Application Notes

nCore Key Hash

01 July 2024

Table of Contents

1. Considerations with key hashing in nCore

In the nCore API and documentation, keys are referred to by their hash. This document describes how the hash of a given key is calculated.

For information about which key type is supported in which nShield firmware or Security World release, see the *nShield Security World Release Notes*.

The hash of a private key is always the hash of a corresponding public key, therefore you can use key hashes to tie together the two halves of a key pair. This implies that computing the hash of a private key involves a private-to-public derivation step.

No Security World parameters are included in this algorithm. If you import the same key in to HSMs that are enrolled in different Security Worlds, you get the same key hash.

The subject of the hash is not simply the `M_KeyData` structure, marshaled in a consistent way. This is because:

- A single ECC key may have multiple types: `ECDHPublic` or `ECDHLaxPublic`. nCore gives them the same hash, because they are the same key, used in much the same way, just with slightly different restrictions.
- A single ECC key may have multiple representations: either using a named curve, or as one of two types of a custom curve. nCore gives them the same hash because they are just different representations of the same key.
- An AES key may have the same raw value as an HMAC key. nCore gives them different hashes because the way they are used is fundamentally different.
- An ECDH key may have the same raw value as an ECDSA key. nCore gives them different hashes because the way they are used is fundamentally different.

2. Construction of nCore key hashes

nCore key hashes are SHA1 hashes of a sequence of bytes, or, if you are using `KeyHashEx`, the specified hash function, for example SHA256, of the same sequence.

Key hashes are constructed the following way in nCore:

```
"nFast KeyHash\0" || HASHPREFIX || "\0" || KEY || "invented by nCipher 1997\0"
```

Component	Meaning
`"nFast KeyHash\0"`	zero-terminated header string
`HASHPREFIX || "\0"`	zero-terminated identifying string for the key type, see Identifying strings and key material
`KEY`	Key material, see Identifying strings and key material
`"invented by nCipher 1997\0"`	zero-terminated trailer string

2.1. Identifying strings and key material

The following notation is used in the definitions:

Notation	Meaning
`||`	Concatenation
`bitlen(x)`	The number of bits required to represent the integer `x`.
`B(x)`	The representation of the integer `x` in little-endian form, zero-padded to a multiple of 64 bytes (bytes, not bits). Note that this does not include any kind of length indicator.
`Curve`	Elliptic curve domains, see Elliptic curve domains.
`S(x)`	The representation of the integer `x` in little-endian form in exactly 4 bytes.

Key type	Identifying String (HASHPREFIX)	Key Material (KEY)
`ArcFour`	`RC4`	Raw key material
`ARIA`	`ARIA`	Raw key material
`Camellia`	`Camellia`	Raw key material
`CAST256`	`CAST256`	Raw key material

Key type	Identifying String (HASHPREFIX)	Key Material (KEY)
DES	DES	Raw key material
DES2	DES2	Raw key material
DES3	DES3	Raw key material
DHExPrivate DHExPublic	DH001	B(p) || B(q) || B(g) || B(gx)
DHPrivate DHPublic	DH000	B(p) || B(g) || B(gx)
DKTemplate	TEM01	appdata || nested_acl
DSAComm DSACommVariableSEED	DSC01	B(p) || B(q) || B(g)
DSACommFIPS186_3	DSC02	B(p) || B(q) || B(g)
DSAPrivate DSAPublic	DSA00	B(p) || B(q) || B(g) || B(y)
ECDHPrivate ECDHPublic ECDHLaxPrivate ECDHLaxPublic	ECDH000	Curve || B(Q.x) || B(Q.y)
ECDSAPrivate ECDSAPublic	ECDSA00	Curve || B(Q.x) || B(Q.y)
ECPrivate ECPublic	EC00	Curve || B(Q.x) || B(Q.y)
Ed25519Private Ed25519Public	ED25519	Raw public key material
Ed448Private Ed448Public	ED448	Raw public key material
HMACMD5	HMACMD5	Raw key material

Key type	Identifying String (HASHPREFIX)	Key Material (KEY)
HMACRIPEMD160	HMACRIPEMD160	Raw key material
HMACSHA1	HMACSHA1	Raw key material
HMACSHA224	HMACSHA224	Raw key material
HMACSHA256	HMACSHA256	Raw key material
HMACSHA384	HMACSHA384	Raw key material
HMACSHA512	HMACSHA512	Raw key material
HMACSHA3b224	HMACSHA3b224	Raw key material
HMACSHA3b256	HMACSHA3b256	Raw key material
HMACSHA3b384	HMACSHA3b384	Raw key material
HMACSHA3b512	HMACSHA3b512	Raw key material
HMACTiger	HMACTiger	Raw key material
KCDSAComm	KCDSAC01	B(p) || B(q) || B(g)
KCDSAPrivate KCDSAPublic	KCDSA00	B(p) || B(q) || B(g)
KMAC128	KMAC128	Raw key material
KMAC256	KMAC256	Raw key material
MILENAGEOP	MILENAGEOP	Raw key material
MILENAGEOPC	MILENAGEOPC	Raw key material
MILENAGERC	MILENAGERC	Raw key material
MILENAGESubscriber	MILENAGESubscriber	Raw key material
Random	Random	Raw key material
Rijndael (AES)	Rijndael	Raw key material
RSAPrivate RSAPublic	RSA00	If bitlen(e) > 32: S(bitlen(e)) || B(e) || B(n) Otherwise: B(e) || B(n)

Key type	Identifying String (HASHPREFIX)	Key Material (KEY)
`SEED`	`SEED`	Raw key material
`TUAKTOP`	`TUAKTOPC`	Raw key material
`TUAKTOPC`	`TUAKTOPC`	Raw key material
`TUAKSubscriber`	`TUAKSubscriber`	Raw key material
`Wrapped`	`WRP01`	Raw key material

`X25519Private`
 	
`X25519Public`
	`X25519`	Raw public key material

3. Representation of ECC domain parameters

Elliptic curve domains are represented as one of two possibilities.

For curves over prime fields:

```
Curve = S(type) || S(field.type) || S(field.bitsize) ||
        B(p) ||
        B(a) || B(b) || B(G.x) || B(G.y) || B(r) || B(h)
```

For curves over binary polynomial fields:

```
Curve = S(type) || S(field.type) || S(field.bitsize) ||
        S(num_terms) || S(T1) || ⋯ || S(Tnum_terms) ||
        B(a) || B(b) || B(G.x) || B(G.y) || B(r) || B(h)
```

In binary fields, field elements are represented as integers. Bit `i` of the integer is the coefficient of x^i.

Component	Field type	Meaning
`B(a)`	both	Curve parameter `a`
`B(b)`	both	Curve parameter `b`
`B(G.x)`	both	`x` coordinate of `G`, the subgroup generator (also known as the base point)
`B(G.y)`	both	`y` coordinate of `G`
`B(h)`	both	`h`, the cofactor (that is, curve order divided by subgroup order)
`B(p)`	prime	The prime modulus `p`, for a prime field
`B(r)`	both	The name `r` is used in nCore and corresponds to FIPS 186-2. In FIPS 186-4 this parameter is called `n`.
`S(field.bitsize)`	both	An indicator of the the field size: • `field.bitsize=bitlen(p)` for a prime field • `field.bitsize` the highest power in the irreducible polynomial, for a binary field. For example, `409` in the polynomial $x^{409}+x^{87}+1$.

Component	Field type	Meaning
`S(field.type)`	both	The field type: • `field.type=0` for a prime field • `field.type=1` for a binary field with polynomial basis
`S(num_terms)`	binary	The number of nonzero terms in the irreducible polynomial, for a binary field. For example, `3` for the polynomial $x^{409}+x^{87}+1$ example above.
$S(T_i)$	binary	The indices of the nonzero terms T_i in ascending order of the irreducible polynomial, for a binary field. For example, the polynomial $x^{409}+x^{87}+1$ is represented by the ordered list of indices `S(0) || S(87) || S(409)`
`S(type)`	both	The curve type: • `type=0` for prime field • `type=1` for binary field

4. nCore key hash example

This example demonstates calculating a key hash using the nShield implementation, and then manually.

- The input to the hash includes the full domain parameters for the elliptic curve used, even though the key is represented (in nShield terms) as a named curve.
- Different ECC key types (`ECPublic`, `ECDSAPublic`, `ECDHPublic`, etc) have different `HASHPREFIX` values

```
#!/opt/nfast/python3/bin/python3
import choosealg
import hashlib

keydata = choosealg.KeyData(['ECPublic',
    'NISTP256',
    0,
    0x08f81421c7e89597d9b5a2bbd536f7305bcbd669194a281338cdb613bb0f9fd1,
    0x2d5b7759bd9e3fcd4aea4f4853f2aaeac675aa165947e3438044f6e57ccd89c4
])
print("Public key material:")
print(keydata)
print()

print("Hash calculated by nShield implementation:")
keyhash = choosealg.calchash(keydata, choosealg.KeyHashMech('SHA1Hash'))
print(keyhash.data.hash)
print()

def S(x):
    """The representation of the integer x in little-endian form
    in exactly 4 bytes."""
    return x.to_bytes(4, 'little')

def B(x):
    """The representation of the integer x in little-endian form,
    zero-padded to a multiple of 64 bytes."""
    xbytes = (x.bit_length() + 7) // 8
    return x.to_bytes(((xbytes + 63) // 64) * 64, 'little')

# Header string
input = b"nFast KeyHash\0"

# HASHPREFIX identifies key type
input += b"EC00\0"

# Curve type is prime
input += S(0)

# Field type is prime
input += S(0)

# Field bit size
input += S(256)

# Field modulus
input += B(0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff)

# Curve parameters a, b
input += B(0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc)
input += B(0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b)
```

```
# Subgroup generator G.x, G.y
input += B(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296)
input += B(0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)

# Subgroup order r (or sometimes n)
input += B(0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551)

# Cofactor h
input += B(1)

# Point coordinates
input += B(keydata.data.Q.x)
input += B(keydata.data.Q.y)

# Trailer
input += b"invented by nCipher 1997\0"

print("Input to hash function:")
input_hex = input.hex()
for i in range(0, len(input_hex), 64):
    print(input_hex[i:i+64])
print()

print("Hash calculated manually:")
print(hashlib.sha1(input).hexdigest())
```

The output is as follows:

```
Public key material:
KeyData.type= ECPublic
       .data.curve.name= NISTP256
            .Q.flags= 0x0
              .x= 0x8f81421c7e89597d9b5a2bbd536f7305bcbd669194a281338cdb613bb0f9fd1
              .y= 0x2d5b7759bd9e3fcd4aea4f4853f2aaeac675aa165947e3438044f6e57ccd89c4

Hash calculated by nShield implementation:
6ac2377c eaac44ea b378518d 1b6f4ebf 0d4d0dec

Input to hash function:
6e46617374204b657948617368004543303000000000000000000000010000ff
ffffffffffffffffffffffff0000000000000000000000000100000fffffff00
00000000000000000000000000000000000000000000000000000000000000fc
ffffffffffffffffffffffff0000000000000000000000000100000fffffff00
000000000000000000000000000000000000000000000000000000000000004b
60d2273e3cce3bf6b053ccb0061d65bc86987655bdebb3e7933aaad835c65a00
0000000000000000000000000000000000000000000000000000000000000096
c298d84539a1f4a033eb2d817d0377f240a463e5e6bcf847422ce1f2d1176b00
000000000000000000000000000000000000000000000000000000000000f5
51bf376840b6cbce5e316b5733ce2b169e0f7c4aebe78e9b7f1afee242e34f00
0000000000000000000000000000000000000000000000000000000000000051
2563fcc2cab9f3849e17a7adfae6bcffffffffffffffff00000000ffffffff00
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000d1
9f0fbb13b6cd3813284a1969d6cb5b30f736d5bba2b5d99795e8c72114f80800
00000000000000000000000000000000000000000000000000000000000000c4
89cd7ce5f6448043e3475916aa75c6eaaaf253484fea4acd3f9ebd59775b2d00
0000000000000000000000000000000000000000000000000000000000000069
6e76656e746564206279206e436970686572203139393700

Hash calculated manually:
6ac2377ceaac44eab378518d1b6f4ebf0d4d0dec
```