.

This working code can be downloaded in source form as part of the TSS System API library code and tests. Because the code is a bit long, to help you better understand the flow, notes are interspersed before each major block of functionality. And now for the actual code:

```
UINT32 writeDataString = 0xdeadbeef;

void TestEncryptDecryptSession()
{
    TSS2_RC             rval = TSS2_RC_SUCCESS;
    SESSION             encryptDecryptSession;
    TPMT_SYM_DEF        symmetric;
    TPM2B_MAX_NV_BUFFER writeData, encryptedWriteData;
    TPM2B_MAX_NV_BUFFER encryptedReadData, decryptedReadData,
                        readData;
    size_t              decryptParamSize;
    uint8_t             *decryptParamBuffer;
    size_t              encryptParamSize;
    uint8_t             *encryptParamBuffer;
    TPM2B_AUTH          nvAuth;
    TPM2B_DIGEST        authPolicy;
    TPMA_NV             nvAttributes;
    int                 i;
    TPMA_SESSION        sessionAttributes;
```

The following lines set up the authorization used for the NV `Undefine` command.

```
// Authorization structure for undefine command.
TPMS_AUTH_COMMAND nvUndefineAuth;

// Create and init authorization area for undefine command:
// only 1 authorization area.
TPMS_AUTH_COMMAND *nvUndefineAuthArray[1] = { &nvUndefineAuth };

// Authorization array for command (only has one auth structure).
TSS2_SYS_CMD_AUTHS nvUndefineAuths = { 1, &nvUndefineAuthArray[0] };

printf( "\n\nDECRYPT/ENCRYPT SESSION TESTS:\n" );
```

Copy the write data array into a TPM2B structure.

```
writeData.t.size = sizeof( writeDataString );
memcpy( (void *)&writeData.t.buffer, (void *)&writeDataString,
        sizeof( writeDataString ) );
```

Create the NV index.

```
// Create NV index with empty auth value.
*(UINT32 *)( (void *)&nvAttributes ) = 0;
nvAttributes.TPMA_NV_AUTHREAD = 1;
nvAttributes.TPMA_NV_AUTHWRITE = 1;
nvAttributes.TPMA_NV_PLATFORMCREATE = 1;

// No authorization required.
authPolicy.t.size = 0;
nvAuth.t.size = 0;
rval = DefineNvIndex( TPM_RH_PLATFORM, TPM_RS_PW,
        &nvAuth, &authPolicy, TPM20_INDEX_TEST1,
        TPM_ALG_SHA1, nvAttributes,
        sizeof( writeDataString ) );

//
// 1st pass with CFB mode.
// 2nd pass with XOR mode.
//
for( i = 0; i < 2; i++ )
{
```

Set up authorization structures for NV `read` and `write` commands and responses.

```
// Authorization structure for NV
// read/write commands.
TPMS_AUTH_COMMAND nvRdWrCmdAuth;

// Authorization structure for
// encrypt/decrypt session.
TPMS_AUTH_COMMAND decryptEncryptSessionCmdAuth;
```

```
// Create and init authorization area for
// NV read/write commands:
// 2 authorization areas.
TPMS_AUTH_COMMAND *nvRdWrCmdAuthArray[2] =
        { &nvRdWrCmdAuth, &decryptEncryptSessionCmdAuth };

// Authorization array for commands
// (has two auth structures).
TSS2_SYS_CMD_AUTHS nvRdWrCmdAuths =
        { 2, &nvRdWrCmdAuthArray[0] };

// Authorization structure for NV read/write responses.
TPMS_AUTH_RESPONSE nvRdWrRspAuth;

// Authorization structure for decrypt/encrypt
// session responses.
TPMS_AUTH_RESPONSE decryptEncryptSessionRspAuth;

// Create and init authorization area for NV
// read/write responses:  2 authorization areas.
TPMS_AUTH_RESPONSE *nvRdWrRspAuthArray[2] =
        { &nvRdWrRspAuth, &decryptEncryptSessionRspAuth };

// Authorization array for responses
// (has two auth structures).
TSS2_SYS_RSP_AUTHS nvRdWrRspAuths =
        { 2, &nvRdWrRspAuthArray[0] };
```

Set the session for CFB or XOR mode encryption/decryption, depending on which pass through the code is being run. Then start the policy session.

```
// Setup session parameters.
if( i == 0 )
{
    // AES encryption/decryption and CFB mode.
    symmetric.algorithm = TPM_ALG_AES;
    symmetric.keyBits.aes = 128;
    symmetric.mode.aes = TPM_ALG_CFB;
}
else
{
    // XOR encryption/decryption.
    symmetric.algorithm = TPM_ALG_XOR;
    symmetric.keyBits.exclusiveOr = TPM_ALG_SHA256;
}
```

```
// Start policy session for decrypt/encrypt session.
rval = StartAuthSessionWithParams( &encryptDecryptSession,
       TPM_RH_NULL, TPM_RH_NULL, 0, TPM_SE_POLICY,
       &symmetric, TPM_ALG_SHA256 );
CheckPassed( rval );
```

Write the NV index using a password session for authorization and a policy session for encryption/decryption. First marshal the input parameters (`Tss2_Sys_NV_Prepare`).

```
//
// Write TPM index with encrypted parameter used
// as the data to write. Set session for encrypt.
// Use asynchronous APIs to do this.
//
// 1st time:  use null buffer, 2nd time use populated one;
// this tests different cases for SetDecryptParam function.
//

// Prepare the input parameters, using unencrypted
// write data. This will be encrypted before the
// command is sent to the TPM.
rval = Tss2_Sys_NV_Write_Prepare( sysContext,
        TPM20_INDEX_TEST1, TPM20_INDEX_TEST1,
        ( i == 0 ? (TPM2B_MAX_NV_BUFFER *)0 : &writeData ),
        0 );
CheckPassed( rval );
```

Set the authorization structures (`Tss2_Sys_SetCmdAuths`) for the command.

```
// Set up password authorization session structure.
nvRdWrCmdAuth.sessionHandle = TPM_RS_PW;
nvRdWrCmdAuth.nonce.t.size = 0;
*( (UINT8 *)((void *)&nvRdWrCmdAuth.sessionAttributes ) ) = 0;
nvRdWrCmdAuth.hmac.t.size = nvAuth.t.size;
memcpy( (void *)&nvRdWrCmdAuth.hmac.t.buffer[0],
        (void *)&nvAuth.t.buffer[0],
        nvRdWrCmdAuth.hmac.t.size );

// Set up encrypt/decrypt session structure.
decryptEncryptSessionCmdAuth.sessionHandle =
        encryptDecryptSession.sessionHandle;
decryptEncryptSessionCmdAuth.nonce.t.size = 0;
*( (UINT8 *)((void *)&sessionAttributes ) ) = 0;
```

```
decryptEncryptSessionCmdAuth.sessionAttributes =
        sessionAttributes;
decryptEncryptSessionCmdAuth.sessionAttributes.continueSession
        = 1;
decryptEncryptSessionCmdAuth.sessionAttributes.decrypt = 1;
decryptEncryptSessionCmdAuth.hmac.t.size = 0;

rval = Tss2_Sys_SetCmdAuths( sysContext, &nvRdWrCmdAuths );
CheckPassed( rval );
```

Get the location and size of the decrypt parameter in the byte stream (`Tss2_Sys_GetDecryptParam`), encrypt the write data (`EncryptCommandParam`), and copy the encrypted write data into the byte stream (`Tss2_Sys_SetDecryptParam`).

```
// Get decrypt parameter.
rval = Tss2_Sys_GetDecryptParam( sysContext,
        &decryptParamSize,
        (const uint8_t **)&decryptParamBuffer );
CheckPassed( rval );

if( i == 0 )
{
    // 1st pass:  test case of Prepare inputting a NULL decrypt
    // param; decryptParamSize should be 0.
    if( decryptParamSize != 0 )
    {
        printf( "ERROR!! decryptParamSize != 0\n" );
        Cleanup();
    }
}

// Roll nonces for command.
RollNonces( &encryptDecryptSession,
        &decryptEncryptSessionCmdAuth.nonce );

// Encrypt write data.
rval = EncryptCommandParam( &encryptDecryptSession,
        (TPM2B_MAX_BUFFER *)&encryptedWriteData,
        (TPM2B_MAX_BUFFER *)&writeData, &nvAuth );
CheckPassed( rval );

// Now set decrypt parameter.
rval = Tss2_Sys_SetDecryptParam( sysContext,
        (uint8_t )encryptedWriteData.t.size,
        (uint8_t *)&encryptedWriteData.t.buffer[0] );
CheckPassed( rval );
```

Write the NV data (`Tss2_Sys_ExecuteAsync` and `Tss2_Sys_ExecuteFinish`). The write uses asynchronous calls to illustrate this feature of the TSS System API.

```
// Now write the data to the NV index.
rval = Tss2_Sys_ExecuteAsync( sysContext );
CheckPassed( rval );

rval = Tss2_Sys_ExecuteFinish( sysContext, -1 );
CheckPassed( rval );
```

Get the response authorizations to set up for the next use of the sessions (`Tss2_Sys_GetRspAuths`).

```
rval = Tss2_Sys_GetRspAuths( sysContext, &nvRdWrRspAuths );
CheckPassed( rval );

// Roll the nonces for response
RollNonces( &encryptDecryptSession,
        &nvRdWrRspAuths.rspAuths[1]->nonce );

// Don't need nonces for anything else, so roll
// the nonces for next command.RollNonces( &encryptDecryptSession,
        &decryptEncryptSessionCmdAuth.nonce );
```

Read the data back as plain text to be sure the decrypt session worked correctly during the NV write operation.

```
// Now read the data without encrypt set.
nvRdWrCmdAuths.cmdAuthsCount = 1;
nvRdWrRspAuths.rspAuthsCount = 1;
rval = Tss2_Sys_NV_Read( sysContext, TPM20_INDEX_TEST1,
        TPM20_INDEX_TEST1, &nvRdWrCmdAuths,
        sizeof( writeDataString ), 0, &readData,
        &nvRdWrRspAuths );
CheckPassed( rval );
nvRdWrCmdAuths.cmdAuthsCount = 2;
nvRdWrRspAuths.rspAuthsCount = 2;
```

```
// Roll the nonces for response
RollNonces( &encryptDecryptSession,
        &nvRdWrRspAuths.rspAuths[1]->nonce );

// Check that write and read data are equal. This
// verifies that the decrypt session was set up correctly.
// If it wasn't, the data stored in the TPM would still
// be encrypted, and this test would fail.
if( memcmp( (void *)&readData.t.buffer[0],
        (void *)&writeData.t.buffer[0], readData.t.size ) )
{
    printf( "ERROR!! read data not equal to written data\n" );
    Cleanup();
}
```

Now read the NV data encrypted using an encrypt session. This time, use a synchronous call, `Tss2_Sys_Execute`. The reason is simply to demonstrate another method; you could use asynchronous calls similar to how the NV write was performed.

```
//
// Read TPM index with encrypt session; use
// synchronous APIs to do this.
//

rval = Tss2_Sys_NV_Read_Prepare( sysContext, TPM20_INDEX_TEST1,
        TPM20_INDEX_TEST1, sizeof( writeDataString ), 0 );
CheckPassed( rval );

// Roll the nonces for next command.
RollNonces( &encryptDecryptSession,
        &decryptEncryptSessionCmdAuth.nonce );

decryptEncryptSessionCmdAuth.sessionAttributes.decrypt = 0;
decryptEncryptSessionCmdAuth.sessionAttributes.encrypt = 1;
decryptEncryptSessionCmdAuth.sessionAttributes.continueSession = 1;

rval = Tss2_Sys_SetCmdAuths( sysContext, &nvRdWrCmdAuths );
CheckPassed( rval );

//
// Now Read the data.
//
rval = Tss2_Sys_Execute( sysContext );
CheckPassed( rval );
```

Use `Tss2_Sys_GetEncryptParam` and `Tss2_Sys_SetEncryptParam` combined with `DecryptResponseParam` to decrypt the response data.

```
rval = Tss2_Sys_GetEncryptParam( sysContext, &encryptParamSize,
        (const uint8_t **)&encryptParamBuffer );
CheckPassed( rval );

rval = Tss2_Sys_GetRspAuths( sysContext, &nvRdWrRspAuths );
CheckPassed( rval );

// Roll the nonces for response
RollNonces( &encryptDecryptSession,
        &nvRdWrRspAuths.rspAuths[1]->nonce );

// Decrypt read data.
encryptedReadData.t.size = encryptParamSize;
memcpy( (void *)&encryptedReadData.t.buffer[0],
        (void *)encryptParamBuffer, encryptParamSize );

rval = DecryptResponseParam( &encryptDecryptSession,
        (TPM2B_MAX_BUFFER *)&decryptedReadData,
        (TPM2B_MAX_BUFFER *)&encryptedReadData, &nvAuth );
CheckPassed( rval );

// Roll the nonces.
RollNonces( &encryptDecryptSession,
        &nvRdWrRspAuths.rspAuths[1]->nonce );

rval = Tss2_Sys_SetEncryptParam( sysContext,
        (uint8_t)decryptedReadData.t.size,
        (uint8_t *)&decryptedReadData.t.buffer[0] );
CheckPassed( rval );

// Get the command results, in this case the read data.
rval = Tss2_Sys_NV_Read_Complete( sysContext, &readData );
CheckPassed( rval );

printf( "Decrypted read data = " );
DEBUG_PRINT_BUFFER( &readData.t.buffer[0], (UINT32 )readData.t.size );
```

```
// Check that write and read data are equal.
if( memcmp( (void *)&readData.t.buffer[0],
        (void *)&writeData.t.buffer[0], readData.t.size ) )
{
    printf( "ERROR!! read data not equal to written data\n" );
    Cleanup();
}

rval = Tss2_Sys_FlushContext( sysContext,
        encryptDecryptSession.sessionHandle );
CheckPassed( rval );

rval = EndAuthSession( &encryptDecryptSession );
CheckPassed( rval );
    }
```

Delete the NV index.

```
    // Set authorization for NV undefine command.
    nvUndefineAuth.sessionHandle = TPM_RS_PW;
    nvUndefineAuth.nonce.t.size = 0;
    *( (UINT8 *)((void *)&nvUndefineAuth.sessionAttributes ) ) = 0;
    nvUndefineAuth.hmac.t.size = 0;

    // Undefine NV index.
    rval = Tss2_Sys_NV_UndefineSpace( sysContext,
            TPM_RH_PLATFORM, TPM20_INDEX_TEST1, &nvUndefineAuths, 0 );
    CheckPassed( rval );
}
```

Summary

As you can see, there is a fair amount of work involved in using decrypt and encrypt sessions. Abstracting this work into well-designed functions or even using a higher-level API such as the Feature API helps to reduce this work.

Decrypt and encrypt sessions provide secrecy for sensitive information while in transit to and from the TPM. You've seen what they do, some use cases, and how to program them using the TSS System API, and you've learned about some new functionality in the System API.

The next chapter focuses on TPM context management.