# Workshop on Public Key Infrastructure and Red Hat Certificate System

Asha Akkiangady, Huzaifa Sidhpurwala, Niranjan Mallapadi

## Contents

**1 Introduction to Public Key Cryptography** **2**
- 1.1 Encryption . . . 3
  - 1.1.1 Symmetric Encryption . . . 4
  - 1.1.2 Assymetric Encryption . . . 4
- 1.2 Message Digest . . . 5
  - 1.2.1 Message Authentication Code . . . 5
- 1.3 Algorithms . . . 5
- 1.4 Protocols . . . 8
  - 1.4.1 SSL . . . 8

**2 Introduction to Public Key Infrastructure** **11**
- 2.1 Common Terms used in PKI . . . 11
- 2.2 Detailed look on certificate/CRL . . . 12
  - 2.2.1 Certificates . . . 12
  - 2.2.2 Basic Certificate Fields . . . 13
  - 2.2.3 Extensions . . . 14
  - 2.2.4 Revocation . . . 19
  - 2.2.5 CRL Fields . . . 21
  - 2.2.6 CRL Extensions . . . 21
- 2.3 Mozilla NSS . . . 23
  - 2.3.1 Introduction . . . 23
  - 2.3.2 Architecture . . . 23
- 2.4 Exercises . . . 24

**3 Common Terms Used in RHCS** **29**

**4 Red Hat Certificate System** **30**
- 4.1 Overview . . . 30
- 4.2 Version . . . 31
  - 4.2.1 Older Versions . . . 31
  - 4.2.2 Current Version . . . 32
- 4.3 Certificate Manager . . . 32
  - 4.3.1 Introduction . . . 32
  - 4.3.2 Installation . . . 32
  - 4.3.3 Key Features . . . 33
  - 4.3.4 Architecture . . . 33
  - 4.3.5 Interfaces . . . 33

4.3.6 Features . . . 33
4.3.7 Instance Layout . . . 38
4.3.8 Exercises . . . 38
4.4 Key Recovery Authority . . . 38
4.4.1 Introduction . . . 38
4.4.2 Installation . . . 38
4.4.3 Installation Layout . . . 39
4.4.4 Key Features . . . 40
4.4.5 Architecture . . . 40
4.4.6 Interfaces . . . 40
4.4.7 Features . . . 41
4.4.8 Exercises . . . 43
4.5 Online Certificate Status Protocol . . . 43
4.5.1 Introduction . . . 43
4.5.2 Installation . . . 43
4.5.3 Interfaces . . . 43
4.5.4 Exercises . . . 44
4.6 SmartCard Management . . . 44
4.6.1 Introduction . . . 44
4.6.2 Architecture . . . 45
4.6.3 Common Terms Used with regard to Smartcard . . . 45
4.6.4 Token Key Service . . . 46
4.6.5 Token Processing Service . . . 46

# 1 Introduction to Public Key Cryptography

Before we start discussing about public key cryptography, we will in general discuss about how systems communicate and what are the various threat models that are associated with the communication medium and what are the tools to overcome them. [6]

**Example1:** The most common protocol used to communicate between 2 systems is TCP/IP. TCP/IP allows information to be sent from one system to another system directly or through many intermediate systems.

Below are some of the threat models associated with TCP/IP:

- **Eavesdropping:**

  Information remains intact, but it's privacy is compromised. For example: someone could retrieve the credit card number, record a sensitive conversation or intercept a classified information.

- **Tampering:**

  Information in transit is changed or replaced and then sent to the recipient. For example: Some one could alter an order of goods or change a persons resume.

- **Impersonation:**

  Information passes to a person who poses as intended recipient. Impersonation can take 2 forms:

– **Spoofing:**
A person can pretend to be someone else, For example, a person can pretend to have email address *joe@example.net*, or computer can identify itself as a site called www.example.net, when it is not. This type of impersonation is called spoofing.

– **Misrepresentation:**
A person or organization can misrepresent itself, For example, suppose the site *www.example.net* pretends to be a furniture store when it's just a site which accepts payments but never sends any goods.

**Example2:** Consider Alice , Bob and attacker are the parties , where alice and bob want to communicate to each other and Attacker is a threat to the communication. [6]

- Alice can send a postcard to bob to communicate, but this method is very weak as any random eavesdropper could read the postcard.
- Alice could write a letter and put in an envelope and send it to bob, but the attacker could open the envelope and read the letter, both confidentiality and integrity of the message is lost.
- Alice could seal the envelope with wax , but this method too is inefficient as Attacker could read the letter and seal it as it is without bob knowing that it was read by attacker
- Alice could write a letter and put it in a safe which has 2 keys and send one key before to Bob , and send the safe across to bob, this ensures confidentiality, integrity but practically this is not implementable. Considering the number of messages that has to be transferred , it's impractical to implement the mail safe method.

To mitigate above threat models, we will look in to cryptology as one of the tools of the trade.

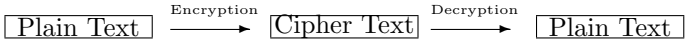
**Cryptology:** Cryptology is the theory of designing the various algorithms we use to provide security

**Cryptography:** Cryptography is the study of using these algorithms to secure systems and protocols.

## 1.1 Encryption

An encryption algorithm takes some data(called plaintext) and converts it to cipher-text under control of a key. cipher text contains random data which makes no sense without the key.

A key is a short random string (8-24 bytes):




When a message is encrypted and received , we cannot say if it's not tampered with. An encryption is strong when it can determine the number of possible keys. The attacker tries each key one at a time until he finds a key that produces a plausible decryption. The security of the algorithm should solely depend on the secrecy of the key. The algorithm should not need to be secret.

If the attacker knows the plaintext corresponding to the ciphertext it's called *"known plaintext attack"* .

An attack where attacker doesn't know the plaintext , it's called *"ciphertext-only attack"*. *Ex:* If the attacker knows that plaintext is ASCII , so any decryption which uses non-ascii characters must be using the wrong key.

### 1.1.1 Symmetric Encryption

When Sender and recipient share the same key(which must be kept secret) is referred to as *Symmetric Key Cryptograph* or also referred to as *Secret Key Cryptography* as opposed to *Public Key Cryptography*citation required?.

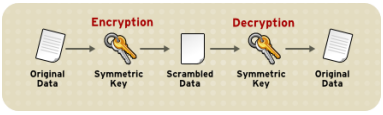



Figure 1: Symmetric Encryption [3]

### 1.1.2 Assymetric Encryption

- First started in stanford university by Whitfield Diffie and Martin Hellman
- The most commonly used implementation of PKC are based on algorithm based on algorithm patented by RSA data security.
- Each public key is published & private key is kept secret. Data encrypted with public key can be decrypted only with private key.
- In general, to send encrypted data to someone, we encrypt with public key & the person encrypt receiving the encrypted data decrypts with private key
- Compared to symmetric key encryption, public key encryption requires more computation & therefore not always appropriate for large amounts of data
- It is possible to use public-key encryption to send a symmetric key, which can then be used to encrypt additional data.

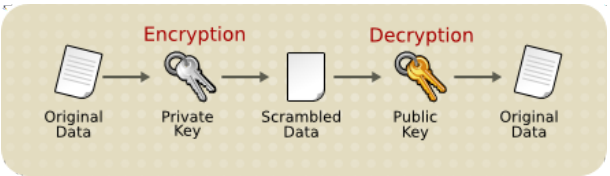



Figure 2: Assymetric Encryption

- Reverse of the above figure also happens i.e. encrypt with private key and decrypt with public-key. But not useful for sensitive information

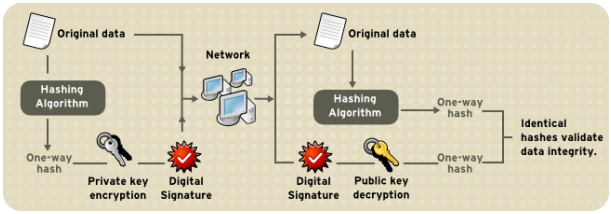



Figure 3: Digital Signature

- There's a problem with the above method, how would the parties get each other's public-key ? If we send the keys through electronically, then the attacker an tamper, while they are in transit to the receiver.
- When the 2 parties want to communicate, the attacker can intercept the keys & instead send his own key to each other, thus each party encrypts to him and & he re-encrypts it to the real recipient. This is called *man-in-the-middle attack*

## 1.2 Message Digest

A message digest is simply a function that takes as an input an arbitrary message and outputs a fixed length string which is characteristic of the message. The important property here is irreversibility. It's extremely difficult to compute a message from the given digest. Property of the message digest:

- For a digest to be secure, it must be difficult to generate any of the message that digests to the same value. You have to search a message space of proportional size of the digest in order to find a matching message text
- It should be difficult to produce 2 messages M and M' such that they have the same digest. This property is called collision-resistance. It turns out that the strength of any message digest against finding collision is only half the size of the digest., so a 128-bit digest is only 64 bits strong against collisions.

### 1.2.1 Message Authentication Code

Consider *Alice* and *Bob* share a key and Alice wants to send a message to *Bob*. The message can be encrypted, and send it across to *Bob*, but we are not sure that the encrypted message would be tampered and also not sure if *Alice* was the one who sent the encrypted message. So we use a new tool called *MAC* . *MAC* is a digest algorithm, but with a key, So the MAC is dependent on both the key and the message being MACed.

## 1.3 Algorithms

- **RSA**
  - RSA is the public-key algorithm widely used in Public-key Cryptography

- Invented by Ron-Rivest, Adi Shamir and Len Adelman (RSA). Each user has a public-key and a private-key
- The public-key can be freely distributed and the private-key must be kept secret.
- Brief Explanation:
  * Generate 2 prime numbers (3, 17), call them p and q
  * n = modulus(p * q)
  * p and q are kept secret
  * Difficulty is in factoring m, so that we get p and q
  * Take another number "e" which is called public exponent, which is a prime number , it's usually small prime numers 3, 17 or 65537
  * Compute d = $e^{-1} mod((p-1)(q-1))$
  * Public Key = (e, n)
  * Private Key = (d, n)
  * Message M is encrypted with public key and decrypted with private key
  * RSA assumes that M is a number, So we need a convention to convert strings in to numbers.

- **DSA**
  - National Institute of standards and technology (NIST) published the Digital signature algorithm in the Digital signature standard.
  - Compared to RSA where it can be used for both encryption and digital signature
  - In DSA the signature generation is faster than signature verification

- **RC4**
  - RC4 is a stream cipher.
  - Assume we have a function (f) which produces 1 byte of data. This output is called keystream (ks)
  - The function (f) takes a encryption key as an input to generate keystream
  - Without the key we can't predict keystream
  - So combine each byte with one byte of plain text which is our ciphertext
  - C[i] denotes the ith unit of ciphertext
  - where a unit refers a 1 byte of data.
  - ks[i] refers the ith unit of keystream
  - M[i] refers to the ith unit of the message.
  - C[i] = ks[i] xor M[i]
  - m[i] = ks[i] xor C[i]
  - Disadvantages:

    - Assume we have used the same key to encrypt the Messages M and M'
    - If the attacker learns M and can compute ks by simply computing M xor C.
    - Once the attacker knows KS he can generate M'
  - RC4 was designed by Ron Rivest . RC4 is a variable key length cipher , with key can be anywhere between 8 to 2048 bytes long
  - SSL/TLS protocol use RC4 with 128-bit (16 bytes) key length
  - RC4 is extremely fast.

- **Block Ciphers**
  - **DES**
    - The data to be encrypted is processed in blocks of bytes (8 or 16).
    - Each possible plain text block corresponds to a row in the table
    - So to encrypt a block you find a column corresponding to the key, run down the table to find the row corresponding to the block you want to encrypt.
    - If the data is huge, it's very difficult to manage, so we define a function that does the computation
    - The idea is to simulate a random table
    - we have a key(k), and a data block . and we define 2 function E for encryption ,and D for decryption
    - C=E(K,M)
    - M=D(k,c)
    - When we have large messages , we do in Electronic code book mode.
      - Break the message up to block sized-chunks.
      - individually encrypt it using encryption algorithm
      - C[i]=E(k,M[i])
      - M[i]=D(K,C[i])
    - DES was desinged by IBM , it's patented but freely available
    - DES is a 64 block cipher with 56-bit key . The data is encrypted in blocks of 8 bytes and a key space of 56 bits.
    - DES is used with public-key techniques.
    - Disadvantages:
      - if M[i] and M[j] are same, then we get the same C[j] and C[k]. Attacker can get the pattern
      - Cipher block chaining mode, The encryption of each plain text block M[i] depends on cipher text of previous block C[i-1].
      - That can be accomplished by XORing previous Cipher text block C[i-1]] Xor M[i] before encryption
  - **3DES**

    - Run the DES algorithm 3 times.
    - It's used in Encrypt-Decrypt-Encrypt (EDE mode).
    - Message is encrypted with key-1, Decrypted with Key-2 and Encrypted with key-3.
    - 3DES is 3 times slower than DES.
  - **RC2**
    - RC2 is a block cipher invented by Ron Rivest,
    - RC2 is a variable-length cipher , with variable length key. It uses 64 bit block size.
  - **AES**
    - Advanced Encryption Standard uses a minimum of 128 bits and 3 key lengths, 128. 192 and 256 bits

- **Digest Algorithms**
  - The two most popular Digest Algorithms are MD5, which was designed by Ron Rivest and SHA-1 by NIST.
  - MD5 and SHA share a common ancestor MD4 also designed by Ron Rivest

## 1.4 Protocols

### 1.4.1 SSL

- Secure Socket Layer (SSL) is a protocol that provides a secure channel between machines
- It has facilities for protecting data in transit and identifying machine with which it is begin communicated
- Protocol is transparent , so it can be run on any protocol.
- SSL has gone through many versions and currently is culminating with the adoption by IETF as Transport Layer Security
- SSL was originally designed for world wide web , but also was intended as a unifying solution to all the other communications (web, mail, news traffic)
- The current version of SSL is tlsv2 which fixes a lot of security problems
- **Overview of SSL Protocol**
  - Primary goal of SSL is to provide privacy and reliability between 2 communicating applications
  - The protocol is composed of 2 layers : **Handshake protocol** and **record protocol**
  - In brief Handshake protocol provides
    - allows server & client to authenticate to each other
    - Negotiate encryption algorithm or Cryptography keys.

- Record protocol provides:
  - Confidentiality
  - Authenticity
  - replay protection

- **Handshake Protocol**
  - Client sends a list of algorithms it's willing to support along with a random number used as input to a key generation proces
  - Server chooses out of that list and sends it back along with a certificate containing servers public-key.
  - certificate also provides the server's identity for authentication purpose and the server supplies a random number which is used as a part of key generation process
  - Client verifies the servers certificate and extracts the server's public-key from the certificate
  - Client then generates a random secret called *pre_master_secret* and encrypts this with server's public-key
  - Client sends this encrypted *pre_master_secret* to the server
  - Client and the Server independently compute the encryption and MAC keys from *the pre_master_secret* and client and server's random values
  - Client sends a MAC of all the handshake messages to the server
  - The server sends a MAC of all the handshake messages to the client

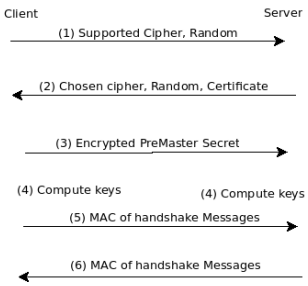



Figure 4: SSL Handshake

  -
  - Step:1 corresponds to single SSL Handshake message, Client Hello
  - Step:2 corresponds to SSL Handshake messages
    - First is a ServerHello  Algorithm preference, Certificate
    - Send ServerHelloDone
  - Step3 correponds to ClientKeyExchange
  - Step 5 and 6 correspond to the finished message. The Finished message is the first message that's the protected using Just-Negotiated algorithms.


Figure 5: SSL Handshake

- To protect the handshake from tampering, the content of the message is MAC of all the previous handshake messages
-
- **Resuming Session**
  - client sends a clientHello with a Session-ID of the session to be resumed.
  - server check it's session cache for a match
  - If the match is found & the server is willing to re-establish the connection under specified session state , it will send a Server-Hello with same session-ID
  - At this point both client and server must send change Cipher spec messages & proceed directly to finished messages
  - There is no exchange of certificates here.
  - Once the re-establishment is complete the client & server may begin exchange of application data
  - If the Session-ID match is not found the server generates a new session-ID & the client and Server have to perform a full handshake

- **SSL Record Protcol**
  - The actual data transfer is accomplished by SSL Record Protocol
  - Record Protocol works by breaking up the data stream to be transmitted in to a series of fragments, each of which is independently protected and transmitted.
  - On the receiving end , each record is independently decrypted and verified.
  - Before transmission , a MAC is computed on each record , This MAC is transferred along with the record
  - The concatenated data and MAC are encrypted to form encrypted Payload. We attach a header to that encrypted payload which we refer to as a record
  -
  - The record header provides the information for the other end to interpret the record. It contains:

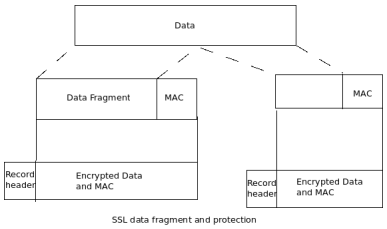



Figure 6: ssl data fragment and protection

  - ∗ Content Type (Application data, alert messages, handshake message, change cipher spec)
  - ∗ length (how many bytes to read off wire)
  - ∗ SSL Version
- The *change_cipher_spec* message indicates a change in encryption and authentication of records
- Once the handshake is completed a new set of keys is negotiated, *change_cipher_spec* record is sent to indicate that those keys will now be used.

# 2 Introduction to Public Key Infrastructure

Below is a simplified architectural model of Public Key Infrastructure using X.509 (PKIX) Specifications

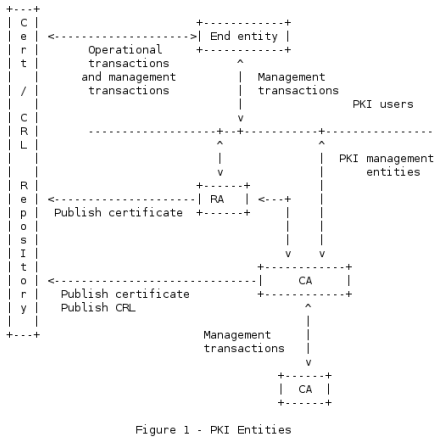



Figure 7: PKI Architectural Model

## 2.1 Common Terms used in PKI

- End Entiy: User of the PKI certificates and/or end user system that is the subject of a certificate
- CA: Certificate Authority

- RA: Registration Authority, Optional System to which CA delegates certain management functions
- CRL issuers: A system that generates CRL
- repository: a system or collection of distributed systems that stores certificates and CRLs and services as a means of distributing these certificates and CRLs to end entities

## 2.2 Detailed look on certificate/CRL

### 2.2.1 Certificates

- Certificates are data structures that bind public key values to the subject. This binding is asserted by a trusted Certificate Authority.
- X.509 defines the standard certificate format and v3 is the latest version.
- Internet Privacy Enhanced Mail(PEM) RFC 1421 and 1422 also include specifications for PKI based on X.509 certs.
- Types of Certificates:
  - End-Entity
  - CA
- RFC 1422 defines hierarchial structure of CA's and there are three types of PEM CA
  - IPRA: Internet Policy Registration Authority , acts as Root of Certificate authority. IPRA operates under Internet Society Organization
  - PCA: Policy Certificae Authority ,(Verisign, Digicert, etc) signed by IPRA
  - CA: Certificate Authorities signed by PCA (Organizational CA's)
- Policies used by CA:
  - IPRA certifies only PCA and not CA's or users cert
  - IPRA will make sure that the DN of the PCA is unique and will not certificate PCA's with similar DN
  - Certificates should not be issued to distinct entities under the same distinguished Name.
  - IPRA should not certifiy two PCA's with same DN
  - PCA's should not certify two CA's with same DN
  - CA's are expected to sign certificates only if the subject DN in the certificates is subordinate to the issuer CA DN.
- Types of Certificate Authorities
  - Cross-Certs: Where Issuer and Subject are different
  - Self-Issue: Where Issuer and Subject are same
    - Self-Signed: Where key bound in to the certificate is same as the key used to sgin the certificate

### 2.2.2 Basic Certificate Fields

- Version:
  - Describes the version of encoded certificate.
  - if extensions are used: **0x2(3)**
  - if extensions are not used but UniqueIdentifier is used: **0x1(2)**
  - if only basic fields are there: **0x0(1)**
  - Values: **0x2(3), 0x1(2), 0x0(1)**
- Serial Number:
  - Positive integer assigned by CA to each certificate
  - It must be unique for each Certificate given by CA
  - Can contain long integers (up to 20 Octets)
  - Values: Integers:
    - **16694152257348400000**
    - **0xe7ad8b07558a1727**
    - **cd:ba:7f:56:f0:df:e4:bc:54:fe:22:ac:b3:72:aa:55**
- Signature:
  - Algorithm used by **CA** to sign the certificate
  - Value:
    - **md2WithRSAEncryption**
    - **sha1WithRSAEncryption**
- Issuer:
  - Identifies the entity that has signed and issued the certificate
  - **MUST** contain non-empty Distinguished Names
  - Names should confirm to X.501 standard
  - Generally contains Country, Organization, Common name, Serial-Number, province, State, title, Surname, Generation Qualifier(Jr, Sr).
  - Values:
    - **C=US, O=VeriSign, Inc., OU=Class 1 Public Primary Certification Authority**
    - **C=DE, ST=Bayern, L=Muenchen, O=Whatever it is, CN=IO::Socket::SSL Demo CA**
    - **C=US, O=VeriSign, Inc., OU=Class 1 Public Primary Certification Authority**
- Validity:
  - The time interval during which the CA warrants that it will maintain status of the certificate

  - Consists sequence of 2 dates
    * Date on which certificate validity begins
    * Date on which certificate validity ends
  - Validity period of a certificate is period from notBefore to notAfter(inclusive)
  - Values:
    * **Not Before: Jan 29 00:00:00 1996 GMT**
    * **Not After : Aug 1 23:59:59 2028 GMT**
    * Special Note:
      · Devices are given certificates where there is no expiration date
      · Value:
      · Certificate is to be used for entire lifetime of the device **Not After: 99991231235959Z.(represented in Generalized Time)**

- Subject:
  - Identifies the entry associated with public key stored in the subject public key field
  - If the subject is CA then it should be populated with same data as issuer
  - Names should confirm to X.501
  - Values:
    * **C=US, ST=North Carolina, O=Fedora Project, OU=Fedora User Cert, CN=mrniranjan/emailAddress=niranjan@ashoo.in**

- Subject Public key Info:
  - This field is used to carry public key and identify the algorithm with which the key is used (RSA, DSA)
  - Supported Cryptographic Algorithms:
    * **Rivest-Shamir-Adelman (RSA)**
    * **Digital Signature Algorithm (DSA)**
    * **Diffie-Hellman (DH)**
    * **Elliptic Curve Digital Signature Algorithm (ECDSA)**
    * **Key Encryption Algorithm (KEA)**
    * **Elliptic Curve Diffie-Hellman (ECDH)**

#### 2.2.3 Extensions

- This field only appears in v3 Certs
- This contains sequence of one or more Certificate Extensions
- Extensions provides method for associating additional attributes with public keys
- Managing relationship between CA's

- Can carry private extensions to carry information unique to their community
- Each Extension is either Critical / Non Critical
- Each Extension includes OID and ASN.1 DER (OCTET)
  - Example: DE:65:01:16:19:2E:51:E0:9A:51:1A:37:50:94:7D:39:29:2A:42:2C
- There cannot be duplicates of Extensions
- Default CRITICAL value is false
- If the Certificate is CA , then they should have below Extensions
  - Basic Key Identifier
  - Authoritative key Identifier
  - Basic Constraints
- **Authority Key Identifier:**
  - Provides a means of identifying the public key corresponding to the private key used to sign the certificate
  - This is required if the issuer has multiple signing keys
  - If the CA certificate is self signed Authority key identifier is skipped
  - Authority key Identifier helps in identifying the issuer certificate.
  - Values:
    - **keyid:48:E6:68:F9:2B:D2:B2:95:D7:47:D8:23:20:10:4F:33:98:90:9F:D4**
- **Subject Key Identifier:**
  - Provides a means of identifying certificates that contain a particular public key
  - This extension is must for CA certificates
  - The value placed in this is same as Authority Key Identifier
  - For End-entity Certificates, this extension provides a means for identifying certificates containing particular key used in application
  - Value:
    - **48:E6:68:F9:2B:D2:B2:95:D7:47:D8:23:20:10:4F:33:98:90:9F:D4**
- **Key Usage:**
  - Defines the purpose of the key contained in the certificate
  - This extension is used to restrict the usage of the key to be used for purpose other than what is defined in Key Usage
    - **digitalSignature**
      - Public key should be used only to verify signatures on objects other than Public key certificates/CRL
    - **nonRepudiation/contentCommitment**

- subject Public key is used to verify digital signatures other than signatures on public key(CRL)
- used to provide nonRepudiation Service that protects against signing entity falsely denying any public action

* **keyEncipherment**
  - Public key is used for enciphering private key or secret keys ,
  - Is used for encrypting symmetric content-decryption key or an assymetric private key
* **dataEncipherment**
  - is used for enciphering the raw data without the use any cipher (very rarely used)
* **keyAgreement**
  - is used for key Agreement,
  - when used Deffie-Hellman key is to be used for key management
* **keyCertSign**
  - Public key is used for verifying the signatures on public keys , if this is true then
* **cRLSign**
  - when the subject public key is used for verifying signatures of CRL
* **encipherOnly**
  - subject's public key may be used only for enciphering data while performing key agreement
* **decipherOnly**
  - subject's public key may be used only for deciphering data while performing key agreement
* Important Note:

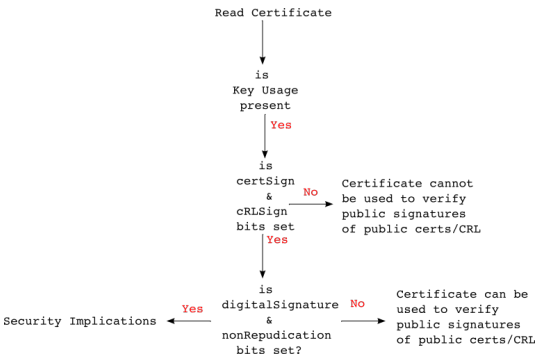



Figure 8: Key Usage restrictions

.

- **Certificate Policies**

- Contains sequence of one or more policy information terms each of which consits of OID, optional Queries
- OID Should not appear more than once
- for End-entity policy information terms indicate the policy under which the certificate has been issued and purposes for which the certificate must be used
- For CA the policy limits the policies for certificate paths that include this certificate
- if CA does not want to limit the set of policies for certification paths then it may assert **anyPolicy** with value **2.5.29.32.0**
- Policy Qualifiers:
  - CPSNotice: Points URL to *certificate practice statement* document that describes the policy under which the subject was issued
  - userNotice: text that describes the policy(not more than 200 characters)

- **Policy Mappings**
  - This extension is used by CA certificates
  - Lists one or more pairs of OID, which indicate that the corresponding policies of one CA are equivalent to policies of another CA [used in context of Cross pair certificates]
  - **OID: 2.5.29.33**

- **Subject Alternative Name**
  - Allows alternate names to be bound to subject of the certificate.
  - The names specified in subjectAltName may be included in addition to or in place of the identity in the subject field of the certificate
  - Defined subject names used:
    - rfc822Name: IA5String
    - otherName: OtherName
    - dNSName: IA5String
    - directoryName: Name
    - URI: IA5string
    - iPAddress: OCTET String
    - UniformResourceIdentifier: IA5String

- **Issuer Alternative Name**
  - Allows alternate names to be bound to certificate issuer
  - Issuer alternative names are not processed as part of certification path validation
  - Issuer Alternative name extension when present should be non-critical

- **Subject Directory Attributes**

- This extension is used to convey identification attributes of the subject
- Value:
  * nationality of the subject

- **Basic Constraints**
  - This extension identifies whether the subject of the certificate is CA ?
  - It also identifies Maximum depth of valid certification paths that include this certificate
  - Values:
    * cA: (boolean): indicates whether certificate is CA cert
    * pathLenConstraint: applicable if cA boolean is true. if prsent asserts keyCertSign bit.
    * It gives maximum number of non-self-issued intermediate Certificates that may follows this certificate in a valid certification path. The last cert is End-entity certificate
    * Ex: PathLength is 2:
      · RootCA - EE Cert
      · RootCA - subca1-EE Cert
      · RootCA - subca1-subca2-EE Cert
      · RootCA - subca1-subca2-subca3-EE Cert

- **Name Constraints**
  - Used only in CA certificate
  - Specifies name space within which all subject names in subsequent certificates ina certification path MUST be located.
  - Restrictions apply to Subject Dishtinguished Name and subject alternative names.
  - Restrictions do not apply for self-issued certs
  - Restrictions are defined in terms of permitted or excluded name subtrees
  - if name matches a restricted excluded subtrees, it 's invalid eventhough the name matches permittedsubtrees.
  - Examples:
    * URI: constraint applies to host part of the name
    * emailAddress: Example .example.org [indicates all emails with domain .example.com]
    * DNS : pki.example.org [www.host1.pki.example.org would satisfy but host1.example.org will not]
    * directoryName: Compares the DN attributes.

- **Policy Constraints**
  - policy constraints extension can be used in certificates issued to CAs.

  - Asserts policy related constraints
    * inhibitPolicyMapping: if present policy mapping is to be inhibited while processing subsequent certificates
    * requireExplicitPolicy: if present indicates subsequent certificates need to include an acceptable policy identifier

- **Extended Key Usage**:
  - This extension is used to indicate one or more purposes for which certificates public key can be used:
    * id-kp-serverAuth: TLS WWW server authentication
    * id-kp-clientAuth: TLS WWW client authentication
    * id-kp-codeSigning: Signing of downloadable executable code
    * id-kp-emailProtection: Email protection
    * id-kp-timeStamping: Binding the hash of an object to a time
    * id-kp-OCSPSigning: Signing OCSP responses

- **CRL Distribution Points**
  - cRLDistributionPoints extension is combination:
    * distributionPoint
    * reasons
    * cRLIssuer

- **Authority Information Access**
  - Indicates how to access information and services for the issuer of the certificate in which the extension appears
  - Information and services include, On-line validation services, CA policy data
  - This extension is added in both EE and CA cert

- **Subject Information Access**
  - This extension indicates how to access information and services for the subject of the certificate in which the extension appears
  - If the subject is CA, informaiton and services may include certificate validation and CA policy data.
  - If the subject is EE, information describes the type of services offerred and how to access them

#### 2.2.4 Revocation

When a certificate is issued, it is expected to be used for it's entire validity period. Due to various circumstances, certificate can be invalidated like:

- Change of name
- Change of association with subject and CA (Employee left)
- Compromise or private key

- CA wants to revoke the certificate
- X.509 defines one method of revoking certificates, where CA periodically issues a signed data structure called Certificate revocation list (CRL).
- CRL is a time-stamped list identifying revoked certificates that is signed by CA or CRL issuer.
- This list is freely available through public repositories
- It is expected that the certificate system user not only verifies certificate validity, signature but also aquires latest CRL from public repositories and check against CRL
- CRL's are issued periodically (hourly, daily or weekly).
- CRLissuers or CA issue CRL
- CAs publish CRLs to provide status information about the certificates they issued
- CA may delegate this responsibility to another trusted authority
- Details of CRL
  - Each CRL ahas particular scope
  - CRL scope is the ste of certificates that could appear in a given CRL
  - Examples:
    - all certificates issued by CA x
    - all CA certificates that has been revoked by key compromise or CA compromise
    - set of certificates base on arbitrary local information (all certificate issued to employees at location X)
  - CRL list all **unexpired** certificates, within it's scope that have been revoked for one or other reason
  - If the scope of the CRL includes one or more certificates issued by an entity other the CRL issuer it's called **indirect CRL**
  - CRL issuer may also generate delta CRL.
  - **Delta CRL** only lists those certificates whose revocation status has changed since the issuance of referenced complete CRL.
  - Referenced complete CRL is called **complete CRL**
  - Scope of the delta crl should be same as base CRL that it references
  - When CRL's are issued CRLs must be version 2 CRLs.

### 2.2.5 CRL Fields

- **Version** All CRLs must be Version 2 CRLs
- **Signature Algorithm**
  - Contains algorithm identifier for the algorithm used by the CRL issuer to sign the certificatesList
  - Values:
    - **SHA256withRSA - 1.2.840.113549.1.1.11**
- **Issuer Name**
  - Issuer Name identifies the entity that has signed and issued the CRL
  - Alternative name forms may also appear in the issuerAltName Extension
  - The issuer must contain a non-empty X.500 DN.
- **This Update**
  - This field indicates th issue date of this CRL.
  - Example:
    - **This Update: Sunday, January 17, 2016 8:06:03 AM IST Asia/Kolkata**
- **Next Update**
  - This field indicates the date by which the next CRL will be issued.
  - The next CRL may be issued before the indicated date, but it will not be issued later than indicated date.
  - CRL issuers(CA) **SHOULD** issue CRLs with nextUpdate time equal to or later thean all previous CRLs.
- **Revoked Certificates**
  - Contains list of certificates revoked by CA
  - Certificates revoked by CA are uniquely identified by their certificate serial Number.
  - Date on which revocation occurred is specified.

### 2.2.6 CRL Extensions

- **Authority Key Identifier**
  - This extension provides a means of identifying the public key correspondign to the private key used to sign the CRL. This identifier can be based on either the key identifier or issuer's name and serial number.
  - Example:
    - Authority Key Identifier Example:

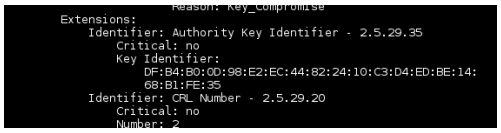



Figure 9: CRL Authority Key identifier

- **Issuer Alternative Name**
  - This extension allows additional identities to be associated with the issuer or CRL.
- **CRL Number**
  - This is a non critical extension that specifies sequence number for a given CRL scope and issuer.
  - This extension allows users to determine if a particular CRL supersedes another CRL.
  - if a CRL issuer generates delta CRLs in addition to complete CRLs for a given scope, the complete CRLs and delta CRLs must share one numbering sequence.
- **Delta CRL Indicator**
  - Delta CRL indicator is a critical CRL extension that identifies a CRL beging a delta CRL.
  - Delta CRLs contain updates to the revocation information previously distributed, rather than all the information that would appear in complete CRL.
  - This helps in reducing network load and processing time in certain environments
  - Example:

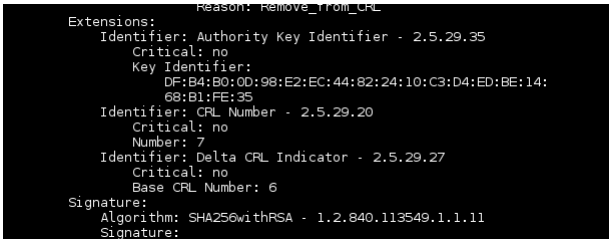



Figure 10: Delta CRL Indicator Extension

*

- **Issuing Distribution Point**
  - This is a critical CRL extension that identifies the CRL distribution point and scope for a particular CRL.

- It indicates whether CRL covers revocation of EE Certificates only, CA certificates etc.
- Example:

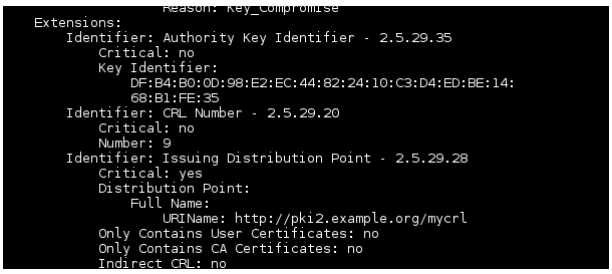



Figure 11: Issuing Distribution Point extension

*

## 2.3 Mozilla NSS

### 2.3.1 Introduction

- RHEL has 3 sets of security libraries:
  - OpenSSL
  - NSS
  - GnuTLS
- Network Security Services(NSS) is a project by Mozilla which provides crypto API majorly for Mozilla's products like *Firefox*, *Thunderbird*.
- Mozilla NSS provides all the Public Key cryptography stands which other implementations provide.

### 2.3.2 Architecture

- With default installation of RHEL, a NSS directory **/etc/pki/nssdb** is created for system wide usage.
- Unlike openssl where certificates are created and stored in text files, certificates created by nss-tools are stored in nss database
- All certificates and keys are stored in berkelely database(or SQLite). There are 3 major files generally seen in a nss directory are:
  - **key3.db** file contains a key used to encrypt and decrypt saved passwords
  - **cert8.db** contains certificates
  - **secmod.db** contains pathnames of pkcs#11 modules

## 2.4 Exercises

1. Setup: Create PKI Infrastructure using OpenSSL

```
$ mkdir -p
    ~/pki/ca/{certs,crl,newcerts,private}
cd ~/pki/ca
chmod 700 private
touch index.txt
touch index.txt.attr
echo 01 > serial
wget
    https://raw.githubusercontent.com/mrniranjan/pkiworkshop/master/openssl.cn
```

2. Create a CA cert using Openssl

```
$ cd ~/pki/ca
$ openssl genrsa -out private/ca.key.pem
    2048 -config openssl.cnf
$ chmod 400 private/ca.key.pem
$ openssl req -config openssl.cnf -key
    private/ca.key.pem -new -x509 \
   -days 365 -extensions v3_ca -out
       certs/ca.cert.pem
```

3. Create a End User cert with CN=Pki User1

```
# Create a private key

$ cd ~/pki/ca

$ openssl genrsa -out private/pkiuser1.key
    2048

# Create Certificate Request

$ openssl req -new -key
    private/pkiuser1.key \
   -out pkiuser1.csr -config openssl.cnf

$ openssl ca -config openssl.cnf
    -extensions usr_cert \
   -days 375 -in pkiuser1.csr -out
       certs/pkiuser1.cert.pem
```

4. Create a User cert *cn=Pki User1* with adding extension "Authority Key Identifier" where extension adds the hash of CA public key

   Add the below lines to **usr_cert** section of openssl.cnf

```
basicConstraints=CA:FALSE
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid
```

5. Create a user cert by adding followin extensions digitalSignature, nonRepudiation

   Add the below lines to **user_cert** section of openssl.cnf

```
keyUsage = nonRepudiation, digitalSignature
```

6. create a server cert by adding following extensions:

   Add the below lines to **server_cert** section of openssl.cnf

```
keyUsage = keyAgreement, digitalSignature
```

7. Create a user cert with subjectAltname having an email address

   Add the below lines to **user_cert** section of openssl.cnf

```
subjectAltName = email:copy
```

8. create a CA cert with pathlength of 2 . Verify path validation by creating multiple CAs

   Create a new section **v3_usr_ca** in openssl.cnf and create certs

```
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid
basicConstraints = critical,CA:true,pathlen:2
keyUsage = cRLSign, keyCertSign
```

9. Create a server cert with TLS server and client authentication

   Create a new section **server_cert** in openssl.cnf with below contents

```
[ server_cert ]
basicConstraints=CA:FALSE
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid
keyUsage = digitalSignature, keyAgreement
extendedKeyUsage = serverAuth,clientAuth
```

10. create a server cert with subject CN=server2.example.org having subjectAltName *.example.org

    Add below line to **server_cert** section and create a new section **alt_names**

```
 [ server_cert ]
 subjectAltName = @alt_names

[alt_names]
DNS.1 = *.example.org
```

11. Create a Name constraint where permitted DN attribute is "CN=pki*,OU=PKI,O=Example Org,C=US"

Add below lines to **v3_usr_ca**

```
[ v3_usr_ca ]
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid
basicConstraints = critical,CA:true,pathlen:0
keyUsage = cRLSign, keyCertSign
nameConstraints = permitted;dirName:dir_sect

[dir_sect]
C=IN
O=Example Org
OU=PKI
CN=*
```

12. Creating Certificate and key databases using NSS

- Create the below structure of directories to store CA, Server and user certificates
  - **/etc/pki/nssdb/CA_db** (for storing root CA certificates)
  - **/etc/pki/nssdb/server_db** (for storing certificates of servers)
  - **/etc/pki/nssdb/client_db** (for storing certificates of users)
- create a new certificate database CA_db

```
$ cd /etc/pki/nssdb
$ mkdir CA_db
$ certutil -N -d CA_db
```

- Create a Self signed Root CA certificate, specify the subject name

```
$ certutil -s -d CA_db -n
    "Example RootCA" -s
    "CN=Example
    RootCA,OU=IDMQE,O=Example
    Org,L=Pune,ST=Maharashtra,C=IN"
    -t "CT,," -x -2
```

Explanation of the flags:

```
-S specifies create a individual certificate
    and add it to the database
-d specifies the directory of the nss
    database ( -d . specifies current
    directory)
-n "Name of the certificate"
-t "CT,," specifies that this certificate is
    a Trusted CA which can issue server
    certficates
-x use the certutil tool to generate the
    signature for certificate being created
    or added to the database rather than
    obtaining from separate CA
```

```
-2 specifies that add basic constraint
    extensions, like the certificate can be
    used to sign specific operations like
    SSL, opbject signing, SMIME
```

- Extract the CA certificate from CA's cert database to a file

```
$ certutil -L -d CA_db -n
    "Example RootCA" -a -o
    rootca.crt
```

- Display the contents fot the CA's Certificate database

```
$ certutil -L -d .
```

- Creating Certificates for Servers, For servers SSL servers or clients

```
$ mkdir /etc/pki/nssdb/server_db
$ certutil -N -d server_db
```

- Import the new CA certificate in the Server certificate's database and we mark it as trusted CA

```
$ certutil -A -d server_db -n
    "Example CA" -t "TC,," -a
    -i CA_db/rootca.crt
```

Explanation of the flags:

```
-A Add an existing certificate to a
    certificate database
-d server_db (directory of certificate
    database)
-t "TC", specifies that this certificate is
    a Trusted CA which can issue server
    certficates, client certificates. ",,"
    specifies uu i.e "CT,u,u" , where u
    specifies certificate can be used
-a is ascii format
-i input file
```

- Create the server certificate request, specifying the subject name for the server certificate. Generally CN should be identical to the hostname of the server

```
$ certutil -R -d server_db -s
    "CN=pki1.example.org,OU=IDMQE,O=Example
    Org,L=Pune,ST=Maharashtra,C=IN"
    -a -o server_db/pki1.req -v
    12
```

- Sign the Certificate from CA , and issues a new certificate in ascii format

```
certutil -C -d CA\_db -c
    "Example RootCA" -a -i
    server_db/pki1.req -o
    server_db/pki1.pem -2 -6
```

- Import (Add) the new server certificate to the server's certificate database in the server_db directory with appropriate nickname

```
$ certutil -A -d server_db -n
    pki1 -a -i
    server_db\pki1.pem -t ",,"
```

- View the certificate

```
certutil -V -d server_db -u V -n
    pki1
```

- By default the certificates that have been created using the above procedure are in pkcs12 standard to use the certificates on Applications, You would require the following
  - RootCA's Certificate (public key)
  - Server Certificate (Certificate and key). The Server's participating in SSL communication should have the certificates and key file
- nss-tools comes with pk12util command , pk12util is a pkcs#12 Tool allowing to share certificates. The tool supports importing certificates and keys from pkcs#12 files in to NSS or export them and also list certificates and keys in such files
- Exporting the Server certifcate pki1.example.org in to file pki1.p12

```
$ pk12util -d server_db -o
    pki1.pk12 -n pki1
```

- The Above pki1.pk12 file has both the certificate and the key file encrypted
- For Applications which require certificate and key file to be in pem format we can use Openssl command to convert the file from p12 to pem format.

```
$ openssl pkcs12 -clcerts
    -nokeys -in pki1.pk12 -out
    pki1.pem
$ openssl pkcs12 -nocerts -in
    pki1.pk12 -out pki1.key.pem
    -nodes
```

# 3 Common Terms Used in RHCS

- Enrollment The point when the certificate is issued to the User/client is called enrollment
- Renewal When certificate is about to be expired or already expired and has CA requires to renew the certificate
- Dual Keys Conventional hierarchical PKI uses a single key pair one public key and one private key. According to VeriSign, Key Pairs are used for one or more of three basic purposes:
  - Encryption
  - Authentication
  - non-repudiation

  A single key used for multiple purposes violates non-repudiation. It is recommended that you assign two key pairs per person – one key pair for signing messages providing authentication and non-repudiation, and a second key pair for encryption. This allows someone to recover the encryption key and decrypt documents that were encrypted using it, without their gaining the ability to sign documents with that users private key as well (which could lead to forgery and violation of non-repudiation)
- ECC is a cryptographic system that uses elliptic curves to create keys for encrypting data. ECC creates cryptographically-stronger keys with shorter key lengths than RSA, which makes it faster and more efficient to implement. But it is not widely supported as RSA
- CRMF Certificate Request Message Format is a syntax use to convey a request for a certificate to Certificate Authority (CA). This request contains a public key, Proof of Possession(POP), Additional registration informtion combined with pop and public key.[7]
- PKCS10 is a standard which specifies syntax for Certificate Request. A certificate request consists of: [4]
  - Dishtinguished Name
  - Public Key
  - Optional Set of attributes
- FIPS Federal Information Processing Standards (FIPS) are standards to be followed by US Federal Govt for use in system by non-military govt. agencies and contractors.
- FIPS-140 standard specifies the security requirements for a cryptographic module utilized within a security system protecting sensitive information in computer and telecommunication systems. US national Institute of Standards and Technology(NIST) publishes FIPS series of standards for the implementation of Cryptographic modules. Cryptographic Module Validation Program (CMVP) validates cryptographic modules to Federal

Information Processing Standard (FIPS) 140-2 and other cryptography based standards.

FIPS 140-2 is primarily of interest to U.S., Canadian, and UK government agencies which have formal policies requiring use of FIPS 140 validated cryptographic software. [5]

Products that have received a NIST/CSE validation are listed on the Cryptographic Module Validation List : http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm

- HSM is a physical computing device that safeguards and manages digital keys for strong authentication and provides cryptoprocessing. HSM's are used in PKI environment. HSM's are designed to protect keys(private). A private key in HSM cannot be extracted.

# 4 Red Hat Certificate System

- RHCS is an Enterprise PKI implementation providing various PKI operations like Enrollment, renewal, revocation
- Provides Certificate Lifecycle Management like CA, Key archival, smartcard management.
- Key Features:
  - Certificate issuance, revocation, and retrieval
  - Certificate Revocation List (CRL) generation and publishing
  - Certificate profiles
  - Encryption key archival and recovery
  - Smartcard lifecycle management

## 4.1 Overview

RHCS Comprises of 5 subsystems each providing a set of PKI operations:

- Certificate Manager:Also called Certificate Authority is the core of PKI.
  - Issues certs
  - Revokes Certs
  - Publishes CRL
  - Establishes Security Domain of trusted subsystems
- Key Recovery Authority(KRA):It's also called Data Recovery Manager(DRM)
  - where private key is archived for certificates which have been created based on unique pair.
  - KRA archives the key pair and can later be retrieved if the private key of the certificate is lost
- Online Certificate Status Protcol(OCSP):

- OCSP Verifies certificate is valid or not expired

- Token Processing System(TPS)
  - TPS interacts with smartcard and manages keys and certificates on those tokens. TPS to interact with Smartcard it takes help of Token Key service which creates keys to interact with smartcard securely

- Token key System(TKS)
  - TKS derives keys based on the token CCID, private information,and a defined algorithm. These derived keys are used by the TPS to format tokens and enroll, or process, certificates on the token.

- Enterprise Security Client(ESC)
  - ESC is software installed on client systems which have tokens connected to it.
  - ESC is used to enroll, format smartcard
  - TPS talks to ESC to insert certificates in to smartcards.

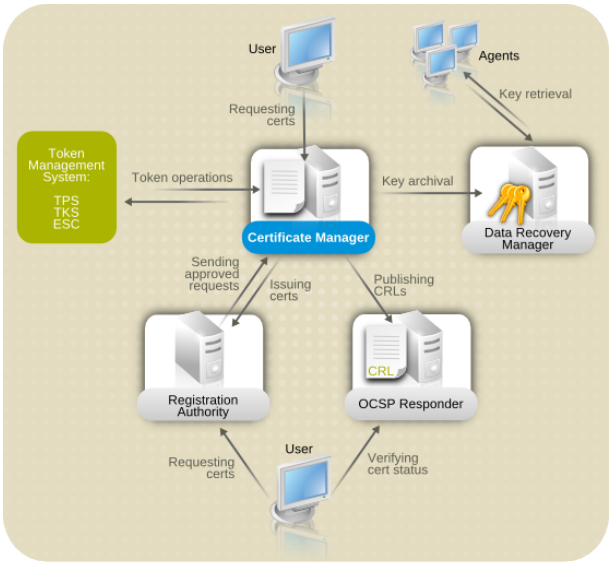



Figure 12: RHCS Subsystems [1]

## 4.2 Version

### 4.2.1 Older Versions

- Red Hat Certificate System 7.2 (2006)
- Red Hat Certificate System 7.3 (2007)

### 4.2.2 Current Version

- Red Hat Certificate System 8 (2009)
  - Support Status: Maitainance Support July 22, 2014 July 03, 2017
- Red Hat Certificate System 9 (2015)
  - Full support (including hardware updates): September 02, 2015 September 02, 2018

## 4.3 Certificate Manager

### 4.3.1 Introduction

Certificate Manager is the first subsystem that needs to be configured in PKI Environment, Certificate Manager can be configured as RootCA, Subordinate CA

### 4.3.2 Installation

- RPM: **pki-ca**
- configuration: CA subsystem is configured using utility **pkispawn**.

  pkispawn provides both interactive configuration or silent configuration by reading a configuration file.

  pkispawn first reads **default.cfg** first and gets other deployment specific information through interactive method or through batch mode by reading a file.

  pkispawn then passes this information to a java servlet which performs the configuration.

```
[root@pki2 ~]# pkispawn

IMPORTANT:

    Interactive installation currently only exists for very basic deployments!

    For example, deployments intent upon using advanced features such as:

        * Cloning,
        * Elliptic Curve Cryptography (ECC),
        * External CA,
        * Hardware Security Module (HSM),
        * Subordinate CA,
        * etc.,

    must provide the necessary override parameters in a separate
    configuration file.

    Run 'man pkispawn' for details.

Subsystem (CA/KRA/OCSP/TKS/TPS) [CA]:

Tomcat:
  Instance [pki-tomcat]:
  HTTP port [8080]:
  Secure HTTP port [8443]:
  AJP port [8009]:
  Management port [8005]:

Administrator:
  Username [caadmin]:
  Password:
  Verify password:
  Import certificate (Yes/No) [N]?
  Export certificate to [/root/.dogtag/pki-tomcat/ca_admin.cert]:
Directory Server:
  Hostname [pki2.example.org]:
  Use a secure LDAPS connection (Yes/No/Quit) [N]?
  LDAP Port [389]:
  Bind DN [cn=Directory Manager]:
  Password:
  Base DN [o=pki-tomcat-CA]:

Security Domain:
  Name [example.org Security Domain]:

Begin installation (Yes/No/Quit)? Yes
```

Figure 13: Configuring CA subsystem using pkispawn

### 4.3.3 Key Features

- CA subsystem issues, renews, revokes Certificates, generates Certificat Revocation lists
- Publishes Certificates/CRL in form of files or can publish to LDAP or OCSP responder
- CA also has an inbuilt OCSP responder enabling OCSP-Compliant clients to query CA about revocation status of Certificate
- Some CA's can delegate some of it's responsibility to another Subordinate CA

### 4.3.4 Architecture

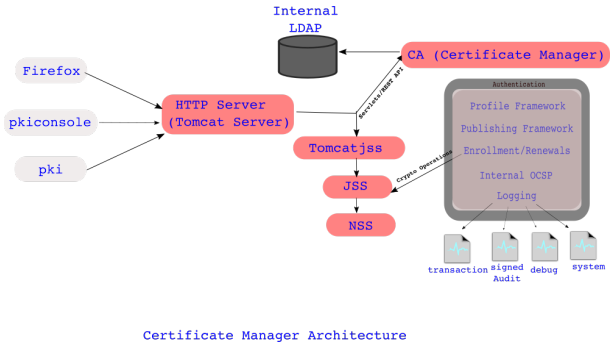



Figure 14: CA Subsystem Architecture

### 4.3.5 Interfaces

- End User Interface(Browser/CLI)
- Agent Interface(Browser/CLI)
- Admin interface(java console)

### 4.3.6 Features

- Enrollment:

  End user Enrolls in the PKI infrastructure by submitting a Enrollment(certificate) request through End Entity Interface. This request can be submitted through 2 Methods:

  - Browser
  - CLI

  There can be different kinds of Enrollment(Certificate) Request:

  - Request for User, Server, SMIME, Dual Cert,.. certificate
  - Request certificate if authentication through ldap, pin, cert etc.

Based on the above types, there are different certificate profiles associated with it. When end-entity(user) enrolls a certificate following events occur:

- The End-entity provides the information in one of the enrollment forms and submits a request
- The enrollment forms triggers the creation of public-key and private-key or dual-key pairs
- The End-entity provides authentication credentials before submitting the request, depending on the authentication type. This can be LDAP authentication, PIN-based authentication or certificate-based authentication.
- The request is submitted either to an agent-approved enrollment process or an automated process.
  - Agent-approved process requires no end-entity authentication, sends the request to the request queue in the agent-services interface.
  - Automatic notification can be setup so an email can be sent to an agent any time a request appears in the queue
  - The automated process, which involves end-entity authentication,process the certificate as soon as the end-entity successfully authenticates
- This form collects information about the end entity from the LDAP directory when the form is submitted
- The profile associated with form determine the aspects of certificate that is issued. Depending upon the certificate profile the request is evaluated to determine if the request meets the constraints set.
- The Certificate request is either rejected because it did not meet the certificate profile or authentication requirement, or a certificate is issued
- The certificate is delivered to end-entity through HTML interface or email or certificate can be retrieved through Agents interface by serial number or request-ID.
- The new certificate is stored in Certificate Managers internal database.

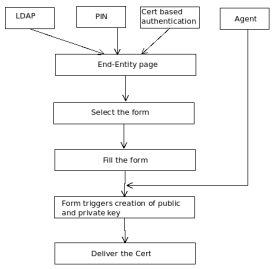



Figure 15: Certificate Enrollment

- Profiles:

  Profile determine the content of a certificate. Certificate manager provides customizable framework to apply policies for incoming certificate requests and to control the input requests types and output certificate types

  Certificate Profile define the following:

  - Authentication Method
  - Authorization Method
  - Certificate content
  - Constraints for the values of content
  - Contents of input
  - Output

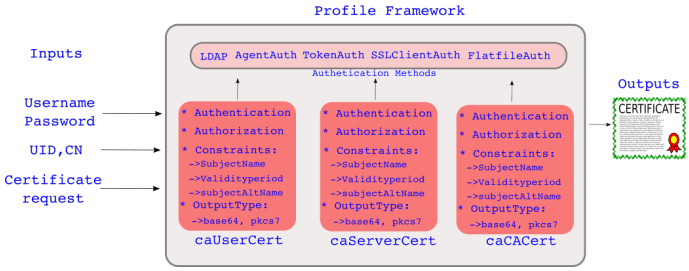



Figure 16: Certificate Profile Architecture

Profile Workflow:

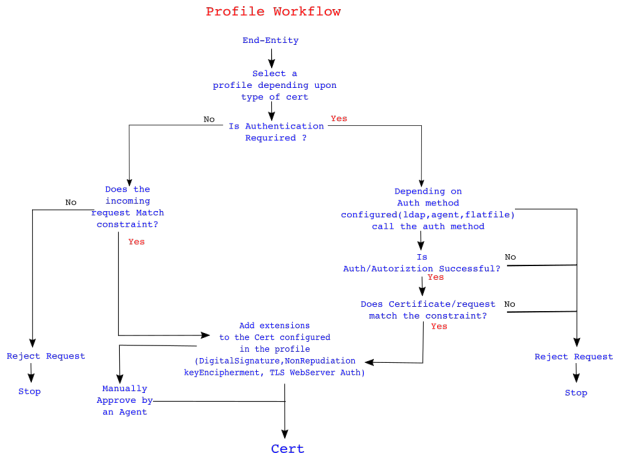



Figure 17: Certificate Profiles Workflow

Each profile are defined in **.cfg** file located at **/var/lib/instance_name/profiles/ca** directory

Example **caUserCert** Profile:


Figure 18: Profile Description

The first part of the profile is the description and specifies whether profile is enabled or disabled, if enabled who enabled it.

The second part of the profile describes the inputs:

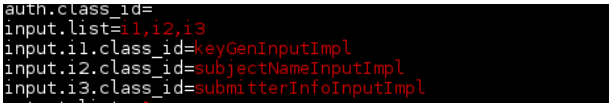



Figure 19: Profile inputs

- KeyGenInputImpl: specifies the key pair generation during the request submission, This provides if the request should be of type CRMF/PKCS10, also provides dropdown specifying the key size.
- subjectNameInputImpl: specifies the subject Distinguished Name(DN) to be used in the cert. The subject DN can be constructed from *UID, Email, Common Name, Organizational Unit, Country*
- submitterInfoImpl: This input specifies three fields: *Requester Name, Requester email, Requester phone*

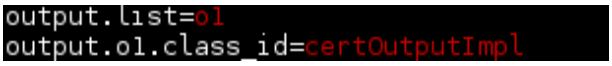



Figure 20: Profile output

Third part of the profile is output,

- certOutputImpl: The certificate output format *base64, pkcs7, prettyprint*

Last part of the profile is constraints, Policies like:

- validity of the cert
- renewal settings,
- key Usage Extensions
- User supplied extensions

- Publishing Certificate System provides customizable framework from CA's to publish.
  - Key Features
    - Publish to a single repositories or multiple repositories
    - Split locations by certificates/CRL

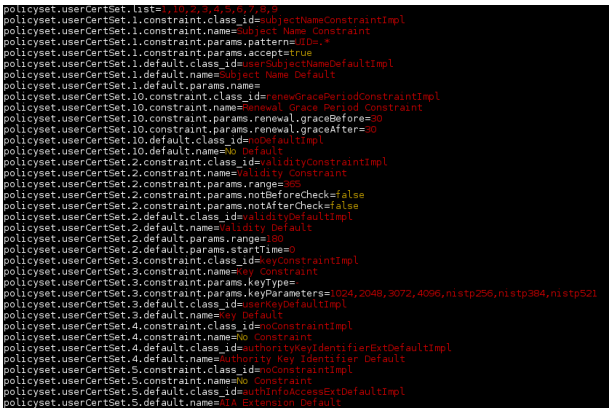



Figure 21: Profile Policies

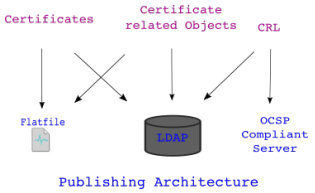



Figure 22: Publishing Architecture

  - Set individual rules for each type of certs/crl
- Publishing framework consists
  - Publishers
  - Mappers
  - Rules
- Publishers: Publishers specify location to which certificates/CRL's are to be published. Example:
  - To publish to a file, publishers specify the location of the publishing directory.
  - To publish to LDAP, publishers specify the attribute in the directory that srores the cert/CRL.
  - To publish to OCSP, we specify OCSP Server details.
- Rules: Rules define
  - what is to be published and where ?
  - What type of certs can be published to what location
  - Set rules to publish certs to file and LDAP
  - Set individual rules for each type of cert/rule
  - There are rules for Files, LDAP and OCSP
- Mappers: Mappers are only used when publishing to LDAP
  - Mappers construct the DN for an entry based on the information from the certificate or certificate request.

    * Mappers use certificate or certificate request's subject name to construct the DN of the entry to which cert/certificate request/CRL has to be published

#### 4.3.7 Instance Layout

#### 4.3.8 Exercises

### 4.4 Key Recovery Authority

#### 4.4.1 Introduction

Key Recovery Authority is an optional subsystem that is configured to archive private keys. Key Recovery Authority subsystem is by default installed after CA subsystem is installed and is dependent on CA for certain operations but can be installed as a standalone subsystem too.

#### 4.4.2 Installation

- RPM: pki-kra
- Configuration: KRA subsystem can be configured through pkispawn using interactive or to have a customize setup a separate file can be specified with pkispawn.
- KRA subsystem can share the same tomcat instance created by the CA subsystem. This allows KRA subsystem to run on the same ports as that of CA.


Figure 23: pkispawn KRA subsystem

### 4.4.3 Installation Layout

- **/var/lib/pki/<instance>**: <Instance> here is the tomcat Instance name given during configuring subsystem
- **/var/lib/pki/<instance>/alias**: Is the NSS Database of the system where the subsystem system certs are stored. When a subsystem is configured, Certain certificates are created for certain tasks like Transport Cert for securely transporting private keys, storage cert for storing the private key, These system certificates reside in alias directory. List of system certs are mentioned below, All the below system certs are issued by CA during configuration of KRA.
  - transportCert
  - Server-Cert
  - auditSigningCert
  - storageCert
  - subsystemCert
- **/var/lib/pki/<instance>/conf/**: Configuration directory containing tomcat configuration, password file containing internal database(ldap) password, and NSS Database password

- **/var/lib/pki/<instance>/kra/**: Contains KRA Subsystem configuration file CS.cfg. This file is the main configuration file for KRA subsystem
- **/var/lib/pki/<instance>/lib/**: Contains kra subsystem related jar files
- **/var/lib/pki/<instance>/bin/**: Tomcat related jar files

### 4.4.4 Key Features

- Key Recovery Authority when configured with with certificate Manager(CA) stores private keys as part of certificate issuance process
- Supports two key recovery modes:
  - Synchronous Recovery(Deprecated)
  - Asynchronous Recovery
- Supports Archiving passwords, symmetric keys, aka acts as a vault to store anything that needs to be stored securely and transported back securely
- Private keys are generally wrapped with a key called storage key and private keys while transporting are wrapped with transport keys, These keys can be rotated

### 4.4.5 Architecture

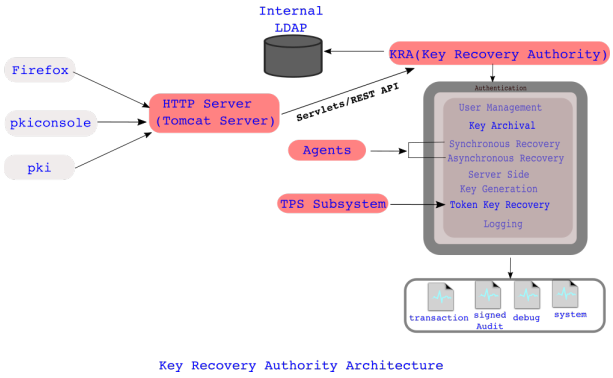



Figure 24: KRA Architecture

### 4.4.6 Interfaces

- End User Interface through **pki** cli
- Agent Interface(Browser/CLI)
- Admin Interface(Java console)

### 4.4.7 Features

- Key Archival
  - key archival is triggered when a user creates a certificate request through firefox(31.0 or older only)/pki/CRMFPopClient utility
  - The key pair generated as part of the request(public/Private key) is submitted through a defined profile.
  - While submitting the private key is wrapped(encrypted) with transport cert(public key) and submitted to CA
  - CA inturn retrieves the encrypted private key, decripts with transport private key
  - Private key is then stored in interan ldap of kRA by encrypting again with Storage Cert
  - CA upon successful archival issues certificate

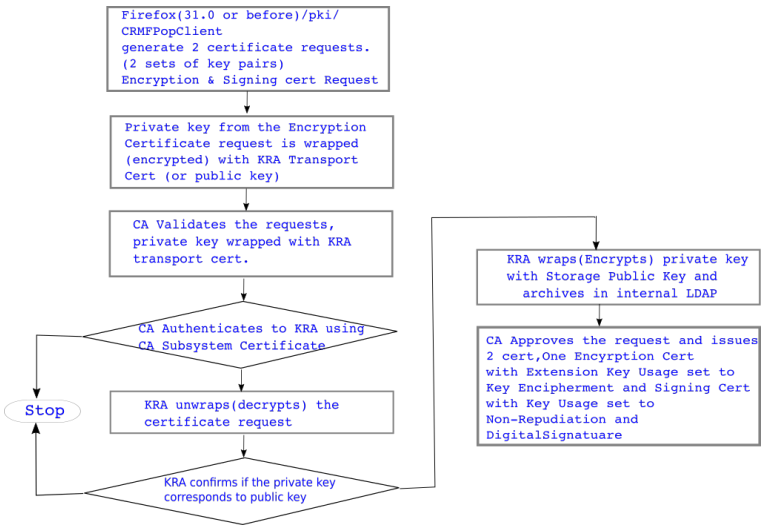



Figure 25: Key Archival Workflow

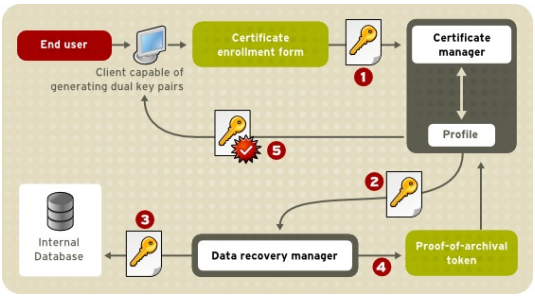



Figure 26: Key Archival Process

- Key Recovery

- KRA supports Agent initiated key recovery
- Designated Agents use the key recovery form on Agent services page to Approve/Reject key recovery
- CS uses m-of-n ACL-based recovery scheme
- CS uses it's ACL to ensure recovery agents are appropriately authenticated over SSL
- ACL also requires that agents belong to specific recovery agent group
- Recovery request is executed only when m of n agents have granted authorization

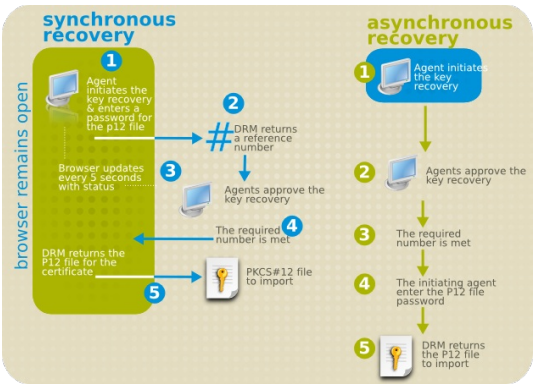



Figure 27: Key Recovery Process

- Archive Symmetric Keys, passwords, Asymetric keys (Vault)
  - KRA has new REST interface to archive and recover
    - Symmetric Keys
    - Random Security Data
    - Asymmetric Keys
  - This features is implemented through pki key CLI to generate, archive,retrieve keys.
- Transport Key Rotation
  - Transport Certs are created when KRA instance is configured using pkispawn
  - Transport Certs are issued by CA to KRA
  - Transport Certs public key is used to *encrypt* private key while it's being transport by CA to KRA
  - In Large organizations, this transport key needs to be rotated.
  - The feature here adds support for second Transport Key

### 4.4.8 Exercises

## 4.5 Online Certificate Status Protocol

### 4.5.1 Introduction

- Online Certificate Status Manager is an subsystem external to CA.
- This subsystem acts as an external OCSP which can be publicly accessible.
- Multiple CA's can publish CRL to a single OCSP
- Clients can verify certs using OCSP

### 4.5.2 Installation

- RPM: **pki-ocsp** OCSP subsystem can be configured through pkispawn using interactive or to have a customize setup a separate file can be specified with pkispawn

  OCSP subsystem can share the same tomcat instance created by the CA subsystem which allows OCSP to use same ports as that of CA.

```
[root@pki2 ~]# pkispawn

IMPORTANT:

    Interactive installation currently only exists for very basic deployments!

    For example, deployments intent upon using advanced features such as:

        * Cloning,
        * Elliptic Curve Cryptography (ECC),
        * External CA,
        * Hardware Security Module (HSM),
        * Subordinate CA,
        * etc.,

    must provide the necessary override parameters in a separate
    configuration file.

    Run 'man pkispawn' for details.

Subsystem (CA/KRA/OCSP/TKS/TPS) [CA]: OCSP

Tomcat:
  Instance [pki-tomcat]: FoobarOCSP
  HTTP port [8080]: 14080
  Secure HTTP port [8443]: 14443
  AJP port [8009]: 14009
  Management port [8005]: 14005

Administrator:
  Username [ocspadmin]:
  Password:
  Verify password:
  Import certificate (Yes/No) [Y]? N
  Export certificate to [/root/.dogtag/FoobarOCSP/ocsp_admin.cert]:
Directory Server:
  Hostname [pki2.example.org]:
  Use a secure LDAPS connection (Yes/No/Quit) [N]?
  LDAP Port [389]: 1389
  Bind DN [cn=Directory Manager]:
  Password:
  Base DN [o=FoobarOCSP-OCSP]:

Security Domain:
  Hostname [pki2.example.org]:
  Secure HTTP port [8443]:
  Name: Foobar Org
  Username [caadmin]:
  Password:

Begin installation (Yes/No/Quit)? Yes
```

Figure 28: pkispawn KRA subsystem

### 4.5.3 Interfaces

- End User Interface through **pki** cli

- Agent Interface(Browser/CLI)
- Admin Interface(Java console)

#### 4.5.4 Exercises

## 4.6 SmartCard Management

### 4.6.1 Introduction

- **What:** Smartcard contains an embedded microprocessor. This microprocess runs Java Virtual Machine(JVM) and provides environment to run applications on top of it.
  - Smartcard contains a chip with java on it.
  - it contains a JVM, Java Runtime Environment, and some API for applications to access the card.
- **Why:**
  - Smartcards are typically used by enterprise to provide identification and authentication
  - Smartcards provide 2-Factor Authentication which is
    - Something you have - smartcard
    - Something you know - pin(password) to access the smartcard
- **Use cases:**
  - System Authentication using Smartcard (Linux/Windows System Authentication)
  - Secure VPN Access
  - Email Signature and Encryption
  - Disc Encryption
- **How:**
  - For enterprise to use smartcard a digital Certificate is issued to the smartcard
  - This certificate is used to identify the user of the smartcard
  - User uses digital certificate on the smartcard to perform 2-factor authentication
- **Mangement:**
  - We need a management software to manage certificates on the smartcard
    - Enrolling certificate
    - Formatting Smartcard
    - What to do with the certificate if smartcard is lost
    - What to do if the user of the smartcard is no longer in the company

– To do the above operations, we have Token Management system in Certificate Services which comprises of the following:
  * **TPS**: The Token Processing System (TPS) interacts with smart cards to help them generate and store keys and certificates for a specific entity, such as a user or device.
  * **TKS**: The Token Key Service (TKS) generates, or derives, symmetric keys used for communication between the TPS and smart card.
  * **CA**: The Certificate Authority (CA) creates and revokes user certificates stored on the smart card.
  * **KRA**: Optionally, the Key Recovery Authority (KRA) archives and recovers keys for the smart card.
  * **Enterprise Security Client**: is the conduit through which TPS communicates with each token over a secure HTTP channel (HTTPS), and, through the TPS, with the Certificate System.

### 4.6.2 Architecture

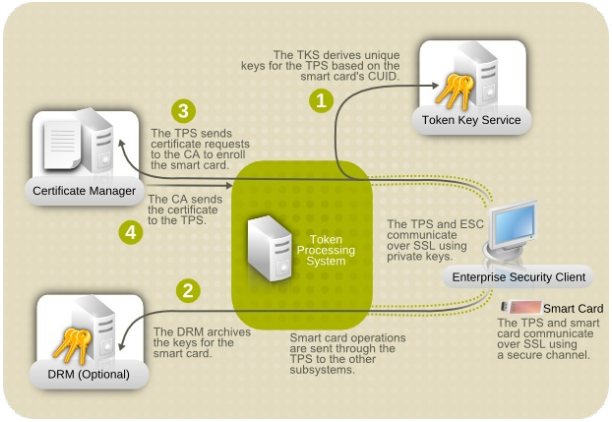



Figure 29: Token Management System

### 4.6.3 Common Terms Used with regard to Smartcard

- Secure Channel Protocol: A secure communication protocol and set of security protocol used to create a secure channel with token during TPS Operations.
- Applets: TPS communicates with an applet on the smartcard.
- APDU(Application Protocol Data Unit): Standard communication messaging protocol between a card accepting device(Reader) and a smart card
- SmartCards come with a cardmanager applet and a vendor applet or with only card manager applet.
- Coolkey: applet is the supported Card Manager applet

- Smartcard has cryptographic functionality supporting symmetric and Assymetric Encryption(like RSA)
- CCID: CCID (chip card interface device) protocol is a USB protocol that allows a smartcard to be connected to a computer via a card reader using a standard USB interface, without the need for each manufacturer of smartcards to provide its own reader or protocol. [8]

#### 4.6.4 Token Key Service

- Introduction
  - A subsystem in the token management system which derives specific, separate keys for every smart card based on the smart card APDUs and other shared information, like the token CUID. [2]
  - The TKS derives keys based on the token CCID, private information, and a defined algorithm. These derived keys are used by the TPS to format tokens and enroll, or process, certificates on the token(smartcard). [2]
  - TKS and TPS have a sharedSecret to wrap sensitive info sent between TKS and TPS
- Features:
  - **Manage shared keys**
    - Token Key Service (TKS) derives keys for the TPS to use.
    - Based on Token CUID, and other token material , Derive keys that encrypt session between TPS and ESC
    - Keys are derived based on Common Master Key which is known to TKS and existent on smartcards
  - **Generate Master Key**
    - Master key is 3DES symmetric Key stored in software/Hardware
    - Transport Key wraps the other keys derived(from master key) and used by TKS and TPS

#### 4.6.5 Token Processing Service

- Introduction
  - TPS interacts with smart cards to help them generate and store keys and certificates for a specific entity
  - Smart card operations go through TPS and are forwarded to the appropriate subsystem for action, such as the Certificate Authority to generate certificates or the Key Recovery Authority to archive and recover key
- Features
  - Secure Channel
    - The Token Key Service (TKS) generates the (token) keys (a set of 3 keys derived) from Master keys

    * These keys are used by TPS to communicate with ESC(smartcard)
    * TPS communicates with the TKS over SSL but TPS communicates with ESC over secure channel
    * Every smartcard has 3 keys:
        · An **auth key** which is used for encryption and authentication: a session key dervied from the auth key each time a secure channel is opened.
        · A **MAC key** which is used for message authentication; like the auth keys, a session key is derived from the MAC key each time a secure channel is opened.
        · A **key encryption key (KEK)** which is to encrypt the session keys.

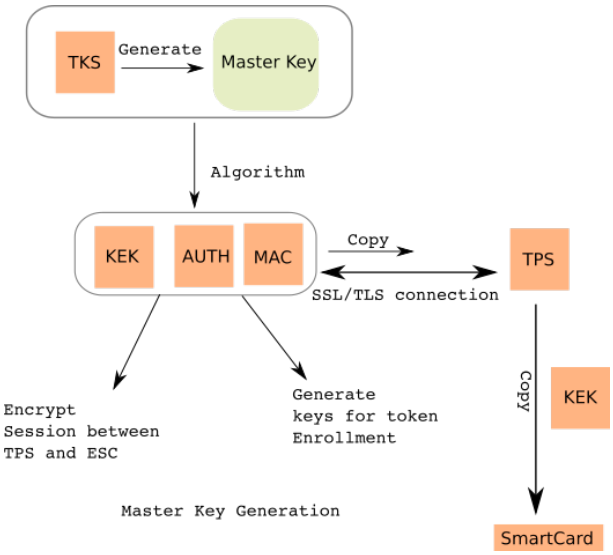



Figure 30: TKS Key Derivation

– Supported TPS Operations
    * Formatting Smartcards
    * Resetting PIN on smartcards
    * Upgrading the applet for smartcard tokens
    * Perform LDAP Authentication
    * Managing token database
    * Logging Token events
– Token Profiles
    * Just like certificate profiles there are different token profiles to format different kind of tokens
    * Token profiles define the below:
        · The steps to format and enroll the token
        · The configuration of the final enrolled token

- Profiles also specify which LDAP server to be used for authentication and which CA should be used for enrolling certificates
- List of Token profiles:

**Table 2.1. Default Token Types**

| Token Type | Description |
|---|---|
| cleanToken | For operations for any blank token, without any other applied token types. |
| soKey | For operations for generating keys for security officer stations. |
| soCleanSOToken | For operations for blank tokens for security officer stations. |
| soKeyTemporary | For operations for temporary security officer tokens. |
| soCleanUserToken | For operations for blank user tokens for security officers. |
| soUserKey | For operations for security officer user tokens. |
| tokenKey | For operations for generating keys for uses with servers or devices. |
| userKey | For operations for regular user tokens. |
| userKeyTemporary | For operations for temporary user tokens. |

Figure 31: List of Token Profiles

– TPS-TKS-Smartcard Workflow

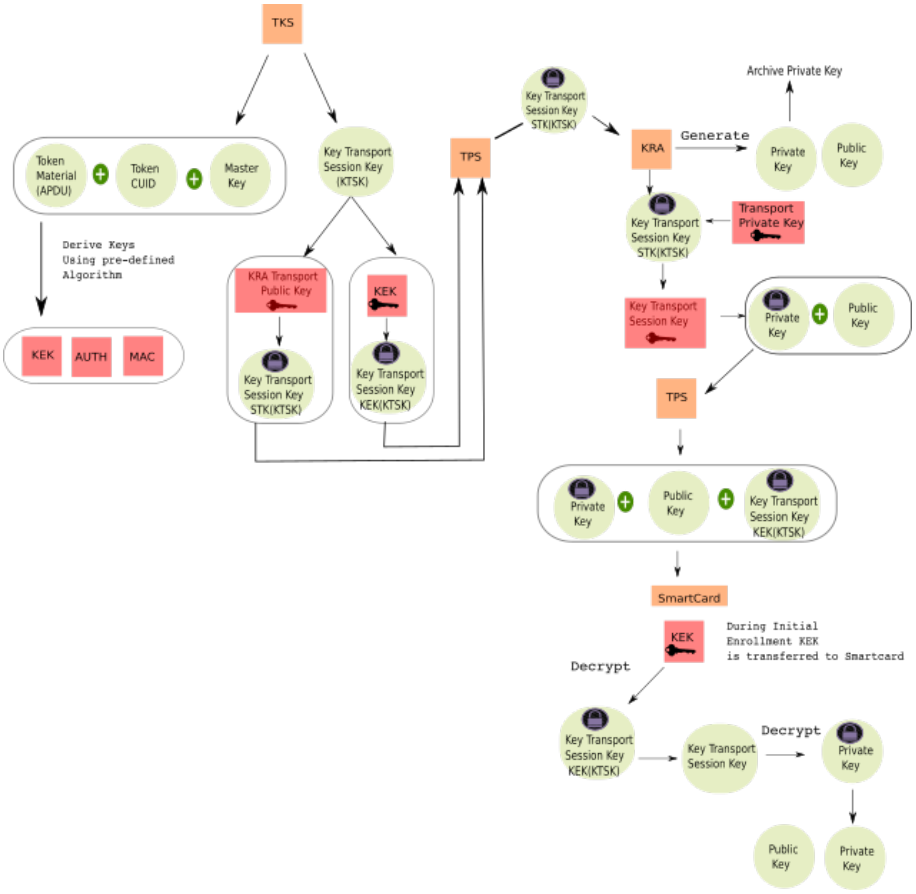



Figure 32: TPS-TKS-Smartcard Worfklow

– TKS Configuration
  * RPM: **pki-tks**
  * Configuration: TKS subsystem can be configured through pkispawn using interactive or to have a customize setup a separate file can be specified with pkispawn
  * TKS subsystem can share the same tomcat instance created by the CA subsystem which allows TKS to use same ports as that of CA

```
[root@pki2 ~]#
[root@pki2 ~]# pkispawn

IMPORTANT:

    Interactive installation currently only exists for very basic deployments!

    For example, deployments intent upon using advanced features such as:

        * Cloning,
        * Elliptic Curve Cryptography (ECC),
        * External CA,
        * Hardware Security Module (HSM),
        * Subordinate CA,
        * etc.,

    must provide the necessary override parameters in a separate
    configuration file.

    Run 'man pkispawn' for details.

Subsystem (CA/KRA/OCSP/TKS/TPS) [CA]: TKS

Tomcat:
  Instance [pki-tomcat]: FoobarTKS
  HTTP port [8080]: 16080
  Secure HTTP port [8443]: 16443
  AJP port [8009]: 16009
  Management port [8005]: 16005

Administrator:
  Username [tksadmin]:
  Password:
  Verify password:
  Import certificate (Yes/No) [Y]? N
  Export certificate to [/root/.dogtag/FoobarTKS/tks_admin.cert]:
Directory Server:
  Hostname [pki2.example.org]:
  Use a secure LDAPS connection (Yes/No/Quit) [N]?
  LDAP Port [389]: 1389
  Bind DN [cn=Directory Manager]:
  Password:
  Base DN [o=FoobarTKS-TKS]:

Security Domain:
  Hostname [pki2.example.org]:
  Secure HTTP port [8443]:
  Name: Foobar Org
  Username [caadmin]:
  Password:

Begin installation (Yes/No/Quit)? Yes

Log file: /var/log/pki/pki-tks-spawn.20160125182959.log
Installing TKS into /var/lib/pki/FoobarTKS.
Storing deployment configuration into /etc/sysconfig/pki/tomcat/FoobarTKS/tks/deployment.cfg.
```

Figure 33: Interactive installation of TKS using pkispawn

– Creating sharedSecret
  * TKS generates a **sharedSecret** to wrap sensitive information passed over TPS.
  * This sharedSecret is to be created on TKS NSS database using tool called **tkstool**

```
[root@pki2 conf]#
[root@pki2 conf]#
[root@pki2 conf]# cat password.conf
internal=26643854353
internaldb=Secret123
replicationdb=-1525134020
[root@pki2 conf]# cd alias/
[root@pki2 alias]# tkstool -T -d /var/lib/pki/FoobarTKS/alias/ -n sharedSecret
Enter Password or Pin for "NSS Certificate DB":
```

Figure 34: NSS Internal database password


Figure 35: tkstool Usage


Figure 36: First Session Key


Figure 37: Second Session Key

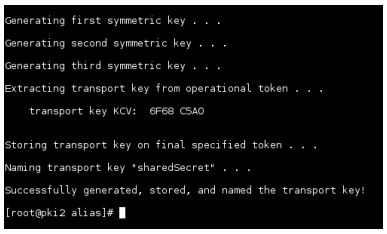



Figure 38: Third Session Key

    * tkstool utility prints out the key shares and KCV values for each of the three session keys that are generated. These are necessary to import the **sharedSecret** to TPS

– TPS Configuration
    * RPM: **pki-tps**
    * Configuration: TPS subsystem can be configured through pkispawn using interactive or to have a customize setup a separate file can be specified with pkispawn
    * TPS subsystem can share the same tomcat instance created by the CA subsystem which allows TPS to use same ports as that of CA


Figure 39: Interactive installation of TKS using pkispawn

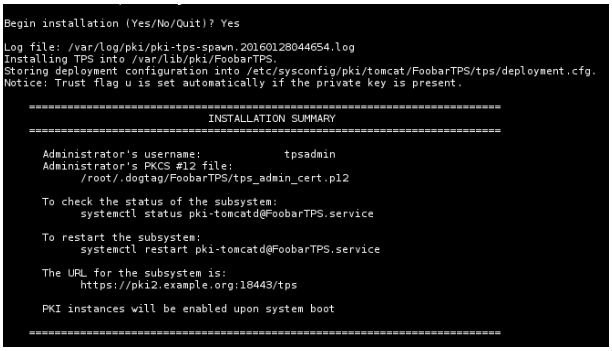



Figure 40: Continuation of pkispawn

* Import SharedSecret
  - Import the sharedSecret created by TKS so that TKS and TPS can wrap sensitive information
  - sharedSecret needs to be imported in to TPS NSS Database


Figure 41: Get NSS Password of TPS NSS database

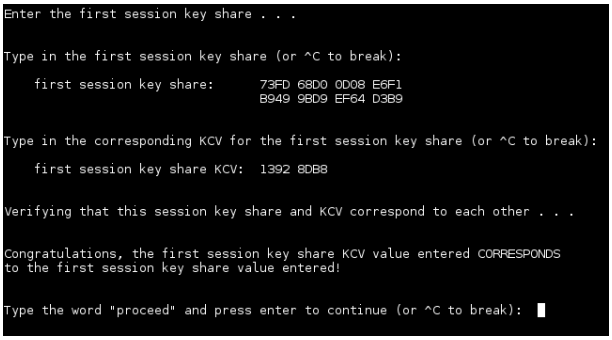



Figure 42: tkstool to import first session key

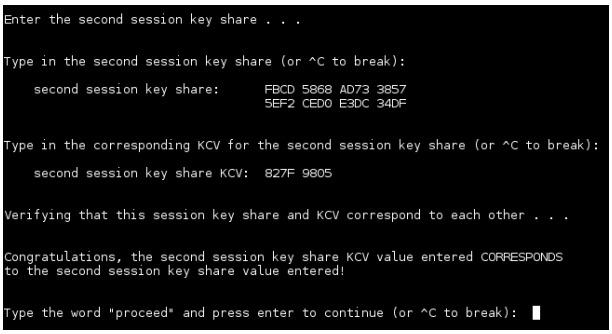



Figure 43: tkstool to import second session key


Figure 44: tkstool to import third session key

* TKS Instance Layout

.

# References


[1] Red Hat Certificate System Installation & Deployment Guide. Red hat certificate system overview, 2015.

[2] Red Hat. Red hat certificate services 8.0 admin guide, 2015.

[3] Red Hat. Symmetric encryption, 2015.

[4] B. Kaliski M. Nystrom. PKCS #10: Certification Request Syntax Specification. RFC 2986, RFC Editor, November 2000.

[5] NIST. Fips publication, 2015.

[6] Eric Rescorla. *SSL and TLS Designing and Building Secure System.* Addison-Wesely, 2001.

[7] J. Schaad. Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF). RFC 4211, RFC Editor, September 2005.

[8] Wikipedia. Ccid (protocol), 2015.