Application Notes

Generic KDF Support

07 October 2024

Table of Contents

1. Introduction

This application note describes the `DeriveMech_NISTKDFmGeneric` mechanism, supported in Security World firmware v13.5.1 and later.

2. Modes of Usage

The mechanism can be use in three modes.

2.1. Full Key Agreement

In this mode, the mechanism executes a full SP800-56Ar3 key agreement with a SP800-56Cr2 s5 two-step (extraction-then-expansion) key derivation to derive key material.

- The `Extract` flag must be set and the `kx` field must be present.
- The same PRF is used for the expansion and extraction steps.
- The `kx` field contains the ciphertext received from the peer.
- The `DeriveRole_BaseKey` input must be the local private key.

For example, if ECDH key agreement is being used:

- The `kx` field will be a `Mech_ECDHKeyExchange` ciphertext
- The base key will be a `KeyType_ECDHPrivate` key.

2.2. Two-Step Key Derivation

In this mode, the mechanism executes just the SP800-56Cr2 two-step key derivation.

- In this case the `Extract` flag must be set and the `kx` field must be absent.
- The same PRF is used for the expansion and extraction steps.
- The `DeriveRole_BaseKey` must be the input to the expansion step (called Z in SP800-56Cr2).

2.3. Expansion-only Key Derivation

In this mode, the mechanism executes just the SP800-108r1 key derivation function.

- In this case the `Extract` flag must be clear and the `kx` field must be absent.
- The `DeriveRole_BaseKey` must be the input to the extraction step (called K_{DK} in SP800-56Cr2 and K_{IN} in SP800-108r1).

3. DeriveMech_NISTKDFmGeneric API

The new mechanism is `DeriveMech_NISTKDFmGeneric`.

3.1. Parameters

The parameter structure is as follows:

```
struct M_DeriveMech_NISTKDFmGeneric_DKParams {
  M_DeriveMech_NISTKDFmGeneric_DKParams_flags flags;
  M_Word keylen;
  M_KeyType keytype;
  M_Mech prf;
  M_ByteBlock salt;
  M_ByteBlock context;
  M_ByteBlock iv;
  M_ByteBlock label;
  int n_fields;
  M_KDFField *fields;
  M_CipherText *kx;
};
```

`flags`	Flags word. See below.
`keylen`	Length of derived key in bits, e.g. 256
`keytype`	Type of derived key, e.g. `KeyType_Rijndael`
`prf`	PRF for randomness extraction and/or expansion, e.g. `Mech_HMACSHA256`
`salt`	Salt parameter for randomness extraction
`context`	Context parameter for expansion KDF
`iv`	Initial value parameter for KDF in feedback mode
`label`	Label parameter for expansion KDF
`n_fields`	Number of elements in `field`
`fields`	List of fields for expansion KDF (see below)
`kx`	Ciphertext for full key agreement

3.1.1. `flags` field

Possible flag bits are:

`DeriveMech_NISTKDFmGeneric_DKParams_flags_kx_present`	The `kx` field is present. The `DeriveRole_BaseKey` key is used to 'decrypt' the `kx` field, to produce the input to the subsequent KDF steps.
`DeriveMech_NISTKDFmGeneric_DKParams_flags_Extract`	Enable the extraction phase. If the `Extract` bit is set then both steps of the two-step KDF are performed. Otherwise only the second step is performed.

3.1.2. `fields` field

This gives the list of values to concatenate to form the input to the PRF, in SP800-108r1 s4.1 step 4(a) or s5.1 step 4(a). The following fields are supported:

`KDFField_Counter1r1`	The counter value, starting from 1, in a single byte
`KDFField_Counter1r2BE`	The counter value, starting from 1, in 2 bytes, in big-endian format
`KDFField_Counter1r4BE`	The counter value, starting from 1, in 4 bytes, in big-endian format
`KDFField_Counter1r2LE`	The counter value, starting from 1, in 2 bytes, in little-endian format
`KDFField_Counter1r4LE`	The counter value, starting from 1, in 4 bytes, in little-endian format
`KDFField_Lengthr1`	The length field, in a single byte
`KDFField_Lengthr2BE`	The length field, in 2 bytes, in big-endian format
`KDFField_Lengthr4BE`	The length field, in 4 bytes, in big-endian format
`KDFField_Lengthr2L`	The length field, in 2 bytes, in little-endian format
`KDFField_Lengthr4LE`	The length field, in 4 bytes, in little-endian format
`KDFField_Label`	The `label` field from the parameters
`KDFField_Context`	The `context` field from the parameters
`KDFField_ZeroByte`	A constant single-byte field with value 0
`KDFField_Feedback`	Feedback from the previous iteration, or the IV

There are some constraints which must be followed:

- No field may appear more than once.
- There can only be, at most, one counter field.
- There can only be, at most, one length field.

- There must be either a counter field or the feedback field (or both).

4. DeriveMech_NISTKDFmGeneric Example

4.1. Example Code

```
// Example of DeriveMech_NISTKDFmGeneric usage
#include <nfastapp.h>
#include <stdlib.h>
#include <stdio.h>

static NFast_AppHandle app;
static NFastApp_Connection conn;

// Hexdump an array.
static void hexdump(const unsigned char *ptr, size_t n)
{
  for (size_t i = 0; i < n; i++) printf("%02x", ptr[i]);
}

// Send a command to the HSM.
// Like everything in this example, in terminates the process
// on error.
static void transact(const M_Command *cmd, M_Reply *reply)
{
  M_Status err;
  char buffer[256];

  err = NFastApp_Transact(conn, NULL, cmd, reply, NULL);
  if (err) {
    NFast_Perror("NFastApp_Transact", err);
    exit(1);
  }
  if (NFastApp_Expected_Reply(app, NULL, buffer, sizeof buffer, reply, cmd->cmd, NULL) <= 0) {
    fprintf(stderr, "NFastApp_Expected_Reply: %s\n", buffer);
    exit(1);
  }
}

// Import a key with a given ACL.
// Returns the key handle.
static M_KeyID import_key(const M_KeyData *keydata, const M_ACL *acl)
{
  M_Command cmd = {
    .cmd = Cmd_Import,
    .args.import.acl = *acl,
    .args.import.data = *keydata,
  };
  M_Reply reply = { 0 };
  transact(&cmd, &reply);
  M_KeyID key = reply.reply.import.key;
  NFastApp_Free_Reply(app, NULL, NULL, &reply);
  return key;
}

// Export a key.
static void export_key(M_KeyID key, M_KeyData *keydata)
{
  M_Command cmd = {
    .cmd = Cmd_Export,
    .args.export.key = key,
  };
  M_Reply reply = { 0 };
  transact(&cmd, &reply);
```

```
    *keydata = reply.reply.export.data;
    memset(&reply.reply.export.data, 0, sizeof reply.reply.export.data);
    NFastApp_Free_Reply(app, NULL, NULL, &reply);
  }

  // Destroy a key.
  static void destroy_key(M_KeyID key)
  {
    M_Command cmd = {
      .cmd = Cmd_Destroy,
      .args.destroy.key = key,
    };
    M_Reply reply = { 0 };
    transact(&cmd, &reply);
    NFastApp_Free_Reply(app, NULL, NULL, &reply);
  }

  // Get the nCore key hash of a key.
  static M_KeyHash get_key_hash(M_KeyID key)
  {
    M_Command cmd = {
      .cmd = Cmd_GetKeyInfo,
      .args.getkeyinfo.key = key,
    };
    M_Reply reply = { 0 };
    transact(&cmd, &reply);
    M_KeyHash keyhash = reply.reply.getkeyinfo.hash;
    NFastApp_Free_Reply(app, NULL, NULL, &reply);
    return keyhash;
  }

  // Import a template key that contains a given derive key ACL.
  // Returns the key handle.
  static M_KeyID import_template_key(void)
  {
    M_Status err;

    // The derived key ACL.
    // In this example the only thing we want to do is export it,
    // so that's all we allow. In the key was going to be used for
    // encryption and decryption then instead we would have the
    // _Encrypt and _Decrypt bits.
    M_Action derived_key_action = {
      .type = Act_OpPermissions,
      .details.oppermissions.perms = Act_OpPermissions_Details_perms_ExportAsPlain,
    };
    M_PermissionGroup derived_key_group = {
      .n_actions = 1,
      .actions = &derived_key_action,
    };
    M_ACL derived_key_acl = {
      .n_groups = 1,
      .groups = &derived_key_group,
    };

    // The template key ACL. This just needs to permit the key to
    // be ued as a template key.
    M_Action template_action = {
      .type = Act_DeriveKey,
      .details.derivekey.mech = DeriveMech_NISTKDFmGeneric,
      .details.derivekey.role = DeriveRole_TemplateKey,
    };
    M_PermissionGroup template_group = {
      .n_actions = 1,
      .actions = &template_action,
    };
    M_ACL template_key_acl = {
```

```
    .n_groups = 1,
    .groups = &template_group,
  };

  // DKTemplate 'key' material is a marshaled ACL
  M_KeyData template_keydata = {
    .type = KeyType_DKTemplate,
  };
  if ((err = NFastApp_MarshalACL(
         app, NULL, NULL, &derived_key_acl, &template_keydata.data.dktemplate.nested_acl))) {
    NFast_Perror("NFastApp_MarshalACL", err);
    exit(1);
  }

  M_KeyID template_key = import_key(&template_keydata, &template_key_acl);
  NFastApp_Free(app, template_keydata.data.dktemplate.nested_acl.ptr, NULL, NULL);
  return template_key;
}

// Import the base key.
// Returns the key handle.
static M_KeyID import_base_key(size_t base_len, unsigned char *base, M_KeyHash template_key_hash)
{
  // The base key ACL
  M_KeyRoleID otherkeys = {
    .hash = template_key_hash,
    .role = DeriveRole_TemplateKey,
  };
  M_Action base_action = {
    .type = Act_DeriveKey,
    .details.derivekey.mech = DeriveMech_NISTKDFmGeneric,
    .details.derivekey.role = DeriveRole_BaseKey,
    // Restrict the ACL of the derive key to that given in
    // template_key_hash. In this example, where we just
    // export the derived key, this is futile, but in
    // some use cases the restriction is useful.
    .details.derivekey.n_otherkeys = 1,
    .details.derivekey.otherkeys = &otherkeys,
  };
  M_PermissionGroup base_group = {
    .n_actions = 1,
    .actions = &base_action,
  };
  M_ACL base_key_acl = {
    .n_groups = 1,
    .groups = &base_group,
  };

  // The base key material
  M_KeyData base_keydata = {
    .type = KeyType_Random,
    .data.random.k.ptr = base,
    .data.random.k.len = (M_Word)base_len,
  };

  return import_key(&base_keydata, &base_key_acl);
}

// Derive a key from a given base key.
// Returns the derived key handle.
static M_KeyID derive_key(size_t context_len, unsigned char *context, M_KeyID template_key,
                          M_KeyID base_key)
{
  // The set of fields used in the input to the PRF.
  static M_KDFField fields[] = { KDFField_Counter1r4BE, KDFField_Context, KDFField_Lengthr4BE };

  // Key handles passed to the command.
```

```
  // The order is always { template, base, wrap, ... }
  M_KeyID keys[] = { template_key, base_key };

  M_Command cmd = {
    .cmd = Cmd_DeriveKey,
    .args.derivekey.n_keys = 2,
    .args.derivekey.keys = keys,
    .args.derivekey.mech = DeriveMech_NISTKDFmGeneric,
    .args.derivekey.params.nistkdfmgeneric.keylen = 32 * 8, // bits
    .args.derivekey.params.nistkdfmgeneric.keytype = KeyType_Random,
    .args.derivekey.params.nistkdfmgeneric.prf = Mech_HMACSHA256,
    .args.derivekey.params.nistkdfmgeneric.context.len = (M_Word)context_len,
    .args.derivekey.params.nistkdfmgeneric.context.ptr = context,
    .args.derivekey.params.nistkdfmgeneric.n_fields = 3,
    .args.derivekey.params.nistkdfmgeneric.fields = fields,
  };
  M_Reply reply = { 0 };

  transact(&cmd, &reply);
  M_KeyID derived_key = reply.reply.derivekey.key;
  NFastApp_Free_Reply(app, NULL, NULL, &reply);
  return derived_key;
}

// Derive one frmo a given context and base key material
static void derive_key_example(size_t context_len, unsigned char *context, size_t base_len,
                               unsigned char *base)
{

  printf("Context: ");
  hexdump(context, context_len);
  printf("\n");
  printf("Base: ");
  hexdump(base, base_len);
  printf("\n");

  // Set up the inputs
  M_KeyID template_key = import_template_key();
  M_KeyID base_key = import_base_key(base_len, base, get_key_hash(template_key));

  // Do the derivation
  M_KeyID derived_key = derive_key(context_len, context, template_key, base_key);

  // Retrieve the derived key material
  M_KeyData derived_keydata;
  export_key(derived_key, &derived_keydata);

  // In this example we assume KeyType_Random output
  assert(derived_keydata.type == KeyType_Random);

  // Display the derived key material
  printf("Output: ");
  hexdump(derived_keydata.data.random.k.ptr, derived_keydata.data.random.k.len);
  printf("\n");

  // Clean up key handles
  destroy_key(base_key);
  destroy_key(template_key);
  destroy_key(derived_key);

  printf("\n");
}

// Run the KDF with some example inputs
static void example(void)
{
  static struct
```

```
  {
    size_t context_len;
    unsigned char context[64];
    size_t base_len;
    unsigned char base[64];
  } example_data[] = {
    {
      .context_len = 2,
      .context = { 0x00, 0x00 },
      .base_len = 32,
      .base = {
        0x66, 0x73, 0x6b, 0x70, 0x5f, 0x6b, 0x64, 0x6b, 0x20, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65,
        0x20, 0x68, 0x6f, 0x77, 0x20, 0x66, 0x73, 0x6b, 0x70, 0x20, 0x64, 0x65, 0x72, 0x69, 0x76, 0x65,
      },
    },
    {
      .context_len = 34,
      .context = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
        0x00, 0x00,
      },
      .base_len = 32,
      .base = {
        0x66, 0x73, 0x6b, 0x70, 0x5f, 0x6b, 0x64, 0x6b, 0x20, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65,
        0x20, 0x68, 0x6f, 0x77, 0x20, 0x66, 0x73, 0x6b, 0x70, 0x20, 0x64, 0x65, 0x72, 0x69, 0x76, 0x65,
      },
    },
    {
      .context_len = 3,
      .context = { 0x01, 0x00, 0x01 },
      .base_len = 32,
      .base = {
        0x66, 0x73, 0x6b, 0x70, 0x5f, 0x6b, 0x64, 0x6b, 0x20, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65,
        0x20, 0x68, 0x6f, 0x77, 0x20, 0x66, 0x73, 0x6b, 0x70, 0x20, 0x64, 0x65, 0x72, 0x69, 0x76, 0x65,
      },
    },
    {
      .context_len = 9,
      .context = { 0x00, 0x01, 0x00, 0x02, 0x00, 0x00, 0x03, 0x00, 0x04,  },
      .base_len = 32,
      .base = {
        0x66, 0x73, 0x6b, 0x70, 0x5f, 0x6b, 0x64, 0x6b, 0x20, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65,
        0x20, 0x68, 0x6f, 0x77, 0x20, 0x66, 0x73, 0x6b, 0x70, 0x20, 0x64, 0x65, 0x72, 0x69, 0x76, 0x65,
      },
    },
  };

  for (size_t i = 0; i < sizeof example_data / sizeof *example_data; i++) {
    derive_key_example(example_data[i].context_len,
                       example_data[i].context,
                       example_data[i].base_len,
                       example_data[i].base);
  }
}

int main(void)
{
  M_Status err;

  // Set up connectivity to the HSM
  if ((err = NFastApp_InitEx(&app, NULL, NULL))) {
    NFast_Perror("NFastApp_InitEx", err);
    exit(1);
  }
  if ((err = NFastApp_Connect(app, &conn, 0, NULL))) {
    NFast_Perror("NFastApp_Connect", err);
```

```
    exit(1);
  }

  example();

  return 0;
}
```

4.2. Running the example

To compile it under Linux:

```
gcc -I/opt/nfast/c/ctd/gcc/include/ \
    -o nistkdfmgeneric \
    nistkdfmgeneric.c \
    -L /opt/nfast/c/ctd/gcc/lib \
    -lnfstub -lnflog -lcutils
```

To run it:

```
./nistkdfmgeneric
```

The output should look like this:

```
Context: 0000
Base: 66736b705f6b646b206578616d706c6520686f772066736b7020646572697665
Output: f89b8dd626faa9a41365f20606969e1113d90f6340258c1308aa3927611b9b1c

Context: 00000000000000000000000000000000000000000000000000000000000000010000
Base: 66736b705f6b646b206578616d706c6520686f772066736b7020646572697665
Output: 0aa0dd4dbb3776e80474b638212041b76fa580d66a3e0fab6a741e482c85be69

Context: 010001
Base: 66736b705f6b646b206578616d706c6520686f772066736b7020646572697665
Output: 9259bb22085e479633e45554034230e44f94f688829993082b9ebeeebefc335f

Context: 000100020000030004
Base: 66736b705f6b646b206578616d706c6520686f772066736b7020646572697665
Output: 467b3dbf1a07d0aa69c3943f11448638f6cd0b16737d6887ed2de1285c6fc2a5
```

4.3. Limitations

- The example only covers using the final (expansion) step.
- The input key is imported rather than derived from some other source.
- No effort is made to secure the derived key

5. RFC5869 HKDF

RFC5869 defines a HMAC-based Extract-and-Expand Key Derivation Function. This can be implemented using `DeriveMech_NISTKDFmGeneric` as follows:

- `prf` should be the HMAC mechanism corresponding to the RFC5869 Hash parameter, e.g. `Mech_HMACSHA256` for SHA-256.
- `iv` should be the empty byte string.
- `label` can be used to pass the RFC5869 `info` string.
- `n_fields` should be 3 and `fields` should be `{ KDFField_Feedback, KDFField_Label, KDFField_Counter1r1 }`