HSM and Thales Basics using the Thales Simulator

Single Double Triple Length Keys

Single length key = 8 bytes = 64 bits = 16 hex chars

Double Length key = 16 bytes = 128 bits = 32 hex chars

Triple Length key = 32 bytes = 256 bits = 64 hex chars

Simple java code test using a single length key to understand the above

```
String hex = "0909090909090909";
byte[] hex_in_bytes = ISOUtil.hex2byte(hex);
```

hex	"0909090909090909" (id=19)
hex_in_bytes	(id=23)
[0]	9 [0x9] [^I (TAB)]
[1]	9 [0x9] [^I (TAB)]
[2]	9 [0x9] [^I (TAB)]
[3]	9 [0x9] [^I (TAB)]
[4]	9 [0x9] [^I (TAB)]
[5]	9 [0x9] [^I (TAB)]
[6]	9 [0x9] [^I (TAB)]
[7]	9 [0x9] [^I (TAB)]

As you can see the byte array has got 8 bytes.

Des keys need to have odd parity.

What is this parity?
Each byte of the key has its last bit as parity.

So let's take byte 0 from above test.

Hex_in_bytes[0] = 9 =0x09 (remember these are hex string)

0x09 = 0000 1001 (in binary)

The **1** is treated as a parity bit

Now in order to make this odd parity the parity bit needs to have a value that causes the count of 1's in the string to become odd. There is just one 1 (excluding the parity bit) so parity bit needs to be 0 to make the count odd.

So the value changes to 0000 1000 = 0x08.

Now from a Des Key perspective the key value is 0808080808080808, so even if you provide a value of 0909090909090909 the key is actually 0808080808080808.

So really the des key = contents + Parity byte(s)

Single length key = 7 bytes + 1 byte of parity = 64 bits = 16 hex chars

Double Length key = 14 bytes + 2 byte of parity = 128 bits = 32 hex chars

Triple Length key = 28 bytes + 4 byte of parity = 256 bits = 64 hex chars

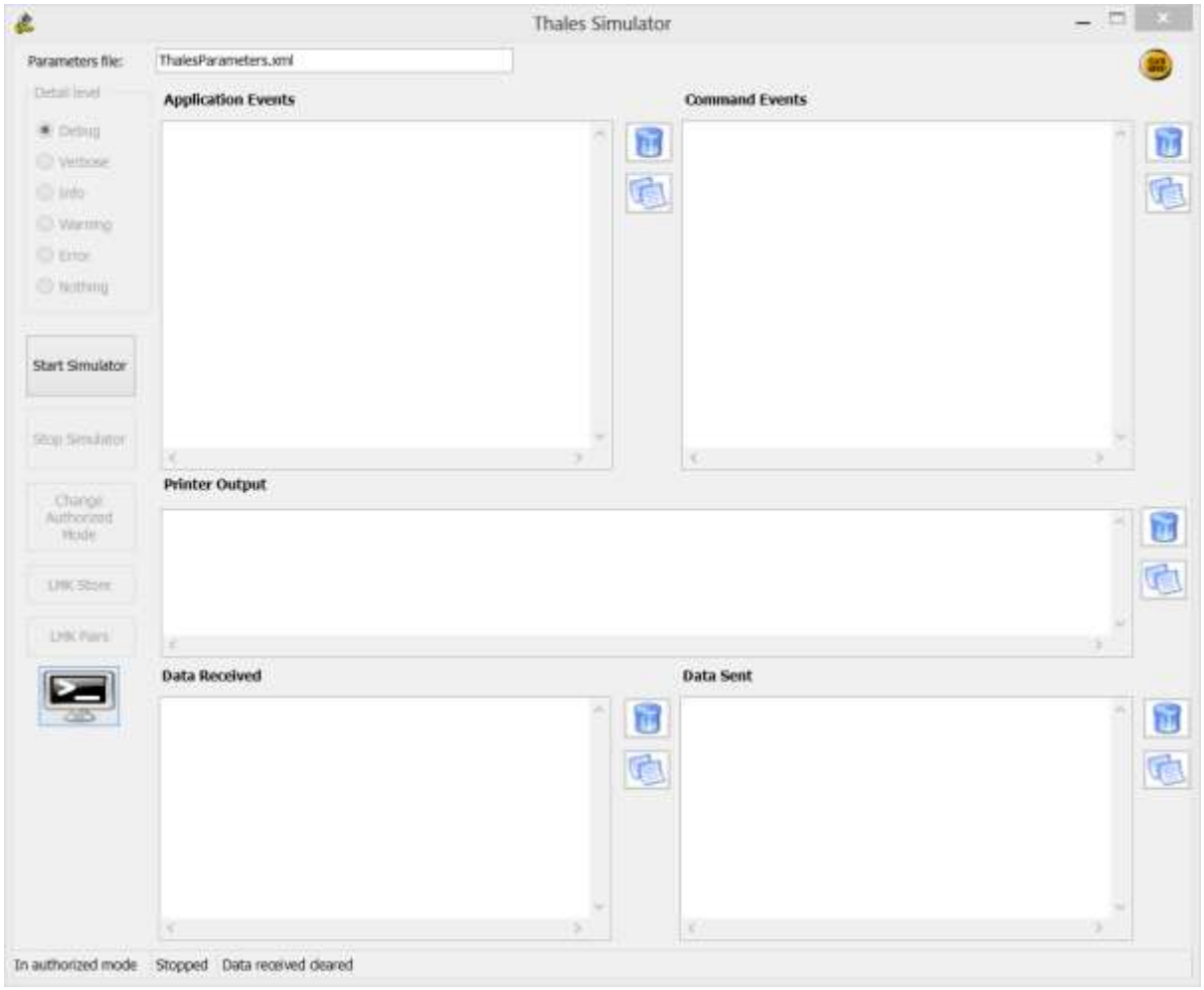
Now for some HSM fun.

Grab the thales simulator from the http://thalessim.codeplex.com/releases/view/88576 , pick the setup one and install it.. For linux guys you can try the mono one (no help from me for that).

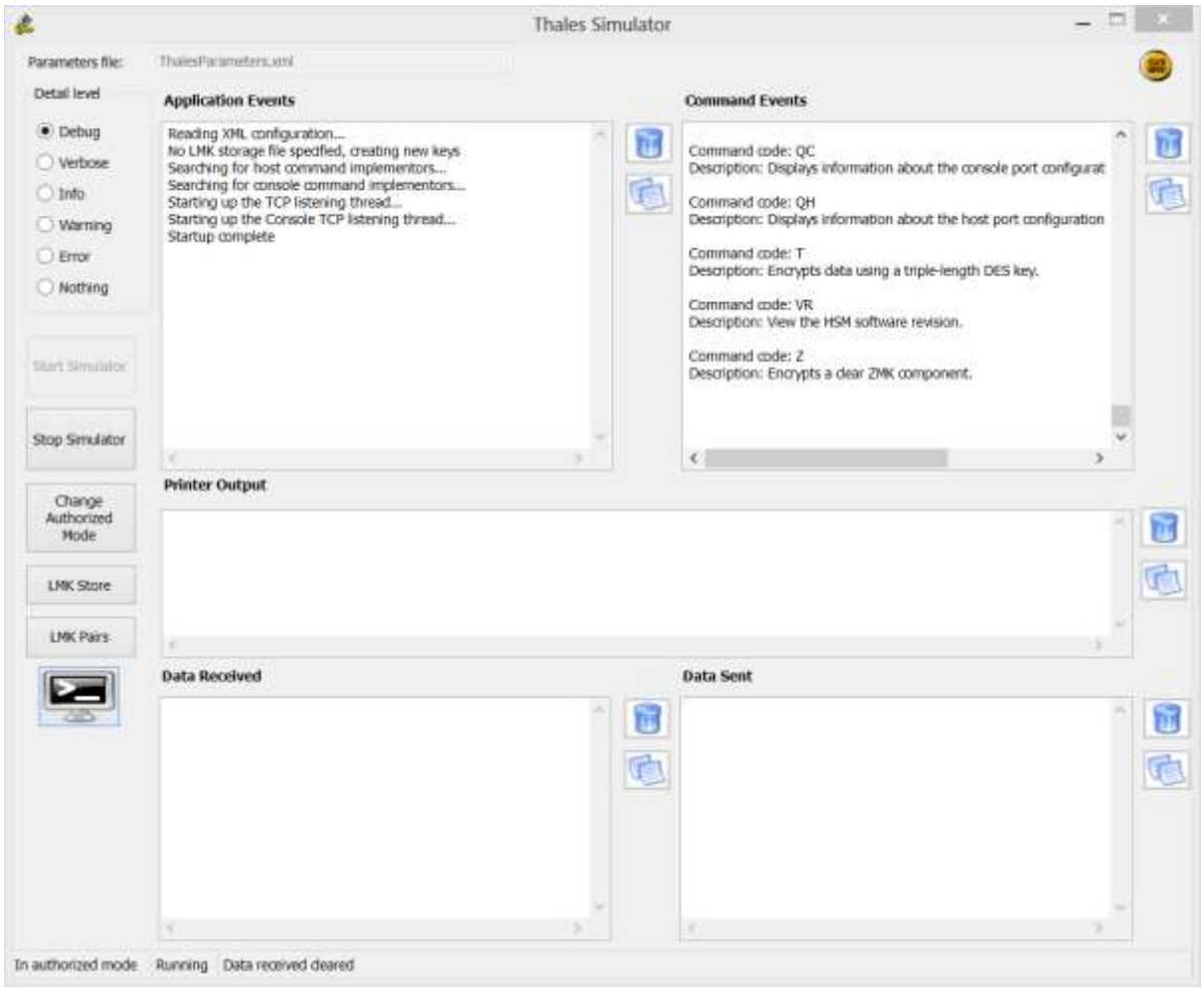
The goal is to

Generate a ZMK. A key shared between to networks.
Generate a ZPK. A key encrypted under ZMK and exchanged via online messages. The ZPK will be used to encrypt pin data (pin block to be more precise) and send it in the transaction request. Since this is symmetric encryption and the 2 entitles share the zpk decryption/translation is possible.

Fire Up the simulator



Click the start simulator.



Now the simulator has started and listening on port 9998.

This is configurable in the ThalesParameters.xml file found in the install directory of the sim. The property is <Port value="**9998**" /> .

Before we go an further lets just understand the key and how to generate it using the Simulator. The simulator is very close to how a real Thales works.

From the thales spec look at the key type table [section 3.1]

Consider I want to work with a **ZMK**. From the above table I gather the following.

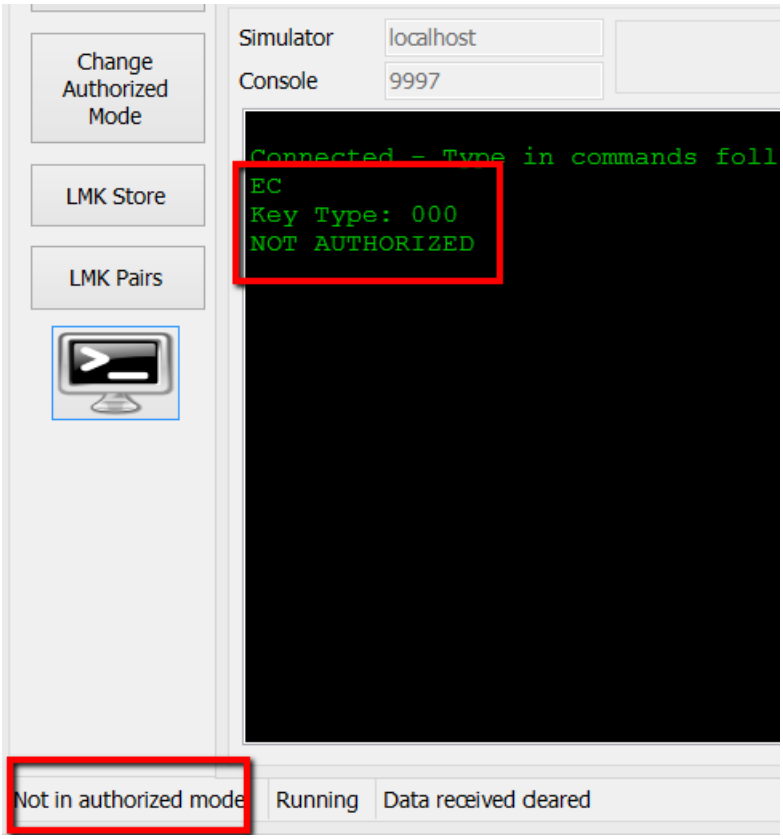
ZMK has a key type of 000 and 001. This is a parameter that the request messages to hsm needs.

The 3 digit key type = (variant value in x axis) + lmk code.

More on variant and lmk code later.

When do I use a 000 and when do I use a 001?
Under each variant there are 3 operations G: Generate , E:Export , I :Import
Against each LMK code for the key there is a condition U and A.
U: HSM does not need to be in authorised state.
A: HSM needs to be in the authorized state

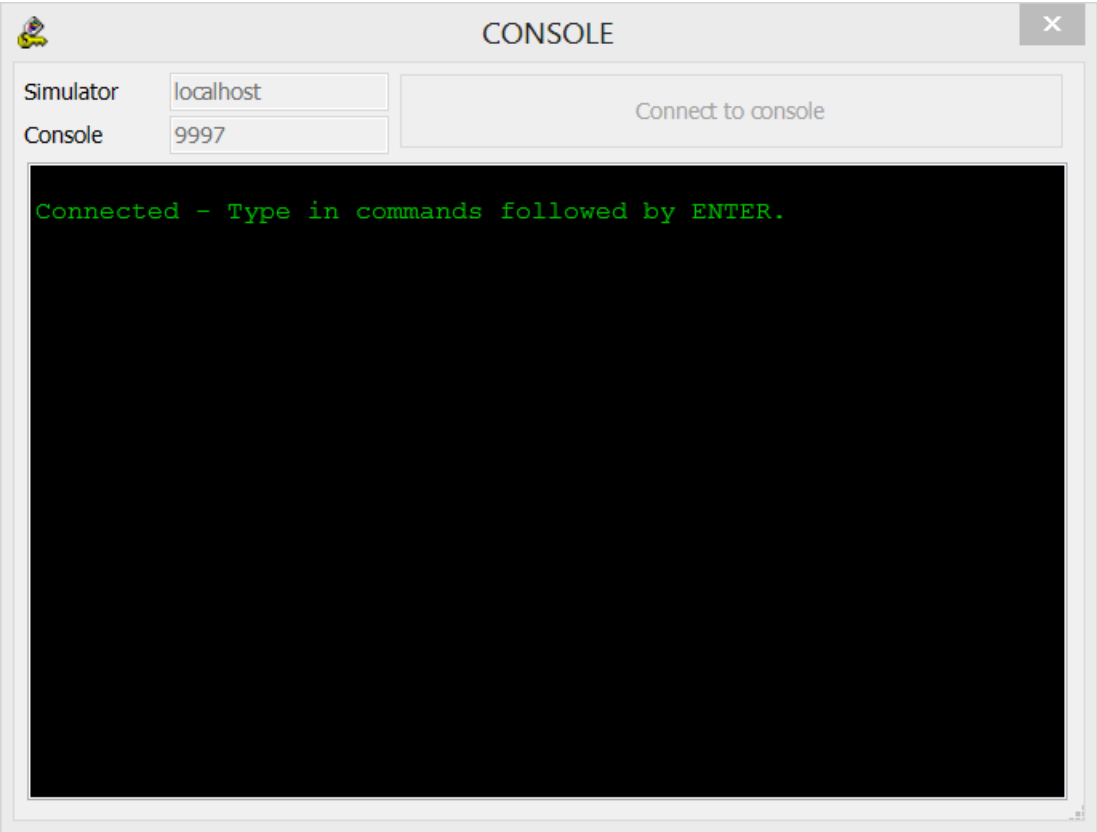
So what is this authorised state?
Its like your admin password, in case of HSM there are physical keys that need to be used to get into this authorised state. From a logistical perspective this is for security to allow certain commands based on your access privileges.



So based on what state you have the HSM in you can use the appropriate key types.

For this example I am going to be in authorized state and use 000. To get into authorised state use the “Change Authorized Mode” and see the status at the bottom change to authorized.

Then click the console button and in the dialog that pops up click connect.

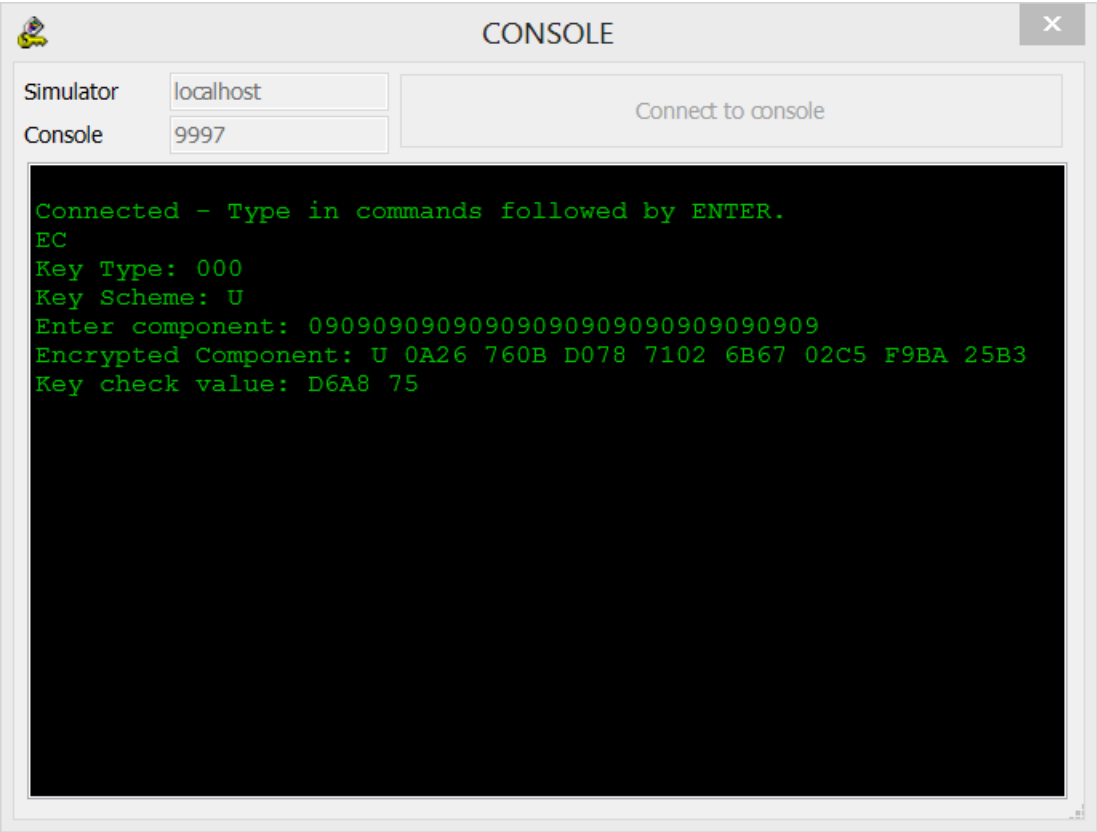


Now we are ready to generate keys.

To generate key I am going to use the EC command here. This allows me to enter a clear key and returns a cryptogram. In real life you would use a GC command so that it generates a random key and gives you an encrypted value.
Type EC in the console and hit enter.
The values I enter are here in text format that you can copy and paste.

EC
Key Type: 000
Key Scheme: U
Enter component: 09090909090909090909090909090909
Encrypted Component: U 0A26 760B D078 7102 6B67 02C5 F9BA 25B3
Key check value: D6A8 75

I have provided a 32 hex char string for clear key, this makes it a double length key and the scheme used is U.
The scheme values are obtained from the spec see Thales spec section 3.2

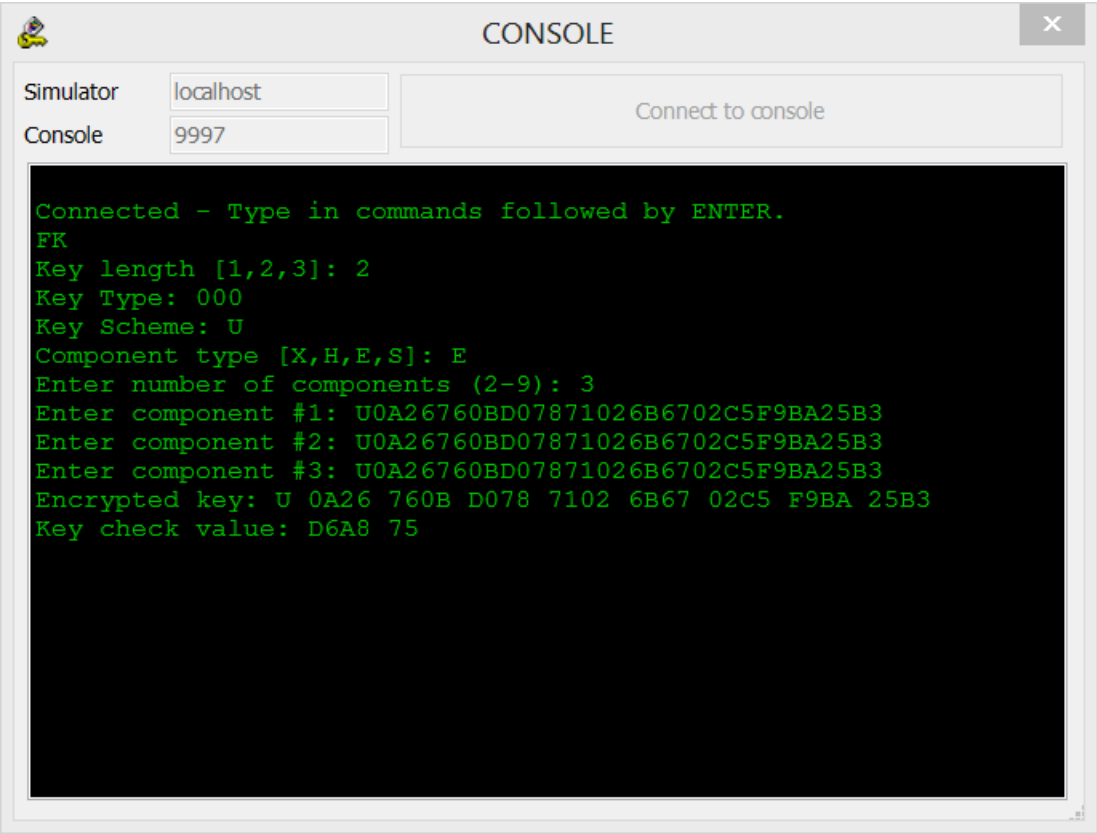


Usually what happens is one person per component runs the command and keeps the component secure.

For our example I will have 3 components and to keep it simple I will use the same component (never do this for production). I will get into some Boolean logic later based on the decision made to use the same components.

Now I have the components and I need to form a key from the component.

We will use the FK command.



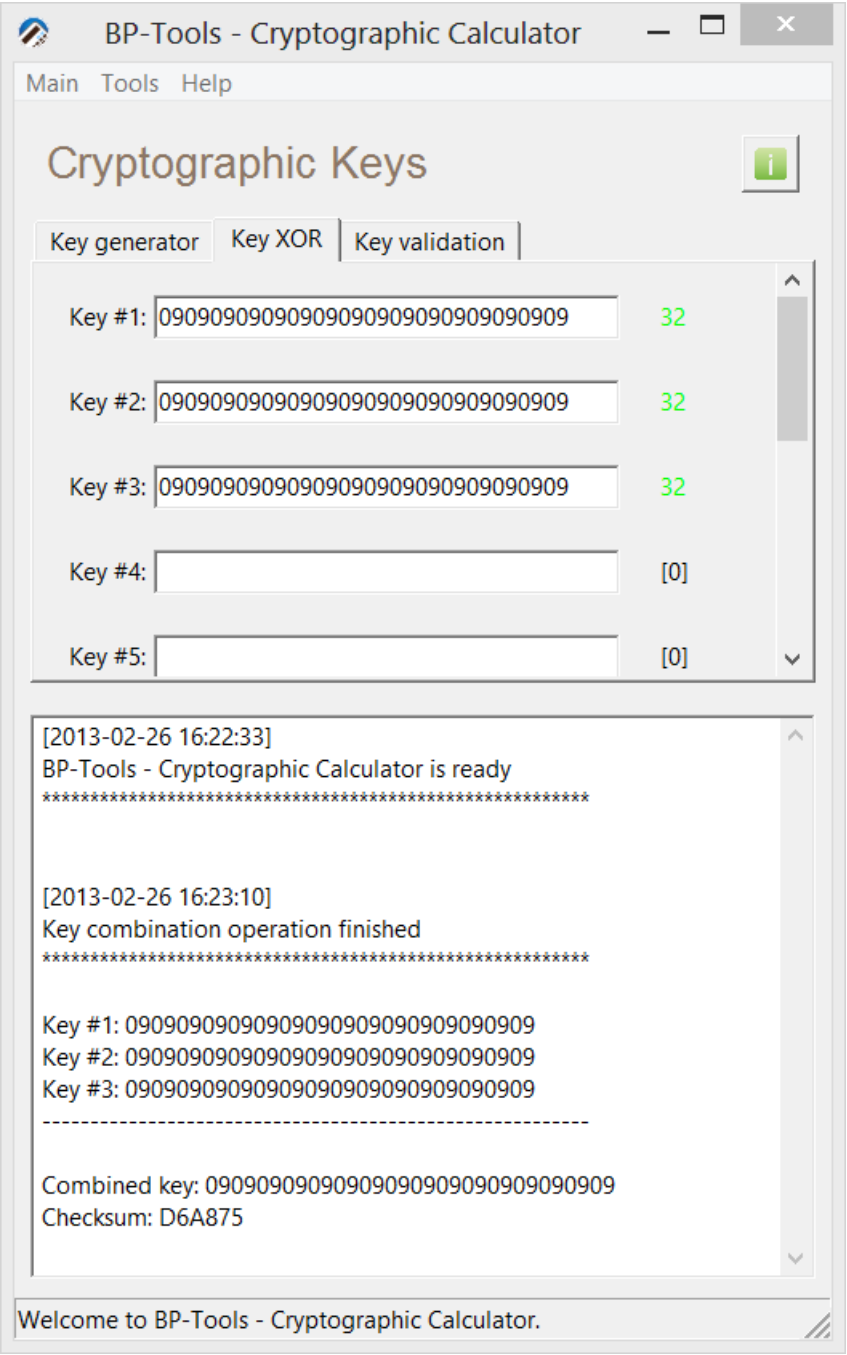
FK
Key length [1,2,3]: 2
Key Type: 000
Key Scheme: U
Component type [X,H,E,S]: E
Enter number of components (2-9): 3
Enter component #1: U0A26760BD07871026B6702C5F9BA25B3
Enter component #2: U0A26760BD07871026B6702C5F9BA25B3
Enter component #3: U0A26760BD07871026B6702C5F9BA25B3
Encrypted key: U 0A26 760B D078 7102 6B67 02C5 F9BA 25B3
Key check value: D6A8 75

Key Length is single/double/tripe. I selected double.
Key Type: We are still in the process of getting a key type 000
Key Scheme indicates a double length key.
Component Type X = Clear XOR, H = Clear Half or Third Key, E = Encrypted, S = Smartcard
Here we have chosen to use the encrypted values we obtained from the EC command, but remember to prepend it with a U (I don't know why). Alternatively you could have used the X and entered 09's for the 3 components (no U required here). Try it out both ways you will get the same result.

You should be wondering now why did we go through this process when the end result encrypted key is the same as the encrypted key with the EC command.

Well the attempt here is to engage you in some Boolean fun.
The components that you entered are xor'd with each other, bit wise XOR work as follows
1 XOR 1 = 0 and 0 XOR 1 = 1. So if you XOR a value with itself you will get a 0 and if you XOR that with the value again it will be the value.

If I had decided to do this with 2 components I would have got a clear key of 0's. HSMs generaly don't like a ke that has a value of all 0's (woud you use that as your pin ☺)

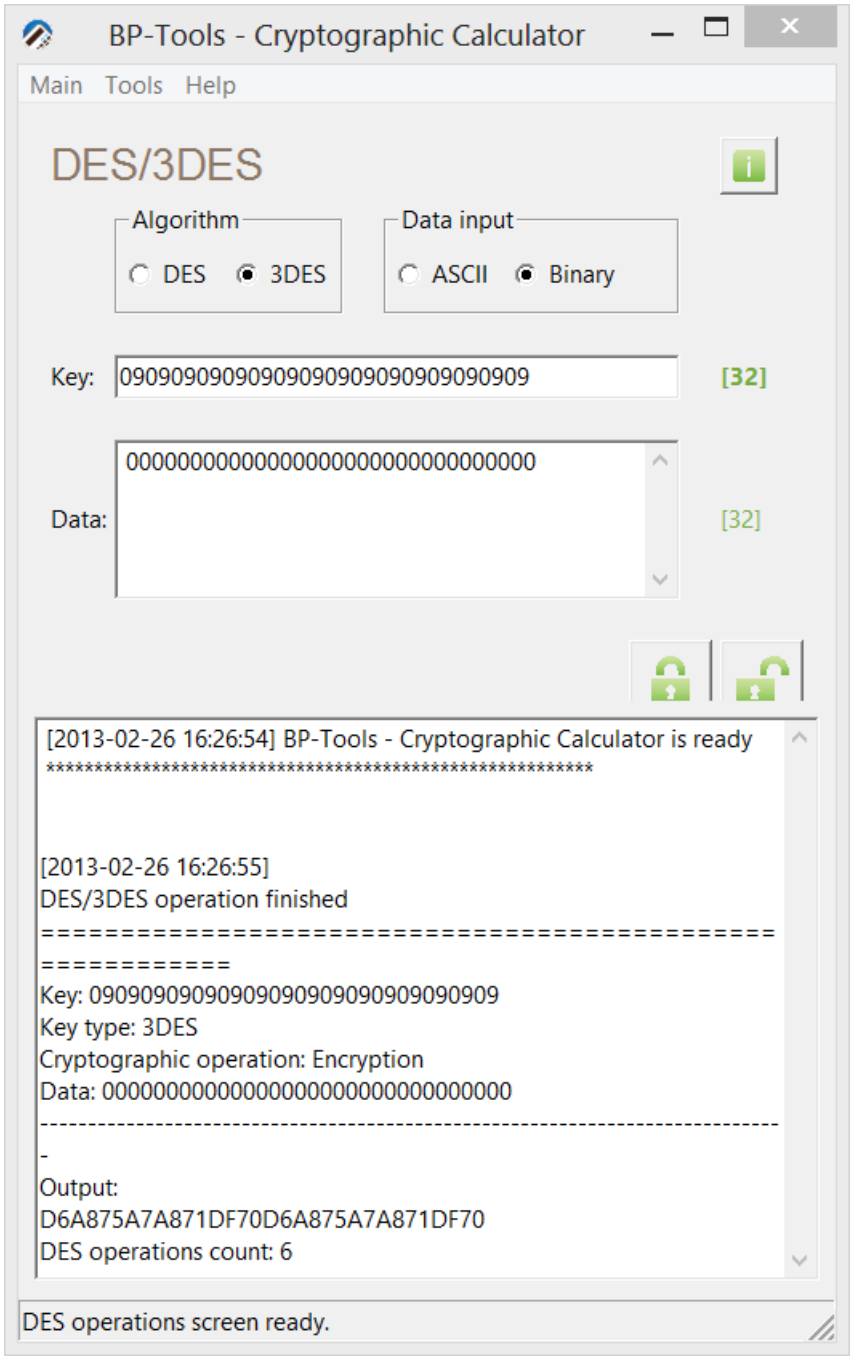


The above is just another tool I have, is used here for demoing, you can use your calculators in hex mode and do the xor operations or just take my word for it.

Note the check digits are also identical.

So how are the check digits calculated?

Encrypt the clear key with a value of all 0's



Tool shows the check value generation.

So now we have an encrypted ZMK with a value of 0A26 760B D078 7102 6B67 02C5 F9BA 25B3 and this needs to be shared with the other entity so that when pin keys (ZPK) are generated encrypted under this ZMK can be decrypted by the other entity and used for encrypting and decrypting pin data.

We have an encrypted ZMK, but what is it encrypted with?

The HSM has these LMK key pairs that stored internally in the HSM and secure, we don't have access to those. From the Key Table we know that while generating keys the LMK Pair is used to do the operation. So the ZMK that we generated is encrypted under the LMK (LMK key pair 04-05 to be specific). From a sim perspective its stored in the file called LMKSTORAGE.TXT in the install directory. It gets generated every time with fixed values. You can provide our file and reference it in the config file shared earlier.

We might as well touch upon the variant bit that I left out earlier. The variant is a constant hex value that is used to XOR certain bytes in the input clear key. This just gives added security. I don't know the internal working of this yet. The ThaleSim has its source code available and details of this can be obtained from there.

So now moving onto the usage of the ZMK to generate a key and do a key exchange.

For this I am using the XYZ project where my client simulator has the ZMK, it uses the ThalesAdaptor code to generate a ZPK using the A0 command. The switch uses the A6 command to import the key provided.

Here is a snippet of the simulator deploy file that gets a new ZPK from the HSM, populates a field and sends it. You can see the ZMK value is initialized to the value we got previously from the ZMK creation exercise.

```
    String getZPK(){
        ThalesAdapter securityModule = (ThalesAdapter) NameRegistrar.get("Thales");
        FSDMsg response  = securityModule.generateZPKA0(ZMK);
        return response.get("key-under-zmk")+ response.get("check-value");

    }
  FIELD18="6011";
  ZMK = "0A26760BD07871026B6702C5F9BA25B3";

  </init>


<test-suite>

  <path>cfgtest/pulse/</path>


<test file="Network/Network" count="1" continue="yes" name="Issuer Key Exchange">
    <init>
      MTI = "0800" ; FIELD70="161"; FIELD96=null; FIELD125=getZPK();
    </init>
  </test>
```

And this is the actual test case file, the ! variables get replaced b values defined in the test init section or call the functions directly.

```
<field id="0" value="!MTI"/>
<field id="7" value="! get_dateMMddhhmmss()"/>
<field id="11" value="! get_stan()"/>
<field id="70" value="!FIELD70"/>
<field id="96" value="50415353574F5244" type = "binary" />
<field id="125" value="!FIELD125"/>
```

The A0 command performed by the sim

```
<log realm="com.ols.security.thales.ThalesAdapter" at="Tue Feb 26 17:00:39 IST 2013.745" lifespan="91ms">
  <trace>
    <fsdmsg schema='file:cfg/hsm-base'>
      command: 'A0'
      mode: '1'
      key-type: '001'
      key-scheme-key-under-lmk: 'U'
      scheme-zmk-or-tmk: 'U'
      key-zmk-or-tmk: '0A26760BD07871026B6702C5F9BA25B3'
      key-scheme-key-under-zmk-or-tmk: 'X'
    </fsdmsg>
     request: 'A01001UU0A26760BD07871026B6702C5F9BA25B3X'
    response: 'A100U4ED6133910003A67D072A808D01CA05FX3C02215B81C1552D0D8CDB2C2946A2E9B2BE50'
     elapsed: 89ms
    <fsdmsg schema='file:cfg/hsm-resp-base'>
      response: 'A1'
      error: '00'
      scheme-key-under-lmk: 'U'
      key-under-lmk: '4ED6133910003A67D072A808D01CA05F'
      scheme-key-under-zmk: 'X'
      key-under-zmk: '3C02215B81C1552D0D8CDB2C2946A2E9'
      check-value: 'B2BE50'
    </fsdmsg>
  </trace>
</log>
```

The A6 performed by the switch to import.

```
<log realm="com.ols.security.thales.ThalesAdapter" at="Tue Feb 26 17:00:39 IST 2013.806" lifespan="35ms">
  <trace>
    <fsdmsg schema='file:cfg/hsm-base'>
      command: 'A6'
      key-type: '001'
      zmk-scheme: 'U'
      zmk: '0A26760BD07871026B6702C5F9BA25B3'
      import-key-scheme: 'X'
      key-to-import-under-zmk: '3C02215B81C1552D0D8CDB2C2946A2E9'
      scheme-key-encrpt-under-lmk: 'U'
    </fsdmsg>
     request: 'A6001U0A26760BD07871026B6702C5F9BA25B3X3C02215B81C1552D0D8CDB2C2946A2E9U'
    response: 'A700U4ED6133910003A67D072A808D01CA05FB2BE50'
     elapsed: 31ms
    <fsdmsg schema='file:cfg/hsm-resp-base'>
      response: 'A7'
      error: '00'
      scheme-key-under-lmk: 'U'
      key-under-lmk: '4ED6133910003A67D072A808D01CA05F'
      check-value: 'B2BE50'
    </fsdmsg>
  </trace>
</log>
```

How do I know this has succeeded?

The check values of the keys match in the A0 and A6.
The ZPK under the LMK match with what was generated and when it was imported.

If the sim used a different hsm the key under LMK would not have matched as each HSM has its own unique LMKs , but as long the check values match means the key in the used for sending has been successful imported and since the check value is based on the clear key value we know the encrypted key was successfully decrypted by the HSM and check value calculated.