OpenSSL

Julian Sequeira and Shayan Ghazi

RSA Overview

1. Select two very large prime numbers p and q.
2. Compute $n = pq$ and $\phi(n) = (p - 1)(q - 1)$.
3. Choose an encryption key e relatively prime to $\phi(n)$.
4. Calculate the decryption key d such that $ed = 1 \pmod{\phi(n)}$.
5. Publish e and n, and keep d, p, and q secret.

RSA Overview (2)

- Encryption:

$$C = M^e \bmod n$$

- Decryption:

$$M = C^d \bmod n$$

- Modulus, n of length b bits
- Public exponent, e
- Private exponent, d
- Prime1, p, and Prime2, q

M: Message

C: Ciphertext (encrypted string)

n = p*q

RSA Overview (3)

- Exponent1, $d_p = d \pmod{p-1}$
- Exponent2, $d_q = d \pmod{q-1}$
- Coefficient, $q_{inv} = q^{-1} \pmod{p}$
- Private key, $PR = (n, e, d, p, q, d_p, d_q, q_{inv})$
- Public key, $PU = (n, e)$

Only for efficiency

RSA Overview (4)

1. **Prime generation is easy** - it's easy to find a random prime number, even large ones
2. **Multiplication is easy** - given p and q, it's easy to calculate $n = pq$
3. **Modulo inverse is easy** - given e and $\varphi(n)$, it's easy to calculate d s.t. $ed \bmod \varphi(n) = 1$
4. **Modular exponentiation is easy** - given n, m and e, it's easy to compute $c = m^e \bmod n$
5. **Prime factorization is hard** - given n it's hard to find primes p and q such that $pq = n$
6. **Modular root extraction is hard** - given n, e and c, it's difficult to recover m such that $c = m^e \bmod n$, without knowing p or q (or d).

RSA Overview (5)

- Often e is picked first, then $n = pq$ is chosen such that $gcd(e, \varphi(n)) = 1$
- The default value of e in OpenSSL is *65,537*
- This is a **Fermat prime**, with the form $2^{(2^k)} + 1$, where $k = 4$
- This allows calculating m^e to be faster - you square m, $k+1$ times, and then multiply the result by m.

Demo

Using OpenSSL to securely communicate between two parties

Demo: Create Alice's private key

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048
-pkeyopt rsa_keygen_pubexp:3 -out privkey-A.pem
```

This command creates a generates a private key (**genpkey**) with a specified algorithm (**RSA**) we provide the size of **n** in bits and and the value of **e** and the output file (**privkey-A.pem**)

Demo: Create Alice’s public key

```
openssl pkey -in privkey-A.pem -pubout -out pubkey-A.pem
```

This command takes in a private key (**pkey -in**) with a specified name (**privkey-A.pem**) and the outputs a public key (**-pubout**) with specified name (**pubkey-A.pem**)

Demo: How to view private keys (as text)...

```
openssl pkey -in privkey.pem -text -noout | less
```

This command takes in a private key (**privkey.pem** in example above). We want to view as **-text** and we pipe into **less** for easy sequential viewing. The **-noout** prevents the base64 encoding from being printed as well.

Demo: How to view public keys (as text)...

```
openssl pkey -in pubkey.pem -pubin -text -noout | less
```

This command takes in a public key (using **-pubin**, **pubkey.pem** in example above). We want to view as **-text** and we pipe into **less** for easy sequential viewing.

Demo: Alice creates a signing request

```
openssl req -new -key privkey-A.pem -out A-req.csr
```

Here you specify details for a certificate you want to create. It will prompt you for your country, organization, email, etc... This request will be processed by a certificate authority, who will generate a certificate for Alice.

Demo: Generate the certificate

```
openssl x509 -req -in A-req.csr -CA root.crt -CAkey root.key
-CAcreateserial -out Alice.crt -days 500 -sha256
```

The Certificate Authority generates a certificate using the **x509** utility. It takes in a request (**-req -in A-req.csr**) and outputs to **A.crt**. We specify the length of its validity (**-days 500**). **-CAcreateserial** creates a serial file and assigns a serial number to the certificate. A fingerprint is created using the **sha256** algorithm

Demo: View the certificate

```
openssl x509 -in Alice.crt -text -noout
```

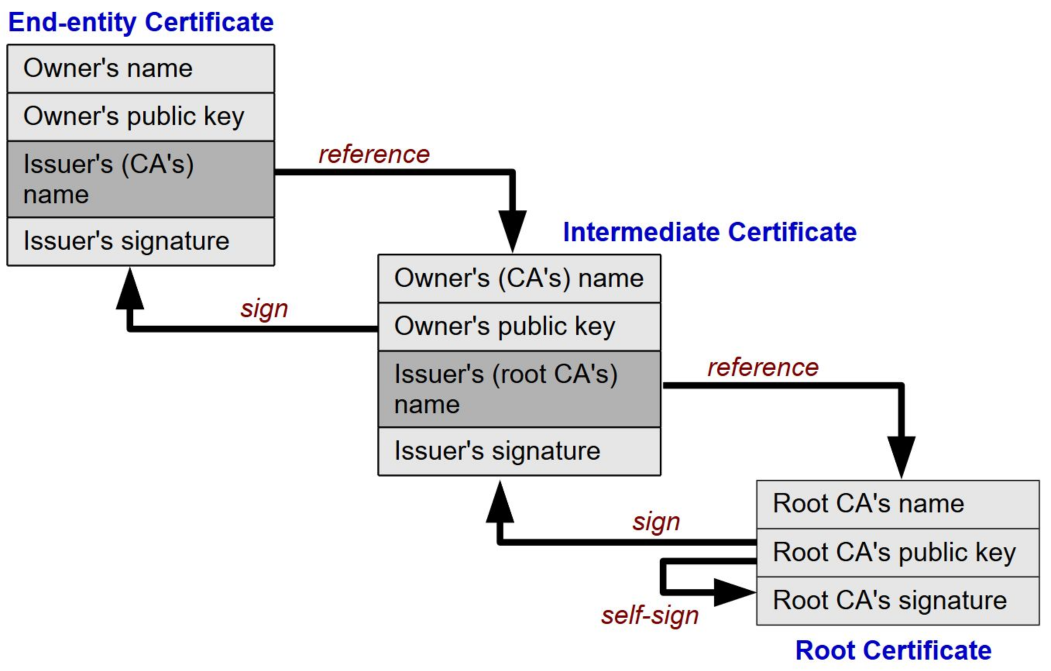
This command allows you to view the certificate in text. The **-noout** option is so the base64 encoding of the certificate does not also get printed. Using this you can view details such as it's expiry date, serial number, issuer, etc…

Demo: Verify the certificate against the CA

```
openssl verify -CAfile root.crt Bob.crt
```

The **verify** utility can be used to check if Bob's certificate was signed by the certificate authority's certificate (**-CAfile root.crt**). You should get a **B.crt: OK** if the certificate can be trusted.

Chain of Trust



Demo: Extract a public key from a certificate

```
openssl x509 -pubkey -in Bob.crt -noout > pubkey-B.pem
```

Extract the public key using the **-pubkey** option and put it in **pubkey-B.pem**

Alice now has Bob’s public key, and can use it to encrypt files to send to Bob.

Demo: Alice encrypts her message

```
openssl pkeyutl -encrypt -in largefile.txt -pubin -inkey
pubkey-B.pem -out ciphertext.bin
```

Using utilities (**pkeyutl**) Alice will encrypt (**-encrpyt**) the message.txt file using a public key (**-pubin**). She will use Bob's public key (**-inkey pubkey-B.pem**) so only Bob can decrypt the file. Alice stores the output in ciphertext.bin.

Demo: Error!

```
Public Key operation error - data too large for key
size:rsa_pk1.c:153:
```

RSA can only encrypt data smaller than the key length- it is not for encrypting arbitrarily large files! The solution here is to use symmetric key encryption- Alice can generate a symmetric key and share it with Bob using RSA.

Demo: Alice generates a random key

```
openssl rand -base64 32 -out symkey.pem
```

The rand command is a pseudo-random byte generator. It is seeded using the $HOME/.rnd file, and can take in additional seed sources using the -rand flag. Alice outputs 32 random bytes to **symkey.pem** and encodes it in base64.

Demo: Alice encrypts her symkey

```
openssl pkeyutl -encrypt -in symkey.pem -pubin -inkey
pubkey-B.pem -out symkey.enc.pem
```

Using utilities (**pkeyutl**) Alice will encrypt (**-encrpyt**) the symkey.pem file using a public key (**-pubin**). She will use Bob's public key (**-inkey pubkey-B.pem**) so only Bob can decrypt it. Alice stores the output in symkey.enc.pem.

Demo: Alice’s encrypted signature

```
openssl dgst -sha1 -sign privkey-A.pem -out signature.bin
symkey.pem
```

Alice will hash the symkey using the sha256 algorithm (**dgst -sha256**). She then encrypts the hash with her private key (**-sign privkey-A.pem**). The output is **signature.bin**

Demo: Alice transmits her encrypted message and signature

```
cp signature.bin ../Bob

cp symkey.enc.pem ../Bob
```

We want to send our (Alice's) encrypted message (**ciphertext**) and signature (**signature.bin**) to Bob. For the demo, we'll simply copy the files over to Bob's folder - but you can imagine this happening over a network via some transfer protocol.

Demo: Bob decrypts Alice's message

```
openssl pkeyutl -decrypt -in symkey.enc.pem -inkey privkey-B.pem
-out symkey.pem
```

Using utilities (**pkeyutl**), Bob can decrypt (**-decrypt**) taking in (**-in**) the file to decrypt (**ciphertext.bin**) using a provided key (**-inkey privkey-B.pem**). Finally, the decrypted text is outputted (**-out**) to (**symkey.pem**)

Demo: Bob verifies message is from Alice

```
openssl dgst -sha1 -verify pubkey-A.pem -signature signature.bin
symkey.pem
```

For Bob to verify the message, he applies the same hash that Alice used for her signature (**dgst -sha1**) and uses her public key (**-pubkey-A.pem**) to verify (**-verify**) the provided signature (**-signature signature.bin**) by decrypting it and comparing the result to the hashed, decrypted message (**symkey.pem**)

Demo: Alice encrypts the largefile using AES

```
openssl enc -aes-256-cbc -pass file:symkey.pem -p -md sha256 -in
largefile.txt -out ciphertext.bin
```

The **enc** utility is used for symmetric key encryption, with **aes-256-cbc** specified as the algorithm. The **key derivation function** uses **sha256**, and the **-p** flag will print the key, salt and initialization vector to the screen.

Demo: Alice sends the ciphertext to Bob

```
cp ciphertext.bin ../Bob
```

Again, we’ll simply copy it over for the demo, but this can be done over a network just as easily.

Demo: Bob decrypts the ciphertext

```
openssl enc -aes-256-cbc -d -pass file:symkey.pem -p -md sha256
-in ciphertext.bin -out largefile_received.txt
```

Bob will decrypt the ciphertext using a similar command that Alice used to encrypt it. Note: Bob needs to know what encryption Alice used for this to work. He will use the **-d** flag to specify decryption and put it in **largefile_received.txt**.

Demo: A summary - what happened?

1. Alice generated public and private keys
2. Alice generated a certificate signing request- the CA took it and made her a certificate
3. Alice extracted Bob’s public key from Bob’s certificate
4. Alice generated a symmetric key and encrypted it with Bob’s public key (symkey.enc.pem)
5. Alice hashed the symmetric key, then encrypted her hash with her private key (signature.bin)
6. Bob decrypted symkey.enc.pem with his private key to get symkey.pem
7. Bob decrypted signature.bin with Alice’s public key, hashed symkey.pem and compared them
8. Alice used the symkey to encrypt largefile.txt
9. Bob used the symkey to decrypt largefile.txt

References

https://prefetch.net/articles/realworldssl.html

http://www.cs.toronto.edu/~arnold/347/17f/lectures/crypto/

https://www.openssl.org/docs/

https://www.openssl.org/docs/manpages.html

https://www.johndcook.com/blog/2018/12/12/rsa-exponent/

https://jamielinux.com/docs/openssl-certificate-authority/create-the-root-pair.html

https://gist.github.com/fntlnz/cf14feb5a46b2eda428e000157447309

Extra: Root CA makes self-signed certificate

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024
-out rootCA.crt
```

Here is a command to self-sign a certificate, we used this for our pseudo root CA (the trusted third-party).