Status and Issues for the “Client-Server” Suite of Drafts

draft-ietf-netconf-crypto-types-10
draft-ietf-netconf-trust-anchors-05
draft-ietf-netconf-keystore-12
draft-ietf-netconf-tcp-client-server-02
draft-ietf-netconf-ssh-client-server-14
draft-ietf-netconf-tls-client-server-14
draft-ietf-netconf-netconf-client-server-14
draft-ietf-netconf-restconf-client-server-14

Adopted

+

draft-kwatsen-netconf-http-client-server-03

Not Yet Adopted

NETCONF WG

IETF 105 (Montreal)

Since IETF 104

All drafts updated and submitted as a set multiple times!

High-level Updates:

crypto-Types:
- algorithms: identities --> enumerations
- lots of tweaks on key creation

trust-anchors:
- renamed to "truststore"
- Added "local-or-truststore" groupings

keystore:
- Added support fro symmetric-keys
- Now have RPCs (not actions) to return (potentially encrypted) key.

tcp-client-server:
- added a couple 'feature' statements

ssh-client-server:
- minor changes

tls-client-server:
- minor changes

http-client-server:
- major changes (but not adopted yet)

netconf-client-server:
- minor changes

restconf-client-server:
- major restructuring of the "server" module
  - same restructuring SHOULD be applied the the "client" module, as well as both the NETCONF client & server modules.
- Added support for HTTP (w/o TLS) for when the TLS is terminated by an external system

trying to Last Call ASAP

This Presentation's Focus

1. ietf-crypto-types
   - algorithm identities strategy
2. ietf-keystore
   - device-generated and hidden keys strategy
3. ietf-restconf-server
   - overall structure strategy

Begin Discussion #1

Algorithm identities strategy.

High-level Decision

Stay the course or Simplify?

The current approach attempts to unify the algorithms produced by disconnected Sec Area efforts.

- This is the Right Thing To Do(™), but is difficult...

The Chairs spoke with Sec Dir representative (Rifaat Shekh-Yusef) last week, who plans to discuss with ADs this week.

While the authors wish to see this approach play out, we're beginning to think about a fallback strategy... (slides coming up)

Crypto Algorithm Identifiers: Identity vs. Enumerate

Background:

- Initially we use identity to identify each security algorithm.
- Recently, some expert from mailing list suggest that Enum could be a better way.
- The benefit of Enum type over identity are:
  - Simple to manage
  - When new algorithm is designed, add a single value in the IANA, RFCs from different group can refer to the value in their protocol definition.

Before:

```
identity hash-algorithm {
      description "A base identity for hash algorithm verification.";
}

identity sha-224 {
      base "hash-algorithm";
      description "The SHA-224 algorithm.";
      reference "RFC 6234: US Secure Hash Algorithms.";
}
```

After:

```
typedef hash-algorithm-t {
    type union {
        type uint16;
        type enumeration {
            enum NONE {
                value 0;
                description "Hash algorithm is NULL.";
            }
            enum sha1 {
                value 1;
                status obsolete;
                description "The SHA1 algorithm.";
                reference "RFC 3174: US Secure Hash Algorithms 1 (SHA1).";
            }
    }
}
```

Unified Identifiers for Crypto Algorithms

In IETF, a number of working groups are working on security related RFCs, such as:

- tls, ipsecme, i2nsf, …
- Many RFCs have been published, many crypto algorithms have been defined and registered to the IANA
  - Different group and different RFC has their own style in defining crypto algorithm
- Efforts are required to define, review, implement and manage these algorithms

Open Question

- Should we come up a unified framework (i.e a unified list or IANA page) for crypto algorithms identifier definition so that they can be shared among different groups and RFCs?
- The Enum list of crypto algorithm in draft-ietf-crypto-types can be a start point
- Better methods are welcome

Public Key Algorithm Name	Public Key Format	Reference	Note
ssh-dss	ssh-dss	[RFC4253]	Section 6.6
ssh-rsa	ssh-rsa	[RFC4253]	Section 6.6
rsa-sha2-256	ssh-rsa	[RFC8332]	Section 3
rsa-sha2-512	ssh-rsa	[RFC8332]	Section 3
spki-sign-rsa	spki-sign-rsa	[RFC4253]	Section 6.6
spki-sign-dss	spki-sign-dss	[RFC4253]	Section 6.6
pgp-sign-rsa	pgp-sign-rsa	[RFC4253]	Section 6.6
pgp-sign-dss	pgp-sign-dss	[RFC4253]	Section 6.6
null	null	[RFC4462]	Section 5
ecdsa-sha2-*	ecdsa-sha2-*	[RFC5656]	
x509v3-ssh-dss	x509v3-ssh-dss	[RFC6187]	
x509v3-ssh-rsa	x509v3-ssh-rsa	[RFC6187]	
x509v3-rsa2048-sha256	x509v3-rsa2048-sha256	[RFC6187]	
x509v3-ecdsa-sha2-*	x509v3-ecdsa-sha2-*	[RFC6187]	

Value	Description	Recommended	Reference
0x0000-0x0200	Reserved for backward compatibility		[RFC8446]
0x0201	rsa_pkcs1_sha1	Y	[RFC8446]
0x0202	Reserved for backward compatibility		[RFC8446]
0x0203	ecdsa_sha1	Y	[RFC8446]
0x0204-0x0400	Reserved for backward compatibility		[RFC8446]
0x0401	rsa_pkcs1_sha256	Y	[RFC8446]
0x0402	Reserved for backward compatibility		[RFC8446]
0x0403	ecdsa_secp256r1_sha256	Y	[RFC8446]
0x0404-0x0500	Reserved for backward compatibility		[RFC8446]
0x0501	rsa_pkcs1_sha384	Y	[RFC8446]
0x0502	Reserved for backward compatibility		[RFC8446]
0x0503	ecdsa_secp384r1_sha384	Y	[RFC8446]
0x0504-0x0600	Reserved for backward compatibility		[RFC8446]
0x0601	rsa_pkcs1_sha512	Y	[RFC8446]
0x0602	Reserved for backward compatibility		[RFC8446]
0x0603	ecdsa_secp521r1_sha512	Y	[RFC8446]

One "ietf-" module --> many "iana-" modules

Assuming we stay the course:

- Assigning to IANA allows IANA to update the modules outside of the standard RFC cycle.
  - this is important.
- Breaking up into many modules enables each to have better focus.
  - unclear how important this may be.

We would continue using "draft-ietf-crypto-types".

The Fallback Strategy

General Idea:

- Admit that TLS is the transport layer for most protocols
- Imagine a more TLS-aligned solution
- TLS protocol is heavily defined using ASN.1
- Already crypto-types defines many ASN.1 types
- So why not also add "OneAsymmetricKey" from RFC 5958 (Asymmetric Key Packages)?

```
OneAsymmetricKey ::= SEQUENCE {
  version                   Version,
  privateKeyAlgorithm       PrivateKeyAlgorithmIdentifier,
  privateKey                PrivateKey,
  attributes            [0] Attributes OPTIONAL,
  ...,
  [[2: publicKey        [1] PublicKey OPTIONAL ]],
  ...
}
```

Notice "PrivateKeyAlgorithmIdentifier"

```
PrivateKeyAlgorithmIdentifier ::= AlgorithmIdentifier
        { PUBLIC-KEY, { PrivateKeyAlgorithms } }
```

- privateKeyAlgorithm identifies the private-key algorithm and optionally contains parameters associated with the asymmetric key pair. The algorithm is identified by an object identifier (OID) and the format of the parameters depends on the OID, but the PrivateKeyAlgorithms information object set restricts the permissible OIDs. The value placed in privateKeyAlgorithmIdentifier is the value an originator would apply to indicate which algorithm is to be used with the private key.

```
AlgorithmIdentifier{}, ALGORITHM, PUBLIC-KEY, CONTENT-ENCRYPTION
  FROM AlgorithmInformation-2009
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-algorithmInformation-02(58) }
```

What is an OpenSSL Private Key

Is this a OneAsymmetricKey?

```
$ openssl ecparam -outform DER -out private_key.der -genkey -name prime256v1

$ openssl asn1parse -inform DER -in private_key.der

    0:d=0  hl=2 l= 119 cons: SEQUENCE
    2:d=1  hl=2 l=   1 prim: INTEGER           :01
    5:d=1  hl=2 l=  32 prim: OCTET STRING      [HEX DUMP]
   39:d=1  hl=2 l=  10 cons: cont [ 0 ]
   41:d=2  hl=2 l=   8 prim: OBJECT            :prime256v1
   51:d=1  hl=2 l=  68 cons: cont [ 1 ]
   53:d=2  hl=2 l=  66 prim: BIT STRING
```

Whatever OpenSSL is using, it would be most convenient to adopt.

Begin Discussion #2

Device-generated and hidden keys strategy.

Goals

1. Enable manufacturers to ship devices with IDevID certificates
   - implies that the associated private keys are hidden (e.g., TPM) (needed for SZTP)
2. Enable devices to generate keys
   - without the key values ever being disclosed (best practice)
3. Enable clients to install subsequently "hidden" keys
   - protected by a mechanism better than NACM
4. Support standard backup/restore interactions
   - all but the manufacturer-generated hidden keys should be migratable
5. Don't have keys in <running> with values only in <operational>
   - strange idioms should be avoided!

Several Strategies Tried

1. Action statements creating values in <operational>
    1. Pros: follows general rule about actions and <operational>
    2. Cons: copying "values" into <running> not easy

2. Encoded (crypt-hash like) values describing behavior
    1. Pros: all values in <running>
    2. Cons: encoding "verbs" into values bad

3. Disconnected RPCs (only return output, no datastore effect)
    1. PROs: satisfies all use cases (it seems)
    2. CONs: round-trip required to set generated key

Strategy #3 has traction, discussed next...

RPC-Based Approach

In ietf-crypto-types:

```
grouping symmetric-key-grouping {
  leaf algorithm { ... }
  choice key-type {
    leaf key { type binary; ... }
    leaf hidden-key { type empty; ... }
}

grouping asymmetric-key-pair-grouping {
  leaf algorithm { ... }
  leaf public-key { ... }
  choice private-key-type {
   leaf private-key { type binary; ... }
   leaf hidden-private-key { type empty; ... }
}
```

The secret part of key can be reported as "hidden", in which case the value is empty.

How keys come to be hidden is outside scope, but primarily conceived as being set during manufacturing-time magic and/or vendor-specific modules defining proprietary RPCs.

The ability to report *encrypted* keys is added by the ietf-keystore module. (see next slide)

RPC-Based Approach (cont.)

In ietf-keystore:

```
grouping key-reference-type-grouping {
 choice key-type {
   leaf symmetric-key-ref { ... }
   leaf asymmetric-key-ref { ... }
  }
}

grouping encrypted-value-grouping {
 uses "key-reference-type-grouping";
 leaf value { type binary; }
}

grouping symmetric-key-grouping {
 uses ct:symmetric-key-grouping {
    augment "key-type" {
     container encrypted-key {
       uses encrypted-value-grouping;
      }
    }
  }
}

grouping asymmetric-key-pair-grouping {
 uses ct:asymmetric-key-pair-grouping {
    augment "private-key-type" {
     container encrypted-private-key {
       uses encrypted-value-grouping;
      }
    }
  }
}
```

```
rpc generate-symmetric-key {
   input {
     leaf algorithm { ... }
     container encrypt-with {
       uses key-reference-type-grouping;
     }
   }
   output {
     uses ks:symmetric-key-grouping;
   }
 }

 rpc generate-asymmetric-key {
   input {
     leaf algorithm { ... }
     container encrypt-with {
      uses key-reference-type-grouping;
     }
   }
   output {
     uses ks:asymmetric-key-pair-grouping;
   }
 }
```

RPC-Based Approach (cont.)

```
module: ietf-keystore
  +--rw keystore
     +--rw asymmetric-keys
     |  +--rw asymmetric-key* [name]
     |     +--rw name                                  string
     |     +--rw algorithm                             asymmetric-key-algorithm-t
     |     +--rw public-key                            binary
     |     +--rw (private-key-type)
     |     |  +--:(private-key)
     |     |  |  +--rw private-key?                    binary  // clear-text value
     |     |  +--:(hidden-private-key)
     |     |  |  +--rw hidden-private-key?             empty   // no value
     |     |  +--:(encrypted-private-key)
     |     |     +--rw encrypted-private-key
     |     |        +--rw (key-type)
     |     |           |  +--:(symmetric-key-ref)
     |     |           |  |  +--rw symmetric-key-ref?     leafref // ref to another key in keystore
     |     |           |  +--:(asymmetric-key-ref)
     |     |           |     +--rw asymmetric-key-ref?    leafref // ref to another key in keystore
     |     |           +--rw value?                        binary  // cipher-text value
     |     +--rw certificates ...
     |     +---x generate-certificate-signing-request ...
     |
     +--rw symmetric-keys
        +--rw symmetric-key* [name]
           +--rw name                                   string
           +--rw algorithm                              encryption-algorithm-t
           +--rw (key-type)
              +--:(key)
              |  +--rw key?                             binary  // clear-text value
              +--:(hidden-key)
              |  +--rw hidden-key?                      empty   // no value
              +--:(encrypted-key)
                 +--rw encrypted-key
                    +--rw (key-type)
                    |  +--:(symmetric-key-ref)
                    |  |  +--rw symmetric-key-ref?     leafref // ref to another key in keystore
                    |  +--:(asymmetric-key-ref)
                    |     +--rw asymmetric-key-ref?    leafref // ref to another key in keystore
                    +--rw value?                        binary  // cipher-text value
```

Example

```
<keystore>
 <asymmetric-keys>
    <asymmetric-key>
      <name>tpm-protected-key</name>
      <algorithm>rsa2048</algorithm>
      <public-key>base64encodedvalue==</public-key>
      <hidden-private-key/>
      <certificates>
        <certificate>
          <name>builtin-idevid-cert</name>
        </certificate>
      </certificates>
    </asymmetric-key>
    <asymmetric-key>
      <name>encrypted-key</name>
      <algorithm>secp256r1</algorithm>
      <public-key>base64encodedvalue==</public-key>
      <encrypted-private-key>
        <symmetric-key-ref>operators-encrypted-key</symmetric-key-ref>
        <value>base64encodedvalue==</value>
      </encrypted-private-key>
    </asymmetric-key>
  </asymmetric-keys>
 <symmetric-keys>
    <symmetric-key>
      <name>operators-encrypted-key</name>
      <algorithm>aes-256-cbc</algorithm>
      <encrypted-key>
        <asymmetric-key-ref>tpm-protected-key</asymmetric-key-ref>
        <value>base64encodedvalue==</value>
      </encrypted-key>
    </symmetric-key>
  </symmetric-keys>
</keystore>
```

encrypted-by

Special key shipped with device. Used to encrypt operator's key.

Example general-use key. Can be migrated to other devices (i.e. RMA)

Special key used to encrypt all other keys. Installed by operator's crypto officer.

Satisfies Goals (Goals from earlier slide)

1. Enable manufacturers to ship devices with IDevID certificates
   - Key + certs in <operational> and, optionally, in <factory-default>
   - Private key reported via "hidden" leaf with "empty" type.

2. Enable devices to generate keys
   - RPCs defined to generate keys. (Client decides if key is encrypted or not)

3. Enable clients to install subsequently "hidden" keys
   - Client can set (e.g., <edit-config>) an encrypted key. (actually, better than hidden)

4. Support standard backup/restore interactions
   - All keys (except "hidden" keys) are reported.
   - RMA procedure can migrate the "operator-wide" key used to encrypt other keys.

5. Don't have keys in <running> with values only in <operational>
   - Check, all key values are "mandatory true". (see open Issue on next slide)

Open Issues

1. Report all key value's in <factory-default> and <running>?
   (i.e., should the "algorithm" or "public-key" nodes be suppressed)

   - PROs:
     - The YANG model is best stating all fields are "mandatory true".
     - There is no harm in reporting these nodes.
   - CONs:
     - The values only have to be reported in <operational> (so, it is unnecessary)
     - Special case handling?
       - Yes, but it's not an issue, and actually something that should be embraced.

         1. Already the server must handle special the <hidden-key/> empty type
            - i.e., don't actually delete the "algorithm" and "public-key" nodes
            - So, handling "don't actually modify" said nodes is no more effort

         2. Better would be to define an annotation that could generically identify nodes that are effectively read-only.
            - Such an annotation is needed in other use-cases (e.g., schema-mount)

Open Issues (cont.)

2. RMA requires new special step - Okay?

- Assuming operators follow what would effectively become best-practice
  - i.e., to create an "operator-wide" symmetric key, that is encrypted on each device using the device's "hidden" key, and subsequently used to encrypt all other keys.

- Then the RMA process would change:

  - OLD
    - load config from old device
    - additional customizations (e.g., install node-locked licenses)

  - NEW
    - manually install encrypted "operator-wide" on new device
    - edit config from old device to remove the old "operator-wide" key
      - it's the same value, but just encrypted by a different key
    - load modified config from old device
    - additional customizations (e.g., install node-locked licenses)

Begin Discussion #3

The ietf-restconf-server structure strategy.

Motivation: To follow the patten

(Set by the other client/server drafts)

- In the TCP, SSH, TLS, and HTTP models...
  - there is a grouping that represents the configuration for just a single connection.

- But this pattern is not followed in the NC and RC models
  - instead, an uber "application-level" grouping is provided supporting:
    - both standard and call-home use cases, with many endpoints for each!

Restructuring

Before:

```
module: ietf-restconf-server                          app stack container
 +--rw restconf-server
    +---u restconf-server-grouping

  grouping restconf-server-grouping                   app stack grouping
   +-- listen! {https-listen}?
   |  +-- endpoint* [name]
   |     +-- name?          string
   |     +-- (transport)
   |        +--:(https) {https-listen}?
   |           +-- https
   |              +-- tcp-server-parameters
   |              |  +---u restconf-server-grouping
   |              +-- tls-server-parameters
   |              |  +---u restconf-server-grouping
   |              +-- http-server-parameters
   |                 +---u restconf-server-grouping
   +-- call-home! {https-call-home}?
      +-- restconf-client* [name]
         +-- name?                 string
         +-- endpoints
         |  +-- endpoint* [name]
         |     +-- name?          string
         |     +-- (transport)
         |        +--:(https) {https-call-home}?
         |           +-- https
         |              +-- tcp-client-parameters
         |              |  +---u restconf-server-grouping
         |              +-- tls-server-parameters
         |              |  +---u restconf-server-grouping
         |              +-- http-server-parameters
         |                 +---u restconf-server-grouping
         +-- connection-type
         |  +-- (connection-type)
         |     +--:(persistent-connection)
         |     |  +-- persistent!
         |     +--:(periodic-connection)
         |        +-- periodic!
         |           +-- period?         uint16
         |           +-- anchor-time?    yang:date-and-time
         |           +-- idle-timeout?   uint16
         +-- reconnect-strategy
            +-- start-with?     enumeration
            +-- max-attempts?   uint8
```

no change

After:

```
module: ietf-restconf-server                          app stack container
 +--rw restconf-server
    +---u restconf-server-app-grouping

  grouping restconf-server-grouping                   per-connection grouping
   +-- client-identification
      +-- cert-maps
         +---u x509c2n:cert-to-name

  grouping restconf-server-listen-stack-grouping      listen stack grouping
   +-- (transport)
      +--:(http) {http-listen}?
      |  +-- http
      |     +-- external-endpoint
      |     |  +-- address    inet:ip-address
      |     |  +-- port?      inet:port-number
      |     +-- tcp-server-parameters
      |     |  +---u tcps:tcp-server-grouping
      |     +-- http-server-parameters
      |     |  +---u https:http-server-grouping
      |     +-- restconf-server-parameters
      |        +---u rcs:restconf-server-grouping
      +--:(https) {https-listen}?
         +-- https
            +-- tcp-server-parameters
            |  +---u tcps:tcp-server-grouping
            +-- tls-server-parameters
            |  +---u tlss:tls-server-grouping
            +-- http-server-parameters
            |  +---u https:http-server-grouping
            +-- restconf-server-parameters
               +---u rcs:restconf-server-grouping

  grouping restconf-server-callhome-stack-grouping    call-home stack grouping
   +-- (transport)
      +--:(https) {https-listen}?
         +-- https
            +-- tcp-client-parameters
            |  +---u tcpc:tcp-client-grouping
            +-- tls-server-parameters
            |  +---u tlss:tls-server-grouping
            +-- http-server-parameters
            |  +---u https:http-server-grouping
            +-- restconf-server-parameters
               +---u rcs:restconf-server-grouping

  grouping restconf-server-app-grouping               app stack grouping
   +-- listen! {https-listen}?
   |  +-- endpoint* [name]
   |     +-- name?                                    string
   |     +---u restconf-server-listen-stack-grouping
   +-- call-home! {https-call-home}?
      +-- restconf-client* [name]
         +-- name?                 string
         +-- endpoints
         |  +-- endpoint* [name]
         |     +-- name?                              string
         |     +---u restconf-server-callhome-stack-grouping
         +-- connection-type ...
         +-- reconnect-strategy ...
```

no change

Continue?

- Currently only applied to restconf-server...

	Client	Server
NETCONF	✗	✗
RESTCONF	✗	✓

🙏 Thanks for the input! ☺️