ERRATA

ERRATA

Errata Version 1.9
August 23, 2019

FOR

TCG Trusted Platform Module Library

Family “2.0”
Level 00 Revision 1.38
September 29, 2016

Contact: admin@trustedcomputinggroup.org

TCG Published

Disclaimers, Notices, and License Terms

THIS ERRATA IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Contact the Trusted Computing Group at www.trustedcomputinggroup.org for information on specification licensing through membership agreements.

CONTENTS

1. Introduction

This document describes errata and clarifications for the TCG Trusted Platform Module Library Version 2.0 Revision 1.38 as published. The information in this document is likely – but not certain – to be incorporated into a future version of the specification. Suggested fixes proposed in this document may be modified before being published in a later TCG Specification. Therefore, the contents of this document are not normative and only become normative when included in an updated version of the published specification. Note that since the errata in this document are non-normative, the patent licensing rights granted by Section 16.4 of the Bylaws do not apply.

2. Errata

2.1 Object Derivation

2.1.1 Introduction

This section summarizes errata with regards to Object Derivation in TPM2_CreateLoaded(). For interoperability of Derived Objects, it is essential that all parties, given the same Derivation Parent and the same Derivation Parameters, derive the same key. Therefore, external software that uses the Library Spec reference code to implement Object Derivation outside of the TPM needs to consider the code fixes in this section as well.

2.1.2 Code Change Highlighting

The code fixes are highlighted using a color scheme that is specific to this errata document and should help the reader to apply the necessary changes to the reference code. The meaning of the highlighting is explained using the below example (which is copied from 2.1.4.1). The highlighting is following the standard convention of a “diff-compare” using e.g. a version control tool.

EXAMPLE

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 73

```
         DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
                                   scheme->details.xor.hashAlg,
                                   scheme->details.xor.kdf,
-                                  &parent->sensitive.seedValue.b,
+                                  &parent->sensitive.sensitive.bits.b,
                                   &publicArea->unique.derive.label.b,
                                   &publicArea->unique.derive.context.b);
```

The text before the code fix references the Part, clause, and line number in the Library Spec where this code is specified. A code line that should be removed is highlighted in red font and is preceded with a minus (“-”) sign. A code line that should be added is highlighted in green font and is preceded with a plus (“+”) sign. If only a particular part of the code line needs to be replaced, the part to be removed is shaded in red and the part to be added is shaded in green. With the changes applied to the above example, the resulting code would be:

```
         DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
                                   scheme->details.xor.hashAlg,
                                   scheme->details.xor.kdf,
                                   &parent->sensitive.sensitive.bits.b,
                                   &publicArea->unique.derive.label.b,
                                   &publicArea->unique.derive.context.b);
```

2.1.3 Derivation of *sensitive* and *seedValue*

The following section is provided as clarification to Part 1, 28 Object Derivation.

Derived Objects are generated using the key derivation function specified in Part 1, 11.4.9.2 KDFa(). One parameter of the KDFa() is a length value (“L”) that specifies the maximum bit length of the keying material that can be generated by the KDF. For Object derivation this length parameter is set to a constant value of 8192 which indicates that the KDF can generate a maximum of 8k bits (1k bytes). However, the TPM only needs to generate as many bytes from the KDF as necessary for the key derivation. Usually this will be less than the maximum size.

NOTE 1 Only during FIPS CAVP testing, the TPM would need to generate the full 1k bytes.

NOTE 2 In the future, the length parameter (“L”) might need to support a larger size to generate quantum resistant Derived Keys. Conventional keys would continue to be generated with a length value of 8192 to maintain interoperability.

The keying material output by the KDF is used in the following way to generate the *sensitive* and *seedValue* fields of a Derived Object.

1) Generation of *sensitive*

- For ECC Derived Keys the method of FIPS 186-4, Annex B.4.1 *Key Pair Generation Using Extra Random Bits* is used. The first N + 8 bytes (where N is the size of *sensitive*) of the keying material are used as starting value (the “c” in FIPS 186-4, B.4.1) to compute the ECC private key. The method using extra random bits will always generate a valid ECC key. This ensures that no key needs to be regenerated.
- For symmetric Derived Keys, the first N bytes (where N is the size of *sensitive*) of the keying material are used as *sensitive*.

2) Generation of *seedValue*

- The following N bytes (where N is the size of *seedValue*) of the keying material are used as *seedValue*.

Consecutive bytes from the keying material are used for *sensitive* and *seedValue*. No bytes are skipped or reused. The only type of Derived Key that needs to be regenerated are 3DES keys. If the *sensitive* value of a 3DES Derived Key results in a prohibited key value, a new *sensitive* value is generated. The *sensitive* value is generated before *seedValue* is generated.

2.1.4 Incorrect KDF Seed

The reference code in Part 3, 12.9 TPM2_CreateLoaded uses an incorrect key in the key derivation function (KDF) to generate a Derived Object. The reference code uses the Derivation Parent’s *seedValue* instead of the Derivation Parent’s *sensitive* value. This affects the key generation of all types of Derived Objects (TPM_ALG_SYMCIPHER, TPM_ALG_KEYEDHASH, and TPM_ALG_ECC).

This issue is caused by an incorrect parameter in the function call to DRBG_InstantiateSeededKdf(). To fix this, the seed used for the KDF should be replaced with the *sensitive* value (see code fix in 2.1.4.1). The correct KDF parameters for Object derivation are specified in Part 1, 28.4 Entropy for Derived Objects.

2.1.4.1 Code Fix

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 73

```
         DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
                                   scheme->details.xor.hashAlg,
                                   scheme->details.xor.kdf,
-                                  &parent->sensitive.seedValue.b,
+                                  &parent->sensitive.sensitive.bits.b,
                                   &publicArea->unique.derive.label.b,
                                   &publicArea->unique.derive.context.b);
```

2.1.5 Incorrect Label and Context Storage

The reference code in Part 3, 12.9 TPM2_CreateLoaded does not correctly include *label* and *context* in the key derivation function (KDF) when a Derived Object of the type TPM_ALG_ECC is generated.

The reference code reuses the *unique* field in the public area of the object to store the *label* and *context* parameters that are provided by the caller. However, the *unique* field is also used during the key generation to output the ECC public key. As a result, the *label* and *context* values are overwritten and incorrect parameters are used in the derivation of the *sensitive* value and *seedValue*. To fix this, a separate structure variable needs to be allocated to store *context* and *label* (see code fixes in 2.1.5.1).

2.1.5.1 Code Fix

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 16

```
     TPMT_PUBLIC                 *publicArea;
     RAND_STATE                   randState;
     RAND_STATE                  *rand = &randState;
+    TPMS_DERIVE                  labelContext;
// Input Validation
```

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 38

```
     // unmarshaled like other public areas. Since it is not, this command needs its
     // own template that is a TPM2B that is unmarshaled as a BYTE array with a
     // its own unmarshal function.
-    result = UnmarshalToPublic(publicArea, &in->inPublic, derivation);
+    result = UnmarshalToPublic(publicArea, &in->inPublic, derivation,
+                               &labelContext);
     if(result != TPM_RC_SUCCESS)
         return result + RC_CreateLoaded_inPublic;
```

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 66

```
              return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
         // Process the template and sensitive areas to get the actual 'label' and
         // 'context' values to be used for this derivation.
-        result = SetLabelAndContext(publicArea, &in->inSensitive.sensitive.data);
+        result = SetLabelAndContext(&labelContext, &in->inSensitive.sensitive.data);
         if(result != TPM_RC_SUCCESS)
             return result;
         // Set up the KDF for object generation
```

Part 3, 12.9.3 Detailed Actions (of TPM2_CreateLoaded), line 73

```
         DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
                                   scheme->details.xor.hashAlg,
                                   scheme->details.xor.kdf,
                                   &parent->sensitive.sensitive.bits.b,
-                                  &publicArea->unique.derive.label.b,
-                                  &publicArea->unique.derive.context.b);
+                                  &labelContext.label.b,
+                                  &labelContext.context.b);
         // Clear the sensitive size so that the creation functions will not try
         // to use this value.
         in->inSensitive.sensitive.data.t.size = 0;
```

Part 4, 7.6.3.18 SetLabelAndContext(), line 1070

```
 TPM_RC
 SetLabelAndContext(
-    TPMT_PUBLIC            *publicArea,    // IN/OUT: the public area containing
-                                           //         the unmarshaled template
+    TPMS_DERIVE            *labelContext,  // IN/OUT: the recovered label and
+                                           //         context
     TPM2B_SENSITIVE_DATA   *sensitive      // IN: the sensitive data
     )
 {
+    TPMS_DERIVE             sensitiveValue;
     TPM_RC                  result;
     INT32                   size;
     BYTE                   *buff;
-    TPM2B_LABEL             label;
-
+//
     // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
-    size = sensitive->t.size;
     // If there is something to unmarshal...
-    if(size != 0)
+    if(sensitive->t.size != 0)
     {
+        size = sensitive->t.size;
         buff = sensitive->t.buffer;
-        result = TPM2B_LABEL_Unmarshal(&label, &buff, &size);
-        if(result != TPM_RC_SUCCESS)
-            return result;
-        // If there is a label in the publicArea, it overrides
-        if(publicArea->unique.derive.label.t.size == 0)
-            MemoryCopy2B(&publicArea->unique.derive.label.b, &label.b,
-                         sizeof(publicArea->unique.derive.label.t.buffer));
-        result = TPM2B_LABEL_Unmarshal(&label, &buff, &size);
+        result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
         if(result != TPM_RC_SUCCESS)
             return result;
-        if(publicArea->unique.derive.context.t.size == 0)
-            MemoryCopy2B(&publicArea->unique.derive.context.b, &label.b,
-                         sizeof(publicArea->unique.derive.context.t.buffer));
+        // If there was a label in the public area leave it there, otherwise, copy
+        // the new value
+        if(labelContext->label.t.size == 0)
+            MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
+                         sizeof(labelContext->label.t.buffer));
+        // if there was a context string in publicArea, it overrides
+        if(labelContext->context.t.size == 0)
+            MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
+                         sizeof(labelContext->label.t.buffer));
     }
     return TPM_RC_SUCCESS;
 }
```

Part 4, 7.6.3.19 UnmarshalToPublic(), line 1104

```
 UnmarshalToPublic(
     TPMT_PUBLIC         *tOut,       // OUT: output
     TPM2B_TEMPLATE      *tIn,        // IN:
-    BOOL                 derivation  // IN: indicates if this is for a derivation
```

```
+    BOOL                    derivation, // IN: indicates if this is for a derivation
+    TPMS_DERIVE            *labelContext// OUT: label and context if derivation
     )
 {
```

Part 4, 7.6.3.19 UnmarshalToPublic(), line 1114

```
     // make sure that tOut is zeroed so that there are no remnants from previous
     // uses
     MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
-    // Unmarshal a TPMT_PUBLIC but don't allow a nameAlg of TPM_ALG_NULL
-    result = TPMT_PUBLIC_Unmarshal(tOut, &buffer, &size, FALSE);
-    if((result == TPM_RC_SUCCESS) && (derivation == TRUE))
-    {
-#if ALG_ECC
-        // If we just unmarshaled an ECC public key, then the label value is in the
-        // correct spot but the context value is in the wrong place if the
-        // maximum ECC parameter size is larger than 32 bytes. So, move it.
-        if(tOut->type == ALG_ECC_VALUE)
-        {
-            // This could probably be a direct copy because we are moving data
-            // to lower addresses but, just to be safe...
-            TPM2B_LABEL      context;
-            MemoryCopy2B(&context.b, &tOut->unique.ecc.y.b,
-                         sizeof(context.t.buffer));
-            MemoryCopy2B(&tOut->unique.derive.context.b, &context.b,
-                         sizeof(tOut->unique.derive.context.t.buffer));
-        }
-        else
-#endif
-            // For object types other than ECC, should have completed unmarshaling
-            // with data left in the buffer so try to unmarshal the remainder as a
-            // TPM2B_LABEL into the context
-            result = TPM2B_LABEL_Unmarshal(&tOut->unique.derive.context,
-                                                      &buffer, &size);
-    }
+    // Unmarshal the components of the TPMT_PUBLIC up to the unique field
+    result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
+    if(result != TPM_RC_SUCCESS)
+        return result;
+    result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
+    if(result != TPM_RC_SUCCESS)
+        return result;
+    result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
+    if(result != TPM_RC_SUCCESS)
+        return result;
+    result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
+    if(result != TPM_RC_SUCCESS)
+        return result;
+    result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
+                                         tOut->type);
+    if(result != TPM_RC_SUCCESS)
+        return result;
+    // Now unmarshal a TPMS_DERIVE if this is for derivation
+    if(derivation)
+        result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
+    else
+        // otherwise, unmarshal a TPMU_PUBLIC_ID
+        result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
+                                          tOut->type);
+    // Make sure the template was used up
+    if((result == TPM_RC_SUCCESS) && (size != 0))
+        result = TPM_RC_SIZE;
     return result;
 }
```

2.1.6 Incorrect Label and Context Size

In the reference code, the maximum size of *label* and *context* (LABEL_MAX_BUFFER) is not defined in compliance with the size requirement in Part 2.

Part 2, 11.1.10 TPM2B_LABEL specifies that, “For interoperability and backwards compatibility, LABEL_MAX_BUFFER is the minimum of the largest digest on the device and the largest ECC parameter (MAX_ECC_KEY_BYTES) but no more than 32 bytes.”

The definition of LABEL_MAX_BUFFER should be fixed in the reference code (see code fix in 2.1.6.1).

2.1.6.1 Code Fix

Part 4, 5.12.5 Compile-time Checks (of GpMacros.h), line 126

```
-#define LABEL_MAX_BUFFER   MIN(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE)
-#if LABEL_MAX_BUFFER < 32
-#error "The size allowed for the label is not large enough for interoperability."
+// This is updated to follow the requirement of P2 that the label not be larger
+// than 32 bytes.
+#ifndef LABEL_MAX_BUFFER
+#define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
#endif
```

2.1.7 Incorrect Byte Order

When the reference code creates a Derived Object using TPM2_CreateLoaded(), the byte order of the generated *sensitive* value and *seedValue* of the object is processor dependent. With the same Derivation Parent and the same derivation parameters, a different Derived Object is generated on a big endian and little endian TPM. This affects the key generation of all types of Derived Objects (TPM_ALG_SYMCIPHER, TPM_ALG_KEYEDHASH, and TPM_ALG_ECC).

The reference code generates the random bits that are used as secret (ECC private key or symmetric key) of the Derived Object in an internal format (bigNum). When later converted to canonical form (TPM2B), the byte order changes dependent on the endianness of the TPM. To fix this, the random bits in BnGetRandomBits() should be generated in canonical form (TPM2B) and then converted to internal format for processing (see code fix in 2.1.7.1).

2.1.7.1 Code Fix

Part 4, 10.2.4.3.20 BnGetRandomBits(), line 353

```
 RAND_STATE      *rand
     )
 {
-    n->size = BITS_TO_CRYPT_WORDS(bits);
-    if(n->size > n->allocated)
-        n->size = n->allocated;
-    DRBG_Generate(rand, (BYTE *)n->d, (UINT16)(n->size * RADIX_BYTES));
+    TPM2B_TYPE(LARGEST, LARGEST_NUMBER);
+    TPM2B_LARGEST   large;
+    large.b.size = (UINT16)BITS_TO_BYTES(bits);
+    DRBG_Generate(rand, large.t.buffer, large.t.size);
+    BnFrom2B(n, &large.b);
     BnMaskBits(n, bits);
     return TRUE;
 }
```

2.1.8 Derivation Parameters

Part 1, 28.2 Derivation Parameters contains an incorrect statement which says, “If (*label* or *context* is) provided in the *unique* field, the corresponding value in the *inPrivate.data* field is required to be an empty buffer.”

It should say, “If provided in the *unique* field, the corresponding value in the *inSensitive.data* field is ignored.”

2.1.9 FIPS Compliance

2.1.9.1 KDF Counter Initialization

In the reference code, the counter value for the KDF instance used for the Derivation of Derived Objects should be initialized to zero instead of one as the counter is incremented before the KDF call. This fix ensures that the KDF starts with a counter of 1 which is in alignment with SP800-108.

2.1.9.2 KDF Length Parameter

In the reference code, the length parameter (“L”) used in the KDFa() is set incorrectly when a Derived Object’s *sensitive* and *seedValue* are generated. The length is set to the size of *sensitive* when the *sensitive* value is derived and set to the size of *seedValue* when the *seedValue* is derived.

According to SP800-108, the KDF length parameter is defined as the maximum length of the keying material that can be output from the KDF. Therefore, the length should be set to a constant value of 8k bits. This ensures that it will always be larger than the sum of *sensitive* and *seedValue*.

2.1.9.3 ECC Key Generation Method

The reference code generates ECC Derived Keys using the method of FIPS 186-4, Annex B.4.2 *Key Pair Generation by Testing Candidates* as described in Part 1, C.5 ECC Key Generation.

To follow the guidance from NIST, ECC Derived Keys should be generated using the method of FIPS 186-4, Annex B.4.1 *Key Pair Generation Using Extra Random Bits*. Therefore the reference code needs to be fixed.

2.1.9.4 Check for Leading Zeros

The reference code regenerates keys in the case of too many leading zeros. When the TPM generates the *sensitive* value for a KEYEDHASH or SYMCIPHER object, or the *seedValue* for any type of object, it verifies that the actual bit size of the generated key is at least half the requested bit size. If not, the reference code regenerates the key. This is done by the function CryptRandMinMax().

This check for leading zeros is unnecessary and should be removed as it complicates the generation of Derived Objects.

2.1.9.5 Code Fixes

This section summarizes the code fixes for the issues described in 2.1.9.1 to 2.1.9.4.

Part 4, 10.1.5 CryptRand.h, line 73

```
 {
     UINT64               counter;
     UINT32               magic;
+    UINT32               limit;
     TPM2B               *seed;
     const TPM2B         *label;
     TPM2B               *context;
     TPM_ALG_ID           hash;
     TPM_ALG_ID           kdf;
+    UINT16               digestSize;
+    TPM2B_DIGEST         residual;
 } KDF_STATE, *pKDR_STATE;
 #define KDF_MAGIC    ((UINT32) 0x4048444a) // "KDF " backwards
```

Part 4, A.2 Implementation.h, line 270

```
 #define  CRT_FORMAT_RSA                YES
 #define  VENDOR_COMMAND_COUNT          0
 #define  MAX_VENDOR_BUFFER_SIZE        1024
+#define  TPM_MAX_DERIVATION_BITS       8192


 // Table 1:2 - Definition of TPM_ALG_ID Constants  (TPM_ALG_ID_Processing)
```

Part 3, 12.9 Detailed Actions (of TPM2_CreateLoaded), line 75

```
                                   scheme->details.xor.kdf,
                                   &parent->sensitive.sensitive.bits.b,
                                   &labelContext.label.b,
-                                  &labelContext.context.b);
+                                  &labelContext.context.b,
+                                  TPM_MAX_DERIVATION_BITS);
         // Clear the sensitive size so that the creation functions will not try
         // to use this value.
         in->inSensitive.sensitive.data.t.size = 0;
```

Part 4, 10.2.4.3.20 BnGetRandomBits(), line 353 (this fix is applied on top of fix 2.1.7.1)

```
     RAND_STATE      *rand
     )
 {
-    TPM2B_TYPE(LARGEST, LARGEST_NUMBER);
-    TPM2B_LARGEST   large;
+    // Since this could be used for ECC key generation using the extra bits method,
+    // make sure that the value is large enough
+    TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
+    TPM2B_LARGEST    large;
+    //
     large.b.size = (UINT16)BITS_TO_BYTES(bits);
-    DRBG_Generate(rand, large.t.buffer, large.t.size);
-    BnFrom2B(n, &large.b);
-    BnMaskBits(n, bits);
-    return TRUE;
+    if (DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
+    {
+        if (BnFrom2B(n, &large.b) != NULL)
+        {
+            if (BnMaskBits(n, bits))
+                return TRUE;
+        }
+    }
+    return FALSE;
```

```
 }

 //*** BnGenerateRandomInRange()
```

Part 4, 10.2.9.2.4 CryptGenerateKeyDes(), line 89

```
         BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
         int                      i = (sensitive->sensitive.sym.t.size + 7) / 8;
 // Use the random number generator to generate the required number of bits
-        DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size);
+        if (DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
+            return TPM_RC_NO_RESULT;
         for(; i > 0; pK += 8, i--)
         {
             UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
```

Part 4, 10.2.11.2.20, BnEccGetPrivate(), line 405

```
     RAND_STATE              *rand       // IN: state for DRBG
     )
 {
-//
     bigConst                 order = CurveGetOrder(C);
-//
+    BOOL                     OK;
+    UINT32                   orderBits = BnSizeInBits(order);
+#if 1 // This is the "extra bits" method of key generation
+    UINT32                   orderBytes = BITS_TO_BYTES(orderBits);
+    BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
+    BN_VAR(nMinus1, MAX_ECC_KEY_BITS);
+    //
+    OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
+    OK = OK && BnSubWord(nMinus1, order, 1);
+    OK = OK && BnMod(bnExtraBits, nMinus1);
+    OK = OK && BnAddWord(dOut, bnExtraBits, 1);
+#else
+    // This is the "testing candidates" version of key generation
     do
     {
-        BnGetRandomBits(dOut, BnSizeInBits(order), rand);
-        BnAddWord(dOut, dOut, 1);
-    } while(BnUnsignedCmp(dOut, order) >= 0);
-    return TRUE;
+        OK = BnGetRandomBits(dOut, BnSizeInBits(order), rand);
+        OK = OK && BnAddWord(dOut, dOut, 1);
+    } while (OK && BnUnsignedCmp(dOut, order) >= 0);
+#endif
+    return OK;
 }
```

Part 4, 10.2.11.2.20, BnEccGetPrivate(), line 418

```
     )
 {
     BOOL                 OK = FALSE;
-    int                  limit;
-    for(limit = 100; (limit > 0) && !OK; limit--)
-    {
-        // Get a private scalar
-        BnEccGetPrivate(bnD, E->C, rand);
-        // Do a point multiply
-        OK = BnEccModMult(ecQ, NULL, bnD, E);
-    }
+    // Get a private scalar
+    OK = BnEccGetPrivate(bnD, E->C, rand);
+
+    // Do a point multiply
+    OK = OK && BnEccModMult(ecQ, NULL, bnD, E);
     if(!OK)
         BnSetWord(ecQ->z, 0);
     else
```

Part 4, 10.2.11.2.24 CryptEccGenerateKey()

The following error code should be added to the return code table of this function.

Error Returns	Meaning
TPM_RC_NO_RESULT	could not verify key with signature (FIPS only)

Part 4, 10.2.11.2.24 CryptEccGenerateKey(), line 524

```
     CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
     ECC_NUM(bnD);
     POINT(ecQ);
-    const UINT32             MaxCount = 100;
-    UINT32                   count = 0;
-    TPM_RC                   retVal = TPM_RC_NO_RESULT;
+    BOOL                     OK;
+    TPM_RC                   retVal;

     TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
```

Part 4, 10.2.11.2.24 CryptEccGenerateKey(), line 535

```
     publicArea->unique.ecc.y.t.size = 0;
     sensitive->sensitive.ecc.t.size = 0;

-    // Start search for key (should be quick)
-    for(count = 1; (count < MaxCount) && (retVal != TPM_RC_SUCCESS); count++)
+    OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
+    if (OK)
     {
-        if(!BnEccGenerateKeyPair(bnD, ecQ, E, rand))
-            FAIL(FATAL_ERROR_INTERNAL);
-        retVal = TPM_RC_SUCCESS;
-#ifdef FIPS_COMPLIANT
-        // See if PWCT is required
-        if(publicArea->objectAttributes.sign)
-        {
-            ECC_NUM(bnT);
-            ECC_NUM(bnS);
-            TPM2B_DIGEST    digest;
-            TEST(TPM_ALG_ECDSA);
-            digest.t.size =
-                (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetPrime(
-                    AccessCurveData(E))));
-
-            // Get a random value to sign using the current DRBG state
-            DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
-            BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
-
-            // and make sure that we can validate the signature
-            retVal = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest);
-        }
-#endif
+        BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
+        BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
     }
-// if counter maxed out, put the TPM into failure mode
-    if(count == MaxCount)
-        FAIL(FATAL_ERROR_INTERNAL);
-// Convert results
-    BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
-    BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
+#if defined FIPS_COMPLIANT || 1
+    // See if PWCT is required
+    if (OK && publicArea->objectAttributes.sign)
+    {
+        ECC_NUM(bnT);
+        ECC_NUM(bnS);
+        TPM2B_DIGEST    digest;
+        TEST(TPM_ALG_ECDSA);
+        digest.t.size =
+            (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetPrime(
+                AccessCurveData(E))));
+        // Get a random value to sign using the built in DRBG state
+        DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
+        BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
+        // and make sure that we can validate the signature
+        OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
+    }
+#endif
+    retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
 Exit:
    CURVE_FREE(E);
    return retVal;
```

Part 4, 10.2.12.3.2 BnSignEcdaa()

The following error code should be added to the return code table of this function.

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get values from random number generator

Part 4, 10.2.12.3.2 BnSignEcdaa(), line 157

```
         {
             // generate nonceK such that 0 < nonceK < n
             // use bnT as a temp.
-            BnEccGetPrivate(bnT, AccessCurveData(E), rand);
+            if (!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
+           {
+               retVal = TPM_RC_NO_RESULT;
+               break;
+           }
            BnTo2B(bnT, &nonceK->b, 0);

            T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
```

Part 4, 10.2.13.8.2 CryptKDFa(), line 504

```
     UINT32          *counterInOut,  // IN/OUT: caller may provide the iteration
                                     //     counter for incremental operations to
                                     //     avoid large intermediate buffers.
-    BOOL             once           // IN: TRUE - only 1 iteration is performed
-                                    //     FALSE if iteration count determined by
-                                    //     "sizeInBits"
+    UINT16           blocks         // IN: If non-zero, this is the maximum number
+                                    //     of blocks to be returned, regardless
+                                    //     of sizeInBit
     )
 {
     UINT32                   counter = 0;    // counter value
     INT16                    bytes;          // number of bytes to produce
+    UINT16                   generated;      // number of bytes generated
     BYTE                    *stream = keyStream;
     HMAC_STATE               hState;
     UINT16                   digestSize = CryptHashGetDigestSize(hashAlg);

     pAssert(key != NULL && keyStream != NULL);
-    pAssert(once == FALSE || (sizeInBits & 7) == 0);

     if(digestSize == 0)
         return 0;
```

Part 4, 10.2.13.8.2 CryptKDFa(), line 525

```
     // it is a fatal error.
     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);

-    bytes = once ? digestSize : (INT16)((sizeInBits + 7) / 8);
+    // The number of bytes to be generated is the smaller of the sizeInBits bytes or
+    // the number of requested blocks. The number of blocks is the smaller of the
+    // number requested or the number allowed by sizeInBits. A partial block is
+    // a full block.
+    bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
+    generated = bytes;

     // Generate required bytes
     for(; bytes > 0; bytes -= digestSize)
     {
         counter++;
-        if(bytes < digestSize)
-            digestSize = bytes;
-
         // Start HMAC
         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
             return 0;
```

Part 4, 10.2.13.8.2 CryptKDFa(), line 556

```
         // Adding size in bits
         CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);

-        CryptHmacEnd(&hState, digestSize, stream);
+        // Complete and put the data in the buffer
+        CryptHmacEnd(&hState, bytes, stream);
         stream = &stream[digestSize];
     }
-    // Mask off bits if the required bits is not a multiple of byte size
-    if((sizeInBits % 8) != 0)
+    // Mask off bits if the required bits is not a multiple of byte size. Only do
+    // this if this is a call that is returning all the blocks indicated in
+    // sizeInBits
+#if 0 //?? Masking in the KDF is disabled. If the calling function wants something
+      //?? less than even number of bytes, then the caller should do the masking
+      //?? because there is no universal way to do it here
+    if((blocks == 0) && (sizeInBits % 8) != 0)
         keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
+#endif
     if(counterInOut != NULL)
         *counterInOut = counter;
-    return (UINT16)((sizeInBits + 7) / 8);
+    return generated;
 }

 //*** CryptKDFe()
```

Part 4, 10.2.17.4.4 DRBG_InstantiateSeededKdf(), line 416

```
     TPM_ALG_ID        kdf,           // IN: the KDF to use
     TPM2B            *seed,          // IN: the seed to use
     const TPM2B      *label,         // IN: a label for the generation process.
-    TPM2B            *context        // IN: the context value
+    TPM2B            *context,       // IN: the context value
+    UINT32            limit          // IN: Maximum number of bits from the KDF
     )
 {
     state->magic = KDF_MAGIC;
+    state->limit = limit;
     state->seed = seed;
     state->hash = hashAlg;
     state->kdf = kdf;
     state->label = label;
     state->context = context;
+    state->digestSize = CryptHashGetDigestSize(hashAlg);
-    state->counter = 1;
+    state->counter = 0;
+    state->residual.t.size = 0;
     return TRUE;
 }
```

Part 4, 10.2.17.4.9 DRBG_Generate(), line 511

```
     UINT16            randomSize     // IN: the number of bytes to generate
     )
```

```
 {
-//
     if(state == NULL)
         state = (RAND_STATE *)&drbgDefault;

-    // If the caller used a KDF state, generate a sequence from the KDF
+    // If the caller used a KDF state, generate a sequence from the KDF not to
+    // exceed the limit.
     if(state->kdf.magic == KDF_MAGIC)
     {
         KDF_STATE       *kdf = (KDF_STATE *)state;
-        UINT32           count = (UINT32)kdf->counter;
-        if((randomSize != 0) && (random != NULL))
-            CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
-                      randomSize * 8, random, &count, 0);
-        kdf->counter = count;
+        UINT32           counter = (UINT32)kdf->counter;
+        INT32            bytesLeft = randomSize;
+
+        if(random == NULL)
+            return 0;
+        // If the number of bytes to be returned would put the generator
+        // over the limit, then return 0
+        if((((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
+            return 0;
+        // Process partial and full blocks until all requested bytes provided
+        while(bytesLeft > 0)
+        {
+            // If there is any residual data in the buffer, copy it to the output
+            // buffer
+            if(kdf->residual.t.size > 0)
+            {
+                INT32      size;
+//
+                // Don't use more of the residual than will fit or more than are
+                // available
+                size = MIN(kdf->residual.t.size, bytesLeft);
+
+                // Copy some or all of the residual to the output. The residual is
+                // at the end of the buffer. The residual might be a full buffer.
+                MemoryCopy(random,
+                           &kdf->residual.t.buffer
+                           [kdf->digestSize - kdf->residual.t.size], size);
+
+                // Advance the buffer pointer
+                random += size;
+
+                // Reduce the number of bytes left to get
+                bytesLeft -= size;
+
+                // And reduce the residual size appropriately
+                kdf->residual.t.size -= (UINT16)size;
+            }
+            else
+            {
+                UINT16           blocks = (UINT16)(bytesLeft / kdf->digestSize);
+//
+                // Get the number of required full blocks
+                if(blocks > 0)
+                {
+                    UINT16      size = blocks * kdf->digestSize;
+// Get some number of full blocks and put them in the return buffer
+                    CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
+                              kdf->limit, random, &counter, blocks);
+
+                    // reduce the size remaining to be moved and advance the pointer
+                    bytesLeft -= size;
+                    random += size;
+                }
+                else
+                {
+                    // Fill the residual buffer with a full block and then loop to
+                    // top to get part of it copied to the output.
+                    kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
+                                                     kdf->label, kdf->context, NULL,
+                                                     kdf->limit,
+                                                     kdf->residual.t.buffer,
+                                                     &counter, 1);
+                }
+            }
+        }
+        kdf->counter = counter;
         return randomSize;
     }
     else if(state->drbg.magic == DRBG_MAGIC)
```

Part 4, 10.2.17.4.12 CryptRandMinMax(), line 610

```
+#if 0
 CryptRandMinMax(
     BYTE            *out,
     UINT32           max,
```

Part 4, 10.2.17.4.12 CryptRandMinMax(), line 623

```
     } while(BnSizeInBits(bn) < min);
     BnToBytes(bn, out, &size);
     return size;
+#endif
```

Part 4, 10.2.6.3.3 CryptGenerateKeyedHash()

The following error code should be added to the return code table of this function.

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get values from random number generator

Part 4, 10.2.6.3.3 CryptGenerateKeyedHash(), line 98

```
     }
     else
     {
-        // If the TPM is going to generate the data, then set the size to be the
+        // The TPM is going to generate the data so set the size to be the
```

```
         // size of the digest of the algorithm
-        int                         sizeInBits = digestSize * 8;
-        TPM2B_SENSITIVE_DATA       *key = &sensitive->sensitive.bits;
-        key->t.size = CryptRandMinMax(key->t.buffer, sizeInBits, sizeInBits / 2,
-                                      rand);
+        sensitive->sensitive.bits.t.size =
+            DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
+        if (sensitive->sensitive.bits.t.size == 0)
+            return TPM_RC_NO_RESULT;
     }
     return TPM_RC_SUCCESS;
 }
```

Part 4, 10.2.6.4.3 CryptGenerateKeySymmetric()

The following error code should be added to the return code table of this function.

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get a random value

Part 4, 10.2.6.4.3 CryptGenerateKeySymmetric(), line 204

```
 #ifdef TPM_ALG_TDES
     else if(publicArea->parameters.symDetail.sym.algorithm == TPM_ALG_TDES)
     {
-        sensitive->sensitive.sym.t.size = keyBits / 8;
         result = CryptGenerateKeyDes(publicArea, sensitive, rand);
     }
 #endif
     else
     {
-        sensitive->sensitive.sym.t.size = CryptRandMinMax(
-                    sensitive->sensitive.sym.t.buffer, keyBits, keyBits / 2, rand);
-        result = TPM_RC_SUCCESS;
+        sensitive->sensitive.sym.t.size =
+            DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
+                BITS_TO_BYTES(keyBits));
+        result = (sensitive->sensitive.sym.t.size == 0)
+            ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
     }
     return result;
 }
```

Part 4, 10.2.6.6.8 CryptCreateObject()

The following error code should be added to the return code table of this function.

Error Returns	Meaning
TPM_RC_NO_RESULT	unable to get random values (only in derivation)

Part 4, 10.2.6.6.8 CryptCreateObject(), line 794

```
     if(object->attributes.primary && object->attributes.epsHierarchy)
         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);

-    // Set the seed value to the size of the digest produced by the nameAlg
-    object->sensitive.seedValue.b.size
-        = CryptHashGetDigestSize(publicArea->nameAlg);
-    object->sensitive.seedValue.t.size = CryptRandMinMax(
-                object->sensitive.seedValue.t.buffer,
-                object->sensitive.seedValue.t.size * 8,
-                object->sensitive.seedValue.t.size * 8 / 2, rand);
+    // Generate a seedValue that is the size of the digest produced by nameAlg
+    object->sensitive.seedValue.t.size =
+        DRBG_Generate(rand, object->sensitive.seedValue.t.buffer,
+            CryptHashGetDigestSize(publicArea->nameAlg));
+    if (object->sensitive.seedValue.t.size == 0)
+        return TPM_RC_NO_RESULT;

     // For symmetric values, need to compute the unique value
     if(publicArea->type == TPM_ALG_SYMCIPHER
```

2.2 Attribute Check for KEYEDHASH Objects

It is recommended to add the following attribute check to the reference code in Part 4, 7.6.3.3 CreateChecks().

When a *restricted decrypt* or *restricted sign* TPM_ALG_KEYEDHASH Object is created with *sensitiveDataOrigin* CLEAR (i.e. the sensitive data is provided by the caller), then *fixedParent* and *fixedTPM* are required to be CLEAR, otherwise the TPM will return TPM_RC_ATTRIBUTES.

This attribute check is implemented in the reference code for TPM_ALG_SYMCIPHER Objects, but is missing for TPM_ALG_KEYEDHASH Objects.

2.3 Attribute Check in TPM2_CreatePrimary

The following attribute check is missing in the reference code in Part 3, 24.1 TPM2_CreatePrimary.

When a TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER Object is created using TPM2_CreatePrimary with *sensitiveDataOrigin* CLEAR (i.e. the sensitive data is provided by the caller), then *sensitive.data* must be not empty, otherwise the TPM will return TPM_RC_ATTRIBUTES.

2.4 TPM2_ECC_Parameters

Part 1, C.8 ECC Point Padding contains an inaccurate statement which says, "When the ECC parameters are returned by the command TPM2_ECC_Parameters(), they have to match the exact format as specified in the TCG Algorithm registry."

Only the numerical values of the ECC curve parameters returned by TPM2_ECC_Parameters() must be the same as listed in the TCG Algorithm Registry. The size may not be the same.

An ECC parameter with a numerical value of zero is incorrectly returned by the reference code as Empty Buffer. It should be returned as a sized buffer with only the data value set to zero.

2.5 TPM2_DictionaryAttackParameters - *failedTries*

According to the description and reference code in Part 3, 25.3, TPM2_DictionaryAttackParameters will set the authorization failure count (*failedTries*) to zero.

This is incorrect. TPM2_DictionaryAttackParameters must not set the authorization failure count (*failedTries*) to zero but leave *failedTries* unmodified. As a result, the TPM2_DictionaryAttackParameters() command may cause the TPM to enter lockout. If *maxTries* is changed to a value that is less than the current value of *failedTries*, the TPM goes into lockout until *failedTries* is less than *maxTries*.

In order to avoid accidental lockout when setting new Dictionary Attack parameters, it is recommended to read the current value of *failedTries* with TPM2_GetCapabilty (capability = TPM_CAP_TPM_PROPERTIES, property = TPM_PT_LOCKOUT_COUNTER), and if necessary, use TPM2_DictionaryAttackLockReset() to reset the authorization failure count before setting the new DA parameters.

EXAMPLE For this example, (m, n) is used as notation for (*maxTries, recoveryTime* in minutes). If the parameters are (32, 120) and *failedTries* is 30, and the parameters are changed to (10, 10), then the TPM will be in lockout until *failedTries* counts down to 9 at one count per each 10 minutes elapsed since the moment of the last failed authorization attempt (the one that brought *failedTries* to 30). In this example it may take from 91 to 210 minutes depending on how much time had elapsed within original *recoveryTime* interval by the moment when the parameters were changed (with the possible range being from 0 to 119 minutes).

2.6 Self-healing

According to Part 1, 19.8.2 Lockout Mode Configuration Parameters, paragraph a); 2), *failedTries* is decremented by one after *recoveryTime* seconds if there is no power interruption. This is inaccurate and paragraph 2) should be removed.

It is allowed for the self-healing (*failedTries* decrement) to accumulate between TPM Reset, TPM Restart, and TPM Resume. In the current reference implementation, the self-healing does not accumulate between boots because *selfHealTimer* and *lockoutTimer* are stored in volatile memory. Instead these values could be stored in the orderly data structure which is saved to non-volatile memory on each TPM2_Shutdown. When the DA parameters are initialized at TPM2_Startup, credit can be given for the accumulated time.

A note should be added to Part 1, 19.8.2 Lockout Mode Configuration Parameters that the TPM may keep track of the time elapsed toward *recoveryTime* at shutdown and use that against the *recoveryTIme* upon power up.

2.7 TDES Key Parity Calculation

The following description of the parity calculation of TDES keys should be added to Part 1.

A TDES key is generated by getting 24 bytes from the random number generator appropriate for the type of key generation (such as a KDF for a derived key). The 24 bytes are treated as 3, 64-bit values in canonical TPM form (big-endian bytes). The odd parity is then generated for each byte with the parity replacing the least significant bit in each byte to create 3 DES keys. The resulting three DES keys are then validated to make sure that none of them is on the list of prohibited DES key values. If any of the generated key values is prohibited, then the TPM will repeat the key generating process by generating 24 new bytes.

2.8 Mode validation in TPM2_EncryptDecrypt, and TPM2_EncryptDecrypt2

The reference code in Part 3, 15.2 TPM2_EncryptDecrypt and 15.3 TPM2_EncryptDecrypt2 incorrectly validates the mode. If the symmetric mode specified in the *mode* input parameter is TPM_ALG_NULL and the mode of the key is not TPM_ALG_NULL, then the check for the input IV and the input data block size are performed with a wrong mode variable (set to TPM_ALG_NULL instead of the actual value). As a result, the TPM might return TPM_RC_SIZE even though input IV and input data are correctly set for the selected mode.

2.9 TPM2_Import – encryptedDuplication Check

The General Description in Part 3, 13.3 TPM2_Import says, "If *encryptedDuplication* is SET in the object referenced by *parentHandle*, then *encryptedDuplication* shall be SET in *objectPublic* (TPM_RC_ATTRIBUTES)."

In the reference code, TPM2_Load() verifies that if a parent object has *fixedTPM* CLEAR, the child must have the same *encryptedDuplication* value as its parent and otherwise return TPM_RC_ATTRIBUTES. This check may be done at TPM2_Import(). On TPM2_Load() this must be checked unless it was checked at TPM2_Import().

The parent and child object must have the same value for *encryptedDuplication* (both SET or CLEAR) if they are in the same duplication group. All objects in a duplication group are required to have the same setting for *encryptedDuplication*. Therefore, if a parent object has *fixedTPM* CLEAR, the child must have the same *encryptedDuplication* value as its parent.

2.10 TPMS_TIME_INFO.time

The General Description in Part 3, 9.3 TPM2_Startup says, TPMS_TIME_INFO.time shall be reset to zero on any TPM2_Startup. This text is incorrect and should be removed. The behaviour of TPMS_TIME_INFO.*time* is described in Part 1, 36.2 Time.

2.11 Separation Indicator 0x00 in KDFa

To clarify the use of the separation indicator 0x00 in KDFa, note 2 in Part 1, 11.4.9.2 KDFa() should be replaced with the following text.

As shown in equation (6), there is an octet of zero that separates *Label* from *Context*. In SP800-108, *Label* is a sequence of octets that may or may not have a final octet that is zero. If *Label* is not present, a zero octet is added. If *Label* is present and is not NULL-terminated, a zero octet is added. If *Label* is present and is NULL-terminated, the NULL becomes the zero octet and no additional zero octet is added.

2.12 TPM2_EvictControl

The reference code in Part 3, 28.5 TPM2_EvictControl allows a child key in the NULL hierarchy to be persisted. This is because the hierarchy information is not being properly propagated.

Objects in the NULL hierarchy are Temporary Objects that become unusable after a TPM Reset and that may not be converted into Persistent Objects. The condition when an object is allowed to be persisted is described in Part 1, 37.3 Owner and Platform Evict Objects.

2.13 TPM2B_TIMEOUT

In Part 2, 10.4.10 TPM2B_TIMEOUT is defined as a TPM-dependent structure with the size limited to the same as the digest structure (TPM2B_DIGEST). For the timeout parameter in TPM2_PolicySigned, TPM2_PolicySecret, and TPM2_PolicyTicket, the reference code uses an implementation-specific size of UINT64 plus one where the additional byte serves as an indicator whether an authorization ticket will expire on TPM Reset or TPM Restart.

This causes incompatibility with existing software. To fix this, only the format of TPM2B_TIMEOUT may be TPM-dependent. The size of timeout is allowed to be 8 bytes or less. Therefore, Table 81 in Part 2, 10.4.10 TPM2B_TIMEOUT should be replaced with:

Table 81 — Definition of Types for TPM2B_TIMEOUT

Parameter	Type	Description
size	UINT16	size of the timeout value
buffer [size] {:sizeof(UINT64)}	BYTE	the timeout value

NOTE In the reference implementation the MSb is used as a flag to indicate whether a ticket expires on TPM Reset or TPM Restart.

2.14 TPM2_NV_ChangeAuth

The General Description in Part 3, 31.15 TPM2_NV_ChangeAuth says, “The size of the *newAuth* value may be no larger than the size of authorization indicated when the NV Index was defined.”

This sentence should be replaced with “The size of the *newAuth* value may be no larger than the size of the digest produced by the *nameAlg* of the NV Index.”

2.15 Primary Seed and Proof Size

The Primary Seed and Proof size in the reference code are not set in compliance with the following size requirements in Part 1.

Part 1, 14.3.1 Introduction (of Primary Seed Properties) specifies that, “A Primary Seed is required to have at least twice the number of bits as the security strength of any symmetric or asymmetric algorithm implemented on the TPM.”

Part 1, 14.4 Hierarchy Proofs specified that, "The TPM should produce proof values that are the larger of either

- the size of the largest digest produced by any hash algorithm implemented on the TPM, or
- twice the size of the largest symmetric key supported by the TPM."

In the reference implementation, PRIMARY_SEED_SIZE is set to 32 bytes (in Implementation.h, Part 4, A.2) and PROOF_SIZE is set to be the size of the largest digest (in GpMacros.h, Part 4, 5.2). This is not suitable for all set of algorithms supported by a TPM (in particular not for Suite B where the AES key size is 256 bit). Therefore, the Primary Seed and Proof size should be adapted in the reference code in compliance with Part 1.

2.16 TPM2_NV_DefineSpace – NV Pin Pass/Fail

In the reference code in Part 3, 31.3 TPM2_NV_DefineSpace, line 47, the availability of NV Pin Pass an NV Pin Fail Indices incorrectly depend on the command code of TPM2_PolicySigned() (CC_PolicySigned). This should be changed to CC_PolicySecret.

2.17 OaepDecode()

The return code of the function CryptHashBlock() is incorrectly checked in Part 4, 10.2.18.4.6 OaepDecode(), line 282. The check should be for unequal *hLen* instead of smaller 0. As a result, TPM2_RSA_Decrypt() might fail with the scheme TPM_ALG_OAEP.

2.18 seedValue Size

Part 1, 27.7.4 seedValue specified that, "For all object types, when *seedValue* is present, it is at least half the size of the digest produced by the *nameAlg* of the object." This does not match the reference code implementation.

For an asymmetric parent, the reference code requires the *seedValue* to be between half the size and the size of the digest produced by the *nameAlg* of the object. The *seedValue* is used in the creation of the protection values that involves a KDF using an HMAC. For this, a value of half the digest size of the *nameAlg* is considered to be sufficient.

However, for a symmetric object, the reference code requires the *seedValue* to be exactly the size of the digest produced by the *nameAlg* of the object. The public identity is created from the hash of the *seedValue* and the *sensitive* value. The hash does not provide the same level of protection of the *seedValue* as the HMAC in the KDF, so it is better for the *seedValue* to have the same size as the *nameAlg* digest in this case.

The description in Part 1, 27.7.4 seedValue and Part 2, Table 195 (Definition of TPMT_SENSITIVE Structure) should be changed to match the reference implementation.

2.19 TPMI_DH_SAVED, TPMS_CONTEXT

If a TPM supports less than three transient objects and TPM2_ContextLoad() is executed with *context.savedHandle* = 0x80000002 (a transient object with the *stClear* attribute SET), the TPM might return TPM_RC_VALUE. This is incorrect and caused by a wrong handle type for *savedHandle*. The unmarshalling of the TPMI_DH_CONTEXT handle fails because if MAX_LOADED_OBJECTS is less than three, the value 0x80000002 is outside the allowed range for transient objects (which is TRANSIENT_FIRST to TRANSIENT_LAST).

To fix this, the handle type for *savedHandle* in the TPMS_CONTEXT structure should be changed from TPMI_DH_CONTEXT to TPMI_DH_SAVED. Therefore, Table 210 in Part 2, 14.5 TPMS_CONTEXT should be replaced with:

Table 210 — Definition of TPMS_CONTEXT Structure

Name	Type	Description
sequence	UINT64	the sequence number of the context NOTE Transient object contexts and session contexts used different counters.
savedHandle	TPMI_DH_SAVED	a handle indicating if the context is a session, object, or sequence object (see Table 211 — Context Handle Values)
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy of the context
contextBlob	TPM2B_CONTEXT_DATA	the context data and integrity HMAC

The following table and description for TPMI_DH_SAVED should be added to Part 2, clause 9 Interface Types:

This type defines the handle values that may be used in TPM2_ContextSave() or TPM2_ContextLoad().

Table 49 — Definition of (TPM_HANDLE) TPMI_DH_SAVED Type

Values	Comments
{HMAC_SESSION_FIRST : HMAC_SESSION_LAST}	an HMAC session context
{POLICY_SESSION_FIRST:POLICY_SESSION_LAST}	a policy session context
0x80000000	an ordinary transient object
0x80000001	a sequence object
0x80000002	a transient object with the *stClear* attribute SET
#TPM_RC_VALUE	

The commands affected by this change are TPM2_ContextLoad(), and TPM2_ContextSave().

2.20 Preservation of TPM vendor EKs

The following description on the preservation of Endorsement Keys provisioned by the TPM vendor should be added to Part 1, 14.3.1 Introduction of Primary Seed Properties.

After a field upgrade that changes the Primary Seed strength, or that changes the algorithm that uses the Primary Seed, the TPM shall generate the original EKs corresponding to the EK certificates provisioned by the TPM manufacturer if the same template is provided to the TPM2_CreatePrimary() command until such time as TPM2_ChangeEPS() command changes the EPS.

This requirement shall not be in effect for other keys derived from the EPS or for keys derived from the SPS or PPS.

EXAMPLE A field upgrade can cause TPM2_CreatePrimary() to generate a different key for the same input template. For example, revisions prior to revision 138 used KDFa, while revision 138 used DRBG. In addition, the security strength requirement could cause a change in the seed length if the field upgrade implements a stronger algorithm.

2.21 Encryption of *salt*

Part 1, 19.6.13.1 Overview (of Encryption of *salt*) states, “The *salt* parameter for TPM2_StartAuthSession() may be symmetrically or asymmetrically encrypted using the methods described in this clause.”

This statement is incorrect as the *salt* may only be asymmetrically encrypted. The reference code is implemented correctly.

2.22 TPM_PT_NV_COUNTERS_MAX

The following note should be added to Part 2, Table 23 — Definition of (UINT32) TPM_PT Constants, for the entry TPM_PT_NV_COUNTERS_MAX.

The value zero indicates that there is no fixed maximum. The number of counter indexes is determined by the available NV memory pool.

2.23 ECC Binding Check - AdjustNumberB()

When the public and private part of an ECC key is loaded, the binding between the public and private part is verified by the TPM. The public key is recalculated from the private key and the generator of the curve and compared to the input public key. Before the comparison, the reference code adjusts the size of the x and y coordinate by adding or removing leading zeros using the function AdjustNumberB() (in Part 4, 9.11.6).

However, the function AdjustNumberB() does not work correctly if the number needs to be reduced in size. The operations on the variable i (in line 126) incorrectly count the number of leading zeros. As a result, the binding check may fail with TPM_RC_BINDING.

2.24 TPM_SPEC Date Constants

Table 6 in Part 2, 6.1 TPM_SPEC (Specification Version Values) should be replaced with:

Table 6 — Definition of (UINT32) TPM_SPEC Constants <>

Name	Value	Comments
TPM_SPEC_FAMILY	0x322E3000	ASCII “2.0” with null terminator
TPM_SPEC_LEVEL	00	the level number for the specification
TPM_SPEC_VERSION	138	the version number of the spec (001.38 * 100)
TPM_SPEC_YEAR	2019	the year of the version
TPM_SPEC_DAY_OF_YEAR	235	the day of the year (August 23, 2019)

That is, the spec date fields TPM_SPEC_YEAR and TPM_SPEC_DAY_OF_YEAR should be set to the date of this Errata document.

2.25 Commit Random Value – hash algorithm

In Part 1, Annex C.2.2 Commit Random Value, equation (64):

r ≔ KDFa (*nameAlg*, *commitRandom*, “ECDAA Commit”, *name*, *commitCount*, *bits*)   (64)

The parameter *nameAlg* should be replaced with *vendorAlg* where *vendorAlg* is a vendor-defined hash algorithm (the same hash algorithm as used for context integrity).

The description of *bits* (under the equation (64)) should be changed from “the number of bits in a digest using *nameAlg*” to “the number of bits in the order of the curve of the signing key (*signHandle*)” with a note that when the number of bits is not a multiple of 8, it is rounded up to be a multiple of 8.

The reference code (in Part 4, 10.2.11.2.11 CryptGenerateR()) uses the correct hash algorithm and bit length for the generation of r.

2.26 TPM2_Certify – *qualifiedName*

Part 1, 31.5 Anonymous Signing says, “For TPM2_Certify() using an anonymous signing scheme, both the *qualifiedSigner* and *qualifiedName* of the certified key are set to an Empty Buffer.”

In the reference code, TPM2_Certify() – when using an anonymous signing scheme – does not set *qualifiedName* to an Empty Buffer, but sets it (independent of the signing scheme) to the *qualifiedName* of the object being certified (see Part 3, 18.2 TPM2_Certify, line 29).

The reference code should be fixed to set *qualifiedName* as specified in Part 1. (q*ualifiedSigner* is set correctly in the reference code.)

2.27 TPM2_PCR_Allocate

Both, Part 1, 17.8 PCR Allocation and Part 3, 22.5.1 General Description (of TPM2_PCR_Allocate) indicate that PCR allocation takes effect at TPM2_Startup (TPM_SU_CLEAR).

This is incorrect. The PCR allocation takes effect at _TPM_Init(). The reference code is implemented correctly.

2.28 TPM_PT_PS_REVISION

In Part 2, 6.13 TPM_PT (Property Tag), Table 23 - Definition of (UINT32) TPM_PT Constants, the property TPM_PT_PS_REVISION is described as “the specification Revision times 100 for the platform-specific specification”. This is incorrect and the description should be changed to “a platform specific value”.

2.29 Label in TPM2_RSA_Encrypt/Decrypt and TPM2_CreateLoaded

In the following cases, the TPM allows a label to be provided by the caller:

1) In TPM2_RSA_Encrypt/Decrypt(), where label is used in the RSAES_OAEP encryption scheme.

2) In TPM2_CreateLoaded() if a Derived Object is created, where label is used in KDFa().

The description in Part 1 (B.4 RSAES_OAEP, 11.4.9.2 KDFa()) and Part 3 (14.3 TPM2_RSA_Decrypt) of the Library specification define the label as “NULL-terminated” string, which is a “sequence of non-zero values followed by a value containing zero” (see Part 1, 4.42). Further, Part 3 (14.2 TPM2_RSA_Encrypt) defines that, “If a zero octet occurs before label.buffer[label.size-1], the TPM shall truncate the label at that point.”

However, the reference code does not truncate the label if a zero octet occurs before label.buffer[label.size-1]. In the case of TPM2_RSA_Encrypt/Decrypt(), the reference code verifies that the last octet in the label is zero. In the case of KDFa(), the reference code behaves as described in section 2.11 Separation Indicator 0x00 in KDFa (of this Errata document).

For interoperability with all implementations, it is required that the caller uses a label that is a “NULL-terminated” string.

In the future, the Library specification will define label as an octet string, to allow the label to be a Hash value (for example, in the case where the label would be larger than the digest size) and also to be consistent with NIST and other standards.

2.30 TPM2_LoadExternal – ECC Point Padding

If only the public portion of an ECC key is loaded with TPM2_LoadExternal(), the byte size of the x and y coordinates are compared to the byte size of the associated curve. Therefore, if the size of the x or y coordinate is smaller than the curve size, leading zeros must be added to the point values of *inPublic*, otherwise the TPM may return TPM_RC_KEY.

This requirement for padding the ECC public key for TPM2_LoadExternal() is missing in the description in Part 1, C.8 ECC Point Padding.

2.31 Max Size Check of Data Object

The function CryptValidateKeys() (in Part 4, 10.2.6.6.19) does not correctly check on the maximum size of a data object (KEYEDHASH object with *sign* and *decrypt* CLEAR), which is MAX_SYM_DATA (128). Therefore, the reference code fails to load and import a data object with a size that is larger than the size of the digest produced by the *nameAlg* of the object. Creation and duplication of such a data object succeed, but import and load may return TPM_RC_KEY_SIZE. CryptValidateKeys() is executed:

1) On TPM2_Import() – If the parent of the object has *fixedTPM* SET

2) On TPM2_Load() – If the parent of the object has *fixedTPM* CLEAR

3) On TPM2_LoadExternal()

To avoid this issue, the size of a data object should not be larger than the size of the digest produced by the *nameAlg* of the object.

2.32 pcrUpdateCounter

In Part 3, 22.1 Introduction (of Integrity Collection (PCR)), the NOTE 2 indicates that,

“If a command causes PCR in multiple banks to change, the PCR Update Counter may be incremented either once or once for each bank.”

This is incorrect, it should say,

If a command extends PCR in multiple banks, the PCR Update Counter must be incremented once for each bank. The commands that extend PCR are: TPM2_PCR_Extend, TPM2_PCR_Event, and TPM2_EventSequenceComplete.

If a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. The commands that reset PCR are: TPM2_PCR_Reset, and TPM2_Startup.

The corrected description matches the reference code implementation of *pcrUdpateCounter*.

2.33 Preservation of Orderly NV Index data

Part 1, 37.2.6 Updating an Index, does not correctly describe when the data of an orderly NV Index is preserved. The text should be updated for NV Ordinary, NV Bit Field, NV Extend, NV PIN Index to say,

If the Index has the TPMA_NV_ORDERLY attribute SET, then only the RAM version of the Index is written. The data is only preserved to NV on a Shutdown(STATE); and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

The reference code implementation in Part 4, 8.4.5.28 NvSetStartupAttributes() is correct.

2.34 NV PIN Indices

Part 1, 37.2.8.1 Restricting the number of uses of an object with PIN Pass, specifies the following behavior for PIN Pass NV Indices,

If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization succeeds.

Part 1, 37.2.8.2 Localized Dictionary Attack protection with PIN Fail, specifies the following behavior for PIN Fail NV Indices,

If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization fails. *pinCount* is reset to zero by the TPM whenever *authValue* authorization succeeds.

In the reference code, a successful authorization with the PIN Index *authPolicy* has the same effect on *pinCount* as a successful authorization with the PIN Index *authValue*. That means, for a PIN Pass NV Index, *pinCount* is incremented after *authPolicy* authorization succeeds, and for a PIN Fail NV Index, *pinCount* is reset to zero after *authPolicy* authorization succeeds. This behavior of the reference code is incorrect. Authorization with *ownerAuth*/*ownerPolicy*, *platformAuth*/*platformPolicy* is not affected by this issue, and will not increment or reset *pinCount*.

To avoid this issue, it is recommended to use Owner or Platform authorization to read the PIN Index. The PIN Index *authPolicy* should not be used to read the PIN Index, unless *authValue* is part of the policy for reading the Index. The following setting is recommended for a PIN Pass or PIN Fail NV Index when the Index is defined:

- If Owner authorized the creation of the index, TPMA_NV_OWNERREAD is SET and TPMA_NV_OWNERWRITE is CLEAR

- If Platform authorized the creation of the index, TPMA_NV_PPREAD is SET and TPMA_NV_PPWRITE is CLEAR

- TPMA_NV_POLICYREAD is CLEAR (unless *authValue* is part of the policy for reading the index)

- TPMA_NV_POLICYWRITE is SET

- TPMA_NV_WRITEALL is SET

To avoid that the right to read the PIN Index would allow someone to also write the Index, Owner or Platform authorization should not permit writing the PIN Index.

The PIN Index *authPolicy* may be used to write, or delete the PIN Index. When writing the Index, both, *pinCount* and *pinLimit* should be written, this can be ensured by TPMA_NV_WRITEALL.

To enable reading the NV PIN Index with Owner or Platform policy, the Owner or Platform policy should include a policy OR branch constructed of

1) TPM2_PolicyNameHash() with *nameHash* = $H_{policyAlg}$(TPM_RH_OWNER || *nvIndex→Name*) - AND -

2) TPM2_PolicyCommandCode() with *code* set to TPM_CC_NV_Read (the policy should not permit deleting the PIN Index with TPM2_UndefineSpace()).

2.35 Error Codes

2.35.1 Introduction

The following section resolves ambiguities with regards to errors codes where the specification text and the reference code specify something different.

2.35.2 TPM2_StartAuthSession – key scheme

The General Description in Part 3, 11.1 TPM2_StartAuthSession specifies that the TPM shall return TPM_RC_SCHEME if the scheme of the key (referenced by *tpmKey*) is not TPM_ALG_OAEP or TPM_ALG_NULL. However, the reference code returns TPM_RC_VALUE.

The preferred error code for this failure is TPM_RC_VALUE. But TPM_RC_SCHEME is also acceptable.

2.35.3 Lockout Mode

The text in Part 3, 25.1 Introduction of Dictionary Attack Functions says, “While the TPM is in Lockout mode, the TPM will return TPM_RC_LOCKED if the command requires use of an object’s or Index’s *authValue* unless the authorization applies to an entry in the Platform hierarchy.”

The error code should be TPM_RC_LOCKOUT.

2.35.4 NV Locked

In Part 3, 5.4 Handle Area Validation, paragraph b; 3) the text says,

i) If the command requires write access to the index data then TPMA_NV_WRITELOCKED is not SET (TPM_RC_LOCKED)
ii) If the command requires read access to the index data then TPMA_NV_READLOCKED is not SET (TPM_RC_LOCKED)

Both error codes should be TPM_RC_NV_LOCKED.

2.35.5 BnPointMul

In Part 4, 10.2.11.2.19 BnPointMul(), the entry in the return code table for TPM_RC_VALUE is incorrect. It says, TPM_RC_VALUE is returned if “d or u is not 0 < d < n”.

The values for the scalars d and u are allowed to be zero. The description should be changed to “d or u is not < n” to match the reference code implementation. In detail, this type of error is returned if d and u are NULL, S is present but d is NULL, only one of u or Q is present, or the curve parameters are NULL.

This implies that TPM2_VerifySignature() may verify an ECDSA signature on a digest with a valid size but the data value set to zero.

2.35.6 TPM2_SequenceComplete

The error return code table in Part 3, 17.5.3 Detailed Actions (of TPM2_SequenceComplete) indicates that the TPM should return TPM_RC_TYPE if *sequenceHandle* does not reference a hash or HMAC sequence object. The correct error code is TPM_RC_MODE as returned by the reference code.

2.35.7 TPM2_PolicyTemplate

The following input validation checks are missing in the reference code in Part 3, 23.21 TPM2_PolicyTemplate (due to a code merge issue).

- If *policySession→isTemplateHash* is SET and *policySession→cpHash* is not equal to *templateHash,* the TPM may return TPM_RC_VALUE or TPM_RC_CPHASH. (The preferred error code is TPM_RC_VALUE.)
- Otherwise, if *policySession→cpHash* is already set, the TPM may return TPM_RC_VALUE or TPM_RC_CPHASH. (The preferred error code is TPM_RC_CPHASH.)
- If the size of the *templateHash* input parameter is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE.