# 背景知识

## IVT

Ivt(Image Vector Table), 用来指示image的架构的一段数据。

ROM解析IVT后，获取相关的boot信息后进行boot。例如是否为non-xip boot，image的长度，csf文件位置，image的入口entry等等信息。

**IVT data structure**

|  |  |  |
| --- | --- | --- |
| **Offset** | **Field** | **Description** |
| 0x00 - 0x03 | Header | **D1 00 20 4x**   |  |  | | --- | --- | | Byte 0 | tag, fixed to 0xD1 | | Byte 1,2 | length, bit endian format containing the overall length of the IVT in bytes, fixed to 0x00, 0x20 | | Byte 3 | version, valid values: 0x40, 0x41, 0x42, 0x43 | |
| 0x04 - 0x07 | **Entry** | Absolute address of the first instruction to execute from the image, or the vector address of the image |
| 0x08 - 0x0b | Reserved1 | Reserved for future use, set to 0 |
| 0x0c - 0x0f | DCD | Absolute address of the image DCD. It is optional, so this field can be set to NULL if no DCD is required. |
| 0x10 - 0x13 | **boot\_data** | Absolute address of the boot data |
| 0x14 - 0x17 | Self | Absolute address of the IVT. |
| 0x18 - 0x1b | csf | Absolute address of the Command Sequence File (CSF) used by the HAB library |
| 0x1c - 0x1f | Reserved2 | Reserved, set to 0 |

**Boot Data structure**

|  |  |  |
| --- | --- | --- |
| **Offset** | **Field** | **Description** |
| 0x00-0x03 | **Start** | Absolute address of the bootable image |
| 0x04-0x07 | Length | Size of the bootable image |
| 0x08-0x0b | Plugin flag | Plugin flag, set to 0 if it is a normal boot image |

Machine generated alternative text:
00 
evice 
Boot Device Memory 
ase 
mage Vector 
Tab e offset 
Initial 
Load Region 
Boot Data 
header 
entry 
reservedl 
boot data 
start 
ength 
p ugin flag 
OCD 
Application 
CSF. certificates and signatures 
mage Vector 
Tab e offset 
Boot Data 
Dest. Memory 
header 
entry 
reservedl 
boot data 
ength 
plugin flag 
Application 
CSF. certificates and signatures 
Figure 8-15. Image Vector Table 

## RSA算法介绍

<https://www.cnblogs.com/richard-xiong/p/9923283.html>

RSA算法的的原理简单来说就是将一个很大的质数M，很难分解为两个质数a,b,并且a\*b=M

### 公钥私钥的计算方式

公钥（n, e）

私钥（n, d）

n:

两个不相等的质数p，q，p\*q=n

e: (实际应用中通常都是65537，openssl生成的文件使用的就是这个值)

任选一个数e与φ(n)互质，且1<e<φ(n)

n的欧拉函数φ(n)=(p-1)\*(q-1),计算出来的值就是不大于n且与n互质的整数个数

d:

e对于φ(n)的模反元素d，计算公式d\*e≡1(mod(n))。

所谓["模反元素"](http://zh.wikipedia.org/wiki/%E6%A8%A1%E5%8F%8D%E5%85%83%E7%B4%A0)就是指有一个整数d，可以使得ed被φ(n)除的余数为1。

## Openssl使用详解

Openssl是用来生成证书跟公钥的一个工具。

如下介绍关于openssl一些知识。包括秘钥、证书的生成，秘钥解析，签名认证等

<https://www.cnblogs.com/guogangj/p/4118605.html>

<https://www.madboa.com/geek/openssl/#how-do-i-generate-a-public-rsa-key>

**SSL** - Secure Sockets Layer,现在应该叫"TLS",但由于习惯问题,我们还是叫"SSL"比较多。http协议默认情况下是不加密内容的,这样就很可能在内容传播的时候被别人监听到,对于安全性要求较高的场合,必须要加密,https就是带加密的http协议,而https的加密是基于SSL的,它执行的是一个比较下层的加密,也就是说,在加密前,你的服务器程序在干嘛,加密后也一样在干嘛,不用动,这个加密对用户和开发者来说都是透明的.

**OpenSSL** - 简单地说,OpenSSL是SSL的一个实现,SSL只是一种规范.理论上来说,SSL这种规范是安全的,目前的技术水平很难破解,但SSL的实现就可能有些漏洞,如著名的"心脏出血"

### 证书标准

**X.509** - 这是一种证书标准,主要定义了证书中应该包含哪些内容.其详情可以参考RFC5280,SSL使用的就是这种证书标准.

### 编码格式

同样的X.509证书,可能有不同的编码格式,目前有以下两种编码格式.

**PEM** - Privacy Enhanced Mail,打开看文本格式,以"-----BEGIN..."开头, "-----END..."结尾,内容是BASE64编码.  
查看PEM格式证书的信息:openssl x509 -in certificate.pem -text -noout（使用这条命令可以获取公钥信息）  
Apache和\*NIX服务器偏向于使用这种编码格式.

**DER** - Distinguished Encoding Rules,打开看是二进制格式,不可读.  
查看DER格式证书的信息:openssl x509 -in certificate.der **-inform der** -text -noout  
Java和Windows服务器偏向于使用这种编码格式.

**KEY** - 通常用来存放一个公钥或者私钥,并非X.509证书,编码同样的,可能是PEM,也可能是DER.  
查看KEY的办法:openssl rsa -in mykey.key -text -noout  
如果是DER格式的话,同理应该这样了:openssl rsa -in mykey.key -text -noout **-inform der**

### 文件拓展名

CRT – 应该是certificate的缩写，表示证书

CER – 证书

Key – 秘钥

CSR – 证书签名请求

PFX/P12 – predecessor of PKCS#12 包含证书跟私钥的文件

X509，PKCS12的区别：

<https://blog.csdn.net/caomiao2006/article/details/23677565>

x509，公钥证书，只有公钥。  
p7，签名或加密。可以往里面塞x509，同时没有签名或加密内容。

p12，含有私钥，同时可以有公钥，有口令保护。

X509是证书的标准，只要是证书，都是X509格式。  
P7和P12是两个实现规范，P7是数字信封，P12是带有私钥的证书规范。

生成pfx的命令类似这样:

openssl pkcs12 -export -in certificate.crt -inkey privateKey.key -out certificate.pfx -certfile CACert.crt

实质上就是将证书跟私钥整合在一起，形成一个pfx文件

### 生成私钥

Openssl生成私钥，证书等用法示例：

<https://www.cnblogs.com/littleatp/p/5878763.html>

生成4096长度的私钥，生成的私钥会包含公钥信息

openssl genrsa -out private4096.pem 4096

查看私钥信息(n,e,d,p,q等)：

openssl rsa -in ca0\_privatekey\_rsa4096.pem -text -noout

N,e,p,q详解：

<http://blog.sina.com.cn/s/blog_4fcd1ea30100yh4s.html>

输出到文件

openssl rsa -in ca0\_privatekey\_rsa4096.pem -text -out private\_key.txt

注：

-text会解析传入的pem文件，输出n,e,p,q等信息。如果私钥的格式是der则增加命令 -inform der即可

RSAPrivateKey ::= SEQUENCE {

      version           Version,

      modulus           INTEGER,  -- n

      publicExponent    INTEGER,  -- e

      privateExponent   INTEGER,  -- d

      prime1            INTEGER,  -- p

      prime2            INTEGER,  -- q

      exponent1         INTEGER,  -- d mod (p-1)

      exponent2         INTEGER,  -- d mod (q-1)

      coefficient       INTEGER,  -- (inverse of q) mod p

      otherPrimeInfos   OtherPrimeInfos OPTIONAL

  }

### 私钥生成公钥

openssl rsa -in private\_rsa.pem -pubout -out public\_rsa.pem

查看公钥信息

openssl rsa -in public.pem -pubin -text -noout

### 生成证书

以二级证书为例，顶层证书为自签名CA证书。二级证书为非CA证书，使用CA证书以及CA公钥对其签名。二级证书对应的私钥用于签名image

#### 生成CA证书请求文件csr：

openssl genrsa -out ca0\_privatekey\_rsa4096.pem 4096

openssl req -new -key ca0\_privatekey\_rsa\_4096.pem -out ca0\_v3.csr -extensions v3\_ca -subj "/C=CN/ST=SZ/L=SZ/O=NXP/OU=NXP/"

1.1 查看csr：

openssl req -text -noout -in ca0\_v3.csr

1.2 使用私钥对请求签发csr签发CA x509证书（自签名CA）

生成DER格式证书：

openssl x509 -req -days 365 -in ca0\_v3.csr -signkey ca0\_privatekey\_rsa4096.pem -sha256 -outform der -out self\_signed\_cert0\_ca.der.crt -extfile v3\_ca.ext

生成PEM格式证书（用于签发下级证书）：

openssl x509 -req -days 365 -in ca0\_v3.csr -signkey ca0\_privatekey\_rsa4096.pem -sha256 -out self\_signed\_cert0\_ca.pem.crt -extfile v3\_ca.ext

#### 使用CA证书（PEM格式）跟CA私钥对请求签发X509证书（非CA）

2.1生成私钥用于签名image

openssl genrsa -out ca1\_privatekey\_rsa4096.pem 4096

2.2生成证书请求文件

openssl req -new -key ca1\_privatekey\_rsa4096.pem -out ca1\_v3.csr -extensions v3\_ca -subj "/C=CN/ST=SZ/L=SZ/O=NXP/OU=NXP/"

2.3使用CA证书跟私钥对请求签发证书进行签发，生成der格式证书

openssl x509 -req -days 365 -in ca1\_v3.csr -CA self\_signed\_cert0\_ca.pem.crt -CAkey ca0\_privatekey\_rsa4096.pem -set\_serial 01 -sha256 -outform der -out signed\_cert1\_noca.der.crt -extfile v3\_noca.ext

#### 查看证书公钥跟签名

openssl x509 -in certificate.pem -text -noout

### 私钥签名

<https://blog.csdn.net/scuyxi/article/details/55002130>

openssl dgst -sign ca0\_privatekey\_rsa4096.pem -sha1 -out signature.txt 1.txt

### 公钥认证

openssl dgst -verify public.pem -sha1 -signature signature.txt 1.txt

## Fuse中RKTH计算步骤

Attention:

获取的公钥，第一次计算出来的hash都要以bin文件的方式存储。

Openssl或者python lib读取的是我们看到的数据的asc码，所以必须要以二进制的方式存储

使用HAB的脚本生成的私钥跟证书都是加密的。我们以不带HAB的产品为例，详细介绍

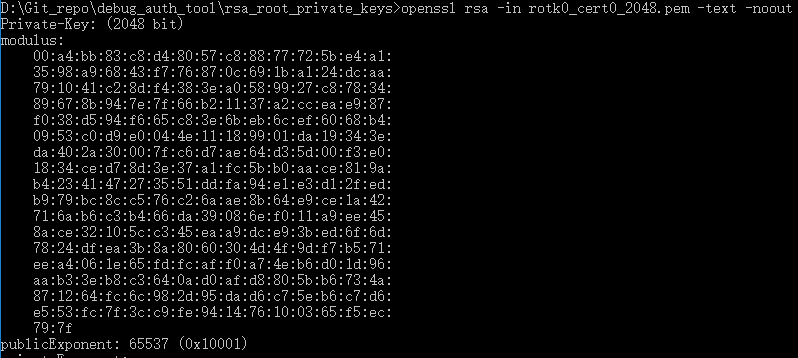
生成RKTH的步骤

首先，参考私钥生成办法，生成需要的根私钥，私钥中包含了公钥的信息，详细请参考[生成私钥](#_生成私钥)。一旦根私钥确定，无论证书有多少级，都不会影响RKTH。因为RKTH就是根公钥hash再hash的结果。

Machine generated alternative text:
RootCert
00tCert#1
RootCert#2
RootCert羽
RootPK
RootPK#1
RootPK#2
RootPK#3
SHA-2S6
SHA-2S6
SHA．256
SHA-2S6
RKHTable
Entry#0
Entry#1
SHA-256
Entry#2
Entry羽
RKTH

Machine generated alternative text:
EachentryinthetableisaSHA-256computedovertheconcatenationOfanRSApublickey'smodulusandexponent(modulus日exponent).Both
modulusandexponentmustbeinbigendianbyteorderwiththeminimumnumberOfbytesrequiredtorepresentthevalue.Forinstancean
exponentOf65537wouldberepresentedbya3-bytevalueOf[010001]，whileanexponentOf3wouldberepresentedbyasinglebyteOfthat
value.TheentireRKHtableisitselfhashedWIthSHA-256.ThisfinalhashisthenstoredintheRKTHdinIFR.
FO了1工n0，3：
O目
2RKH
LetM
LetE
LetRKH
LetRKTH
BE(Modulus刁
BE(Exponent
SHA256（RK_H
RKH

### 从私钥中提取公钥n,e



Module 就是n， publicExponen就是e

将n的第一个字符00，去除所有的字符： 。得到的数据长度为2048 bit

将e拓展为010001

所有字符都是16进制形式，新建一个.bin文件，进行存储。

对后续的三个公钥也使用同样的方式处理。



### 计算RKTH

使用sha256对如上四个bin文件分别计算hash，将得到的四个hash依次组成一个bin文件。4个公钥如果有没有用到的情况，其hash以全0进行替代。再进行一次sha256运算，得到RKTH。



### python计算RKTH



# HAB

## 概述

HAB是飞思卡尔secure team自主研发的一个关于认证，解密（HAB4后支持）的库。

它以lib的形式封装在ROM中。（本文所有案例都基于HAB4）

HAB boot的准备条件

1. 硬件上，我们需要烧写SRK fuse，同时开启secure boot fuse
2. 软件上，我们需要使用CST tool根据csf文件对csf命令以及image进行签名

Boot时，HAB会计算（两次hash）csf.bin中的srk table，与fuse中的SRK进行比对。

1. 通过后，将SRK 公钥存放到key store的slot 0处。然后逐条解析CSF命令
2. 使用SRK认证CSF证书，通过后将CSF公钥存放到key stote slot 1处.CSF公钥用来认证CSF命令
3. 使用SRK认证image证书，通过后将image的公钥存放到某个slot处，slot的index由[Install key]中的Target Index决定
4. 调用image的公钥认证image。[Authenticate Data]命令中需要指定image公钥的index。必须跟[Install key]中的Target Index一致

### Image是如何签名的？

CST处理CSF文件时，会根据image的证书名，从key文件夹中查找对应的image私钥对image进行签名

### Encrypt boot

CST处理CSF文件生成dek.bin跟encrypted image。Dek.bin用于解密image

使用OTPMK（OTP master key）对dek.bin生成blob.bin并存放于IVT指定位置

RT系列产品需要调用flashloader对dek.bin加密生成blob.bin。

Imx产品使用uboot加密dek.bin

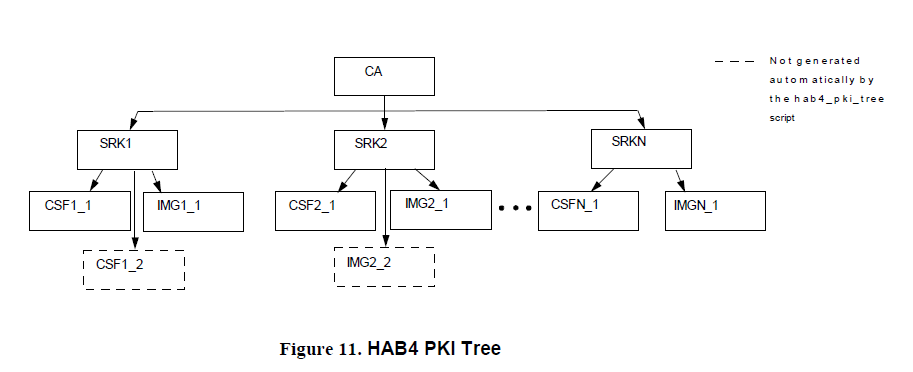
解密流程：

OTPMK解密blob,bin获取dekbin -> dek.bin解密image

OTPMK被硬件保护，只有CAAM能够访问OTPMK。这意味着blob的加密跟解密都必须在同一个的芯片上完成。即便是相同的dek，不同的芯片计算出来的blob也不一样。这就确保了blob的板子唯一性

## HAB4 PKI tree

PKI(Public key infrastructure)



PKI tree用来反映证书之间的关系

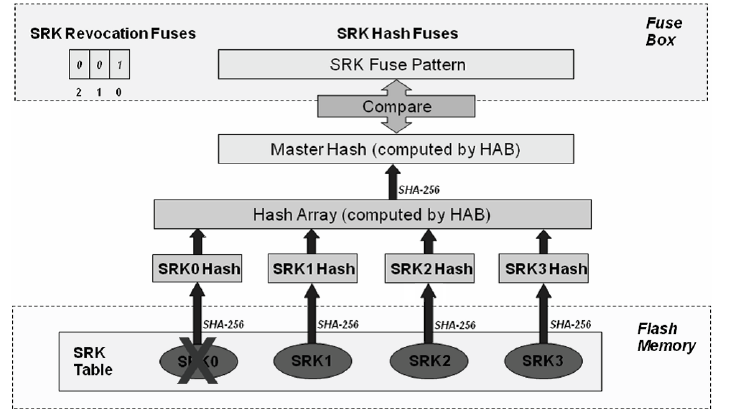
最顶层是CA，用来认证SRK。

接下来是SRK，SRK key经过hash后烧写到fuse中。

如何确保SRK的正确性？HAB对SRK table进行两次hash后跟SRK fuse进行比对。

CSF：SRK对CSF key进行认证，CSF key用来认证CSF命令

IMG：SRK对image key进行认证，image key对image进行认证

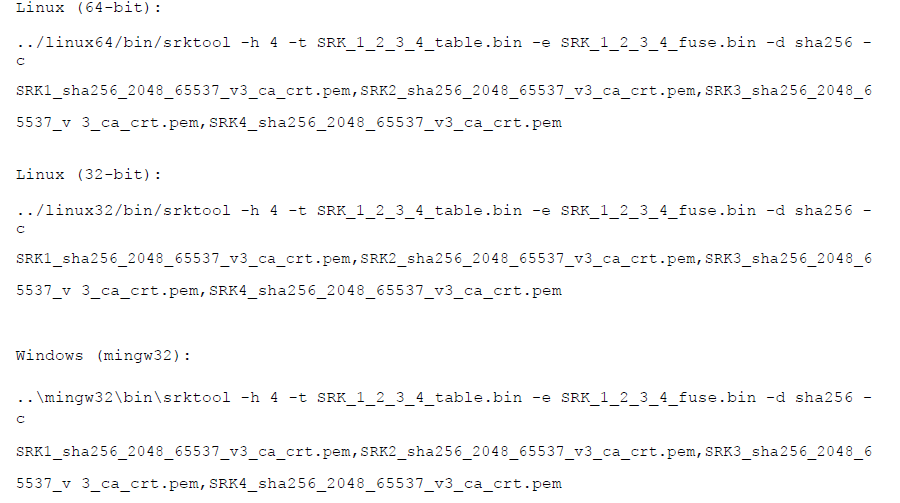


### 生成PKI tree

HAB执行hab4\_pki\_tree.sh脚本即可。

里面的私钥都是加密的，对应的密码存放在key\_pass.txt中

### 生成SRK



## CSF command

CSF（command sequence file）

CST tool根据CSF文件，生成csf.bin。csf.bin中包含CSF命令，数字签名，证书。

HAB以库的方式集成在ROM中，当检测到csf.bin后，解析csf.bin并执行相应的操作。

CSF文件中命令需要遵循的规则：

1. Header command必须在所有命令之前
2. Install SRK 必须在Install CSFK之前
3. Install CSFK必须在Authenticate CSF 之前
4. Install SRK, Install CSFK，Authenticate CSF只能出现一次
5. Authenticate Data中的语句Verification Index必须跟Install Key中的语句Target Index值相同
6. CSF.bin中的命令跟CSF文件中的命令顺序一致

### Header

[Header]

Version = 4.2 #HAB version，4.2以上支持encrypt boot

Hash Algorithm = sha256 #hash算法，支持SHA1, SHA256, SHA512

Engine = SW #ANY，SAHARA, RTIC, DCP, CAAM, SW

Engine Configuration = 0 #根据不同的engine有不同的配置

Certificate Format = X509 #证书格式WTLS, X509

Signature Format = CMS #签名格式PKCS1, CMS

### Install SRK

[Install SRK]

File = "../../keys/SRK\_1\_2\_3\_4\_table.bin" #HAB4 以上此处填写SRK table

Source Index = 0 #0-3. Fuse中如果revoke对应的index，ROM运行此条命令会报错

#HAB使用SRK hash fuse认证SRK

#Install SRK 命令有两个作用

#认证[install CSFK]跟[Install Key]中的public key

#将SRK public key存储在内部public key store的slot 0处

### Install CSFK

[Install CSFK]

File = "../../crts/CSF1\_1\_sha256\_2048\_65537\_v3\_usr\_crt.pem"

#用于认证跟install public key供Authenticate CSF命令使用

#HAB使用SRK认证CSF公钥

#HAB将CSFK public key存储在内部public key store的slot1处

### Authenticate CSF

[Authenticate CSF]

Engine = SW

Engine Configuration = 0

Signature Format = CMS

#HAB使用CSF public key认证正在运行的CSF。通过签名信息来认证这条命令

### Install Key

[Install Key]

File = "../../crts/IMG1\_1\_sha256\_2048\_65537\_v3\_usr\_crt.pem"

Verification Index = 0 #0,2..4

Target Index = 2 #2..4 Public key存储位置

#需要用户自己管理public key的存储位置，不能为0（SRK占用），1（CSFK占用）。在Target index中指定。安装的公钥用于[Authenticate Data]中数据的认证

#必须要在authenticate CSF命令之后

#HAB认证公钥，实质就是认证公钥的证书。使用之前安装好的key进行认证

CST tool发现错误的Target index会报如下错误

Invalid argument TargetIndex in command InstallKey

### Authenticate Data

[Authenticate Data]

Verification Index = 2 #此处需要跟install key中的Target Index相同。需要正确使用HAB key store中存在的public key对应的index

Engine = SW

Engine Configuration = 0

Blocks = 0x20000400 0x400 0x40 "IVT.bin",

0x20002000 0x2000 0x800 "IVT.bin"

#如果 verification Index跟install key中的Target Index不同，cst tool生成CSF.bin时会报如下错误

cst 23192 cygwin\_exception::open\_stackdumpfile: Dumping stack trace to cst.exe.stackdump