Public Key Infrastructure and the Tool Command Language

Roy S. Keene
Jemimah Ruhala

September 23rd, 2013

Abstract

Public key cryptography, also known as asymmetric cryptography, is a mechanism by which messages can be transformed (either encrypted or decrypted) with a private key and then reassembled with a related, but separate public key. The public key infrastructure (PKI) that supports the trust model by which two parties that have no previous knowledge of each other can verify each others identity through trust anchors is implemented using digital signatures made possible by public key cryptography. Being able to participate in PKI directly with Tcl scripts makes many interesting and useful applications possible. The PKI module in Tcllib implements the various PKI related standards and algorithms for Tcl scripts to accomplish this goal including PKCS#1, X.509, and RSA. This paper aims to explain the high-level concepts related to PKI, describe the scope of the various standards and algorithms implemented, and explore the possibilities of PKI in Tcl.

1 PKI

The purpose of Public Key Infrastructure (PKI) is to provide a mechanism to establish the authenticity of an unknown party. Most people assume PKI is a complex system with a lot of moving parts. However, this is not the case. A simple PKI implementation is fairly straight-forward.

PKI establishes authenticity using certificates, which certify that some external entity has verified that an unknown party is who it claims to be. This certificate is presented by the unknown party as a means of identifying itself to you.

1.1 Public Key Cryptography

Public key cryptography, or asymmetric cryptography uses *key pairs* to perform cryptographic operations on values. This is in contrast to symmetric cryptography which uses a single key to perform cryptographic operations. In public key cryptography there are two related, but distinct, keys:

- A public key
- A private key

The relationship between these two keys differs depending on the public key cryptography algorithm, but in all cases it is impossible[1] to derive the private key from the public key.

Public key cryptography uses mathematical operations to perform operations on numerical values that can only be reversed or verified with the opposite key. That is, if something is encrypted with the private key it can only be decrypted or verified with the public key. Conversely, if something is encrypted with the public key it can only be decrypted or verified with the private key.

Given that there are two different keys (public and private) involved with public key encryption, refering to encryption in a general sense can be ambiguous. Which key do we use to encrypt something with ? For reasons that should be made clear later, when we talk about encryption within the context of public key cryptography, we are usually talking about encrypting a plain text value with the public key. This results in a cipher text that can only be decoded by an entity possessing the private key.

In the context of public key cryptography, encrypting a value with the private key is generally used for digital signatures. Because messages encrypted with the private key can only be decrypted (or verified)

[1] Theoretically infeasable given sufficiently large and correctly generated keys

with the public key part of the key pair we can assert that if the message is meaningful when decrypted with the public key then it could have only been encrypted by an entity possessing the private key.

1.2 RSA

The most common public key cryptography algorithm in use today is RSA[2]. RSA is named for Ron Rivest, Adi Shamir, and Leonard Adleman who made the algorithm public and are thus widely credited as the inventors of the algorithm.

1.2.1 Description of RSA Key Pairs

The RSA algorithm defines the following pairs of keys in the following way:

- RSA Public Key, made up of
  - The public exponent (commonly called e), often just called the exponent
  - The public modulus (commonly called n), often just called the modulus
- RSA Private Key, made up of
  - The private exponent (commonly called d), often just called the private key since it's the only part of it

In the remainder of this document whenever RSA is used a simple 16-bit RSA key pair will be used:

- Public Key
  - Public Exponent: 65537
  - Public Modulus: 37837
- Private Key
  - Private Exponent: 40193

In practice RSA keys are much larger, as of this writing the recommended minimum size of RSA keys is 2048-bits.

[2]According to RSA

1.2.2 RSA Algorithm

The RSA algorithm is a relatively easy to demonstrate public key cryptography algorithm. A simple definition of applying RSA is "modular exponentiation". Modular arithmetic (denoted by the "$\mathbb{Z}$" symbol and a modulo value) is applied to values that have had the the exponentiation operator applied to them. A simple example of modular exponentiation is:

$$3^4(\mathbb{Z}5) = 1$$

which is 3 raised to the power of 4 all modulo 5. Since $3^4 = 81$ and $81(\mathbb{Z}5) = 1$ (modulus is the remainder of a division operation and $\frac{81}{5}$ results in 16 even divisions and 1 remainder), the result is 1.

Another way to think of modular arithmetic is to think of it in the same way we think of numeric bases (such as binary, octal, decimal, hexadecimal) and using only the least significant digit of that base. The value 81 in base 5 is 311, the least significant digit of which is 1 and therefore $81(\mathbb{Z}5) = 1$.

1.2.3 RSA Encryption

RSA encryption is a form of public key encryption and therefore uses the public key. The RSA encryption function is defined as:

$$plain^{publicExponent}(\mathbb{Z}publicModulus) = cipher$$

For example:

$$26729^{65537}(\mathbb{Z}37837) = 36784$$

is modular exponentiation of the plain text 26729 raised the power of 65537, which is the public exponent, all modulo the public modulus of 37837, which results in the cipher text 36784. The above example is also an example of encrypting the value 26729 (which is 0x6869 in hexadecimal, or "hi" in ASCII) with an RSA key whose public exponent is 65537 and whose public modulus is 37837 resulting in the encrypted value of 36784.

This can only be reversed using the private key as the exponent instead of the public exponent. That is:

$$cipher^{privateExponent}(\mathbb{Z}publicModulus) = plain$$

For example:

$$36784^{40193}(\mathbb{Z}37837) = 26729$$

1.2.4 RSA Digital Signatures

Looking closer at the RSA encryption example of modular exponentiation, we can see some of the public key cryptography properties that we need starting to emerge. Specifically we cannot take the cipher text value and convert it back to the plain text with just the public information. We must know the private exponent (or be able to derive it by factoring the public modulus, but given a complex and correctly generated key pair this should be impossible) to reverse the operation.

However RSA encryption is not very useful for digital signatures because it requires the private key to do anything meaningful. With a digital signature, we want to perform an operation on some plain text using our private exponent that can be verified using only the public key. Fortunately RSA allows us to do that by simply swapping the values around. That is:

$$plain^{privateExponent}(\mathbb{Z}publicModulus) = cipher$$

For example:

$$12345^{40193}(\mathbb{Z}37837) = 3293$$

which CAN be reversed using only public information, as in:

$$cipher^{publicExponent}(\mathbb{Z}publicModulus) = plain$$

For example:

$$3293^{65537}(\mathbb{Z}37837) = 12345$$

RSA guarantees that the chance of there being another private key which generates the same cipher text for a given plain text is extremely low relative to the size of the key. Therefore we can assume that if a given cipher text can be decrypted to a plain text using a given public key then it must have been encrypted with the secretly held private key and thus as long as the private key is protected as a secret, only the holder could have generated this message.

1.3 Digital Signatures

Directly encrypting a message with a private key is a workable solution for short messages but not in the general case. RSA and most other public key cryptography systems cannot encrypt (with either the public key or the private key) messages larger than the size of the key. That is, if the key is 16 bits then the plain text message can be no larger than 16 bits. For RSA the specific limit is that the value can be no larger than the public modulus due to the fact that all operations are modulo the public modulus.

Instead of directly encrypting the plain text a cryptographically secure message digest algorithm such as MD5[3], SHA1[4], or SHA256[5] is used to compute a cryptographically secure digest of the message which is typically much smaller than the message itself. This digest is then encrypted with the private key and verified with the public key. In this way, arbitrarily long messages can be digitally signed.

1.4 Encryption

The same limitation that exists for message length with respect to digital signatures also applies to encryption. The message encrypted with RSA must be no larger than the public key. Again, for RSA this is due the fact that the public modulo is applied to all operations.

Thus for the general case of encryption of an arbitrarily long message using a public key cryptography system two different cryptographic algorithms are typically used:

- A public key (asymmetric) cryptography key
- A symmetric key

Instead of directly encrypting the plain text a symmetric cipher such as AES[6], ARCFOUR/RC4, or 3DES[7] is used with a secure randomly generated symmetric cipher's key. This symmetric key is then encrypted with the asymmetric cipher's public key. The plain text is then encrypted with the selected symmetric cipher and the generated symmetric cipher's key.

1.5 Certificates

As previously mentioned in section 1, certificates are used to certify that one entity (the issuer of the certificate) says that another entity (the subject of the certificate) is a given identity under certain conditions.

So what makes up a certificate ? Certificates are specified by the ITU-T standard X.509 and contain the following information:

[3] MD5 is defined in RFC 1321
[4] SHA1 is defined in RFC 3174
[5] SHA256 is defined in RFC 4634
[6] AES is defined in FIPS PUB 197
[7] 3DES is defined in FIPS PUB 46-3

- X.509 Standard version number (optional) which identifies the revision of X.509 that this certificate complies with
- Issuer, which is the "distinguished name" of the entity who issued (that is, signed) the certificate
- Serial Number, which is a unique number per issuer to uniquely identify this certificate from the issuer
- Subject, which is the "distinguished name" of the entity who is being certified (and also who holds the private key)
- Issue date and expiration date, which define the time frame in which the certificate is valid
- The public key, including the algorithm and algorithm-specific public key data – for RSA this is the public modulus and public-exponent
- If this is X.509 version number 3 then X.509 extensions may be specified which restrict the uses of this certificate
- The digital signature of all of the previous data encoded in ASN.1 Distinguished Encoding Rules (DER) as specified by ITU-T standard X.690, which for RSA is the cryptographic message digest of the previous data, which is then padded (per PKCS#1), and finally encrypted with the private key of the issuer

So now we have a system where an unknown and untrusted party can assert to you a fully-qualified (also known as "distinguished") name and also provide a reference to who is certifying that this assertion is true within the parameters specified in the certificate.

What is to stop someone from taking a certificate from an existing entity, which must be publicly accessible otherwise there would be no way to identify the entity, and using it? Once again, public key cryptography is the answer here. A certificate only proves that the Issuer signed a request for the Subject. A certificate does not prove that you you are talking to is legitimately the subject specified in the certificate but it *does* provide a mechanism to do that. The public key being certified is in the certificate so all that is needed is for you to issue some sort of challenge for the party presenting the certificate to prove that it has the private key that corresponds with the certified public key. How this challenge is done depends on the protocol and is outside the scope of this document.

1.6 Finally... PKI

At this point we have described all of the vital components to a simple PKI system:

- A method to identify an entity
- A method for one entity to assert the identity of another entity

With these simple tools we can construct a system where we trust few entities to act as authoritative sources of identity information for unknown entities. Entities which act as authoritative sources of trust are known as Certificate Authorities and are identified by certificates with the X.509v3 extension known as "Basic Constraints" set to the value of "true". Certificate authority certificates may be signed by other certificate authorities or they may be self-signed. A self-signed certificate authority certificate is known as a "root certificate authority certificate" while a certificate authority certificate signed by another certificate authority is known as an "intermediate certificate authority certificate".

The result of such a system is a hierarchy of certificate authorities which can issue certificates. If trust is given to one of the certificate authorities then any certificates issued by that certificate authority (directly, or indirectly through a chain of subordinate or intermediate certificate authorities) can also be trusted as having met the requirements of that certificate authority.

1.7 More Complete PKI

While a simple PKI system is straight-forward to describe, a more complete (and secure) PKI system is more complex due to things such as revocation lists (CRLs) and the many X.509v3 extensions which can be used to limit the utility of X.509 certificates. This document will not cover those items due to their complexity and scope.

2 PKI with Tcl

Tcl version 8.5 and newer transparently support arithmetic on arbitrarily big integers which enables us to write a pure Tcl implementation of RSA using the [expr] command. Versions prior to 8.5 could use the *bigmath* package within TCLLIB at a considerable cost for speed.

2.1 The pki Package

The *pki* package is a module within TCLLIB, a collection of popular TCL-based packages. It requires Tcl version 8.5 for its big integer support.

The *pki* package provides an interface for most things related to PKI. It originally started as just an RSA package but as additional needs for interoperability arose more and more PKI support was added.

Currently it only supports the RSA public key cryptography system but is extensible to support additional algorithms (and indeed this is how PKCS#11 support is implemented).

The rest of this section briefly describes the interface to the *pki* package.

2.1.1 pki::encrypt

The [pki::encrypt] command encrypts a message with a key from a public key cryptography algorithm, such as RSA. Either the public key or the private key may be used to encrypt the message. It is worth noting that not every public key cryptography algorithm supports encryption and decryption. Notable the Digital Signature Algorithm[8] (DSA) only supports creating and verifying signatures.

The encrypted message is returned on success and an error is raised upon failure. In general, assuming a valid key has been supplied, the most common error returned is that the message is larger than the key and the algorithm does not support that.

The [pki::encrypt], [pki::decrypt], [pki::sign], and [pki::verify] commands each examine the key to determine what backend to call to handle the specific operation. The RSA backend is always registered as it is a part of the "pki" package. Additional backends may be registered at run-time.

2.1.2 pki::decrypt

The [pki::decrypt] command decrypts a previously encrypted message with a key from a public key cryptography algorithm, such as RSA. Either the public key or the private key may be used to decrypt the encrypted message.

The decrypted message is returned on success and an error is raised upon failure. It is worth noting that the decrypted message returned may be invalid if the key is not valid, however this is usually caught due to RSA PKCS#1 v1.5 padding on encrypted messages.

[8] DSA is specified in FIPS 186-4

2.1.3 pki::sign

The [pki::sign] command creates the digital signature of a message with a key from a public key cryptography algorithm, such as RSA. This will require the key supplied to include the private key.

The signature is returned on success and an error is raised upon failure.

2.1.4 pki::verify

The [pki::verify] command verifies that a digital signature is valid for a given message, signature, and key.

If the message can successfully be verified then "true" is returned otherwise "false" is returned.

2.1.5 pki::pkcs::parse_key

The [pki::pkcs::parse_key] command loads the supplied PKCS#1 key pair (or just the public key, if requested) into a key structure used internally by the "pki" package. If the key is encrypted a password may be supplied to decrypt it. If the key is encrypted and no password is supplied an error is raised.

The key is returned on success and an error is raised upon failure.

2.1.6 pki::pkcs::create_csr

The [pki::pkcs::create_csr] command creates the supplied PKCS#1 certificate signing request (CSR) with a specified key and the request subject given. A certificate signing request is the standardized message format handled by certificate authorities to create a certificate. It includes the public key, the requested name, and a signature of the entire request.

2.1.7 pki::pkcs::parse_csr

The [pki::pkcs::parse_csr] command reads the supplied PKCS#1 certificate signing request (CSR) and returns the public key information as well as the requested name as a single key object.

2.1.8 pki::x509::parse_cert

The [pki::pkcs::parse_cert] command reads the supplied X.509 certificate and returns the public key information as well as the other information in the certificate in a single certificate object, which may also be used as a public key object. At this point is worth nothing that the key object used internally by the

"pki" package is just a Tcl dictionary (dict) and may freely be accessed as such.

2.1.9 pki::x509::verify_cert

The [pki::x509::verify_cert] command verifies that a certificate is properly signed by a trusted certificate authority, which may either be a root certificate authority or an intermediate certificate authority. It is worth noting that this does not verify that the certificate may be used for any particular purpose or even that it may be used at this time, but only that it was legitimately issued by a trusted certificate authority.

If the certificate can be verified to have been issued by a trusted certificate authority then "true" is returned otherwise "false" is returned.

2.1.10 pki::x509::validate_cert

The [pki::x509::validate_cert] command verifies that a certificate may be used for some purpose, such as SSL/TLS, being a certificate authority responsible for signing a subject. It checks the parameters of the X.509v3 certificate such as validity period (issue date and expiration date), the subject distinguished name, the "Basic Constraints" extension, and other attributes based on how it is invoked.

If the certificate can be determined to be valid for the specified purpose then "true" is returned otherwise "false" is returned.

2.1.11 pki::x509::create_cert

The [pki::x509::create_cert] command creates an X.509 certificate from a certificate signing request using a specified certificate authority key to sign it, a specified serial number, and with the given additional X.509 parameters. The process of creating a certificate is sometimes called "signing a certificate" which is a misnomer since there is no such thing as an unsigned certificate[9] in X.509.

The parameters to [pki::x509::create_cert] are rather unwieldy at this point due to the number of required parameters and newer versions will likely replace the positional parameters with named parameters.

Upon success the certificate is returned in either PEM or DER format which are suitable for exchange with other PKI systems. It must be parsed with [pki::x509::parse_cert] before it can be used internally with the "pki" package.

[9] Indeed, an unsigned certificate would certify nothing

2.1.12 pki::rsa::generate

The [pki::x509::generate] command generates an RSA key of a given size. You can also optionally specify the public key exponent to use rather than the default of 65537, but it is not advisable to actually do so since it may decrease the security of the generated key.

2.2 The "pki::pkcs11" Package

The "pki::pkcs11" package uses and extends the "pki" package with support for RSA PKCS#11 hardware security modules (HSMs) such as cryptographic accelerators or smart-cards. Unlike the "pki" module it is not written in Tcl, but is written in C. This is due to the fact that the RSA PKCS#11 standard specifies a C API for "cryptoki modules".

The "pki::pkcs11" module has a relatively simple interface thanks to re-using most of the "pki" package for operations and supporting only the most basic RSA PKCS#11 functionality.

2.2.1 pki::pkcs11::loadmodule

The [pki::pkcs11::loadmodule] command loads a "cryptoki module", which is an RSA PKCS#11 compliant library (DLL or shared object, for example) that will be used for accessing a specific hardware device.

If the specified module can be successfully loaded an opaque handle is returned otherwise an error is raised.

2.2.2 pki::pkcs11::unloadmodule

The [pki::pkcs11::unloadmodule] command unloads and frees allocated structures for an RSA PKCS#11 cryptoki module specified by the opaque handle. After the specified module is unloaded the opaque handle may no longer be used.

Upon success "true" is returned otherwise "false" is returned.

2.2.3 pki::pkcs11::listslots

The [pki::pkcs11::listslots] command lists the slots that are available for a given opaque cryptoki module handle. In RSA PKCS#11 a slots contain at most one token which can contain objects such as certificates, public keys, private keys, etc.

Upon successful operation a list is returned. The returned list contains one element per slot. Each item of the returned list is itself a sub-list containing the following items:

1. The slot identification number for the slot
2. The label of the slot
3. Flags set for the slot

If there is an error in processing the request then an error is raised. If there are no slots available then an empty list is returned.

2.2.4 pki::pkcs11::listcerts

The [pki::pkcs11::listcerts] command lists the certificate objects available for a given handle and slot identification number.

Upon successful operation a list is returned. Each item of the returned list contains a certificate/key object (as would be returned by [pki::x509::parse_cert]). If there is an error in processing the request then an error is raised. If there are no certificate objects for a given slot identification number associated with a given opaque handle then an empty list is returned.

2.2.5 pki::pkcs11::encrypt

The [pki::pkcs11::encrypt] command is not intended to be called directly by end-user applications. It calls the C_Encrypt() function within the loaded RSA PKCS#11 cryptoki module. Instead of calling this command end-user applications should call the [pki::encrypt], [pki::decrypt], or [pki::sign] command which will invoke this command if the key supplied indicates it is a PKCS#11 module.

Because RSA PKCS#11 cryptoki modules expose functions to perform public key cryptography they do not need to export the private key in order for applications to perform cryptographic operations that use the private key. For example a user using a smart-card can prove that he has access to his private key and thus legitimately is associated with the certificates presented without being able read to the private key at all. Instead the smart-card performs the cryptographic operation and returns a cryptographic result. This means that there is no way for an adversary to acquire the private key for a smart-card[10] user since the user themselves cannot read the private key.

2.2.6 pki::pkcs11::decrypt

The [pki::pkcs11::decrypt] command, like the [pki::pkcs11::decrypt] command, is not intended to be called directly by end-user applications.

[10] Except for the possibility of physically acquiring the device

2.2.7 pki::pkcs11::login

The [pki::pkcs11::login] command logs into a device by calling the RSA PKCS#11 cryptoki module's C_Login() function. Because cryptographic modules perform cryptographic operations using the private key they will often be require the user to verify to the hardware security module or smart-card that they legitimately should be able to perform that operation by supplying a password. If a login is required to use a particular hardware token then the LOGIN_REQUIRED flag will be set in the result from [pki::pkcs11::listslots] for the slot that the token is in.

If the password successfully logs into the device then "true" is returned. If the password is incorrect then "false" is returned. If some other error condition is asserted then an error is raised.

2.2.8 pki::pkcs11::logout

The [pki::pkcs11::logout] command logs out of a device by calling the RSA PKCS#11 crytptoki module's C_Logout() function. Once you are logged out, token attempts to perform cryptographic operations will probably fail.

2.3 Doing Something Useful

2.3.1 Establishing a Simple Certificate Authority

The very first thing we need to be concerned with when establishing a public key infrastructure system is the establishment of an authority to certify identities. This is our certificate authority. All we need for a minimal certificate authority is certificate authority certificate and some way to ensure that we do not issue certificates with duplicate serial numbers. We can do this from within Tcl using the "pki" module easily.

First we must load the "pki" package, which can be obtained from TCLLIB or from http://rkeene.org/devel/pki/.

```
% package require pki 0.3
```

Then we need to generate our private key. Since RSA is the only public key cryptography algorithm currently implemented we should use that:

```
% set ca_key [pki::rsa::generate 2048]
```

Next we need to "sign our key" which will necessarily be stored as a certificate (a key that was simply signed would not include any identifying information and would be generally useless). Since this key will be our first certificate authority certificate it must be signed by its own key and is therefore self-signed and also therefore a root certificate authority. The "pki" package does not provide a nice way to handle certificate authorities. To accomplish that we just add a "subject" key to the "ca_key" dictionary:

```
% set ca_key_subject \
    "CN=Example Root CA"
% dict set ca_key subject \
    $ca_key_subject
```

After we have updated this object we can use sign it using the [pki::x509::create_cert] command:

```
% set issue_date [clock seconds]
% set expire_date [clock add \
    $issue_date 1 year]
% set ca_cert
    [pki::x509::create_cert \
    $ca_key $ca_key 1 $issue_date \
    $expire_date 1 [list] 1]
```

At this point we have successfully established our certificate authority by creating a private key and a signed certificate which identifies us. We should save our private key, which is stored in the dictionary named "ca_key", and our certificate which is represented by the value stored in the variable "ca_cert" somewhere to prevent losing them and also in such a way that others may not access them.

2.3.2 Provisioning User Certificates

Now that we have a certificate authority, our users may start using it. The first thing they will want to do is obtain a copy of our certificate authority certificate through a secure and trusted channel in order to establish trust with it. After that they will also want to generate their own RSA private key:

```
% package require pki 0.3
% set user_key [pki::rsa::generate 2048]
```

Next the user should generate a certificate signing request (CSR) so that the certificate authority will know the public key as well as what identity the user is claiming to be. This is done with the [pki::pkcs::create_csr] command:

```
% set user_csr [pki::pkcs::create_csr \
    $user_key [list "CN" "Joe User"] 1]
```

Then the user can distribute the CSR to the certificate authority over a trusted channel (although all of the information in the CSR is public if the channel is compromised, a malicious man-in-the-middle could alter the request so that his or her public key be used instead) and the certificate authority can generate a certificate:

```
# (On Certificate Authority)
% set user_csr {//From User//}
% set user_csr_list \
    [pki::pkcs::parse_csr $user_csr]
% set issue_date [clock seconds]
% set expire_date [clock add \
    $issue_date 1 year]
% set user_cert \
    [pki::x509::create_cert \
    $user_csr_list $ca_key 2 \
    $issue_date $expire_date 0 [list] \
    1]
```

The certificate authority can then give the user his or her certificate over any channel since it's public.

2.3.3 Securely Exchanging Messages

Once several users have certificates issued by our certificate authority they can use the trust they have established with the certificate authority and the certificate authority certificate to validate messages between them, creating a secure and trusted channel between two users who may have never previously communicated.

This can be done by creating a simple ad-hoc protocol for the users to communicate where users:

1. Initially exchange certificates with the peer
2. Each side performs some action to verify that the Subject of the certificate is who they are talking to
3. Each side validates that the certificate is valid using [pki::x509::validate_cert]
4. Each side securely generates a 128-bit key for AES using the "aes" package (from TCLLIB)
5. Each side encrypts the 128-bit key as well as the initialization vector (IV) with the peer's public key, which can only be successfully decrypted by the peer
6. Each side decrypts the received 128-bit key and IV using his or her private key

7. Each side initializes an AES chain block cipher mode stream for sending encrypted blocks to its peer and another AES chain block cipher mode stream for receiving encrypted blocks

3 Legal Information

Since the *pki* package implements RSA, which is strong cryptography, it must be registered with the United States Department of Commerce. Currently the *pki* module is registered with Export Registration Number (ERN) R103416 and is authorized for export and re-export from the United States. No effort has been made to ensure that it can be imported to- or exported from other countries.