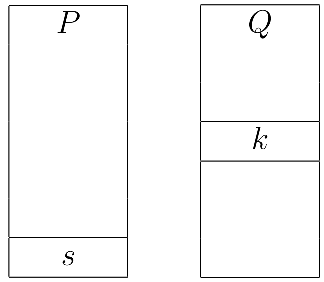
Contents

1 Signing algorithm

Consider two certificates P and Q where P is signed by Q. Let s be the signature in P and let k be the public key in Q.



By necessity s and k are compatible. For example, if k is RSA 2048 then s is a PKCS[1] signature that is 2048 bits in length (256 bytes).

A hash digest of P is contained in s. For example, the following unencrypted signature s is for RSA 2048 and hash digest SHA256. (Numerals are in hexadecimal.)

`00 01 ff ··· ff 00 30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20`	HASH

Signature s (plaintext)

The length of HASH is 32 bytes because SHA256 is used. The 19 byte sequence starting with `30` is from "Abstract Syntax Notation One." The `ff` bytes are pad bytes that are added to make the total length of the signature 2048 bits (256 bytes). Hence there are $256 - 3 - 19 - 32 = 202$ pad bytes.

Note that the above signature is the unencrypted value of s. In the actual certificate, s is encrypted using the private key associated with k. After encryption, s is still 256 bytes long.

`76 b6 97 82 0f 06 b7 48 59 02 a0 2c f4 ··· fe c3 61 25 5b 1c da 77 9a a1 63 d4 49 cd`

Signature s (encrypted)

To prove that P is signed by Q, s is decrypted using Q's public key k. Then if HASH matches a digest of P, the signing of P by Q is proven.

Proving "P signed by Q" proves that s was encrypted using the private key associated with k. Only the owner of Q knows the private key. No one can change the contents of P without breaking the hash digest in s, and no one can change s without knowing the private key. Hence we can trust the contents of P if we trust Q.

[1] RFC 3447 Public-Key Cryptography Standards (PKCS)

2 RSA functions

```
void rsa_encrypt_signature
          (uint8_t *sig, int len, struct keyinfo *key)
```

Encrypts signature `sig` in situ using the private key in `key`. Argument `len` is the length of the signature in bytes. This function is used to sign a certificate.

```
uint8_t *rsa_decrypt_signature
          (struct certinfo *p, struct certinfo *q)
```

Decrypts the signature in certificate `p` using the public key in `q` and returns the result. On success, the result is returned in a malloc'd buffer that the caller should free. The length of the buffer is the same as the signature length. Returns NULL on error.

3 Elliptic curve functions

3.1 prime256v1

```
void ec256_encrypt
          (struct keyinfo *key, uint8_t *hash, int len, uint8_t *sig)
```

Encrypts `hash` using the private key in `key`. The 64 byte result is returned in `sig`. This function is used to sign a certificate.

`key`	Pointer to a private key.
`hash`	Pointer to a hash digest value.
`len`	Byte length of hash digest value (32 bytes maximum).
`sig`	Pointer to a 64 byte buffer.

```
int ec256_verify
          (struct certinfo *p, struct certinfo *q)
```

Returns 0 if certificate `p` is signed by `q` where the public key type of `q` is prime256v1.

3.2 secp384r1

```
void ec384_encrypt
          (struct keyinfo *key, uint8_t *hash, int len, uint8_t *sig)
```

Encrypts `hash` using the private key in `key`. The 96 byte result is returned in `sig`. This function is used to sign a certificate.

`key`	Pointer to a private key.
`hash`	Pointer to a hash digest value.
`len`	Byte length of hash digest value (48 bytes maximum).
`sig`	Pointer to a 96 byte buffer.

```
int ec384_verify
          (struct certinfo *p, struct certinfo *q)
```

Returns 0 if certificate `p` is signed by `q` where the public key type of `q` is secp384r1.

4 Hash functions

```
void md5(uint8_t *buf, int len, uint8_t *out)
void sha1(uint8_t *buf, int len, uint8_t *out)
void sha224(uint8_t *buf, int len, uint8_t *out)
void sha256(uint8_t *buf, int len, uint8_t *out)
void sha384(uint8_t *buf, int len, uint8_t *out)
void sha512(uint8_t *buf, int len, uint8_t *out)
```

The hash value of `buf` is returned in `out`. Argument `len` is the length of `buf` in bytes. The following table shows the byte length of each result.

Digest	Length
MD5	16 bytes
SHA1	20
SHA224	28
SHA256	32
SHA384	48
SHA512	64

The hash value of a certificate is encrypted in the certificate signature.

5 ASN.1 Encodings

Encoded integers may have an extra leading 00 byte added due to encoding rules. Encoding rules add the 00 byte if the first byte of the unencoded integer is greater than 127.

For example, a 256 byte RSA modulus that begins with hex 80 is encoded as 257 bytes.

5.1 RSA

5.1.1 Public key

The following example is for RSA 1024 which has a 128 byte modulus. The two integers in the encoding are the modulus followed by the exponent.

```
134 159:     SEQUENCE {
137  13:       SEQUENCE {
139   9:         OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
150   0:         NULL
       :         }
152 141:       BIT STRING, encapsulates {
156 137:         SEQUENCE {
159 129:           INTEGER
       :             00 C4 B9 4C 7D 39 E6 64 4E 66 88 92 60 24 AC C8
       :             C7 80 19 3D B9 05 A9 0D 56 EE 2D B6 DB D9 73 C2
       :             0C 97 6F E8 08 FE DC 5B 55 0A 5E FA B0 66 F7 B2
       :             EA 78 31 45 4C 58 3D 49 F2 09 AF C8 37 81 86 7D
       :             C1 55 1F 9F EA 8C DB ED 5B 28 2E 2D 7B CD 84 77
       :             4D 06 9D 57 E7 BE 23 6F 39 08 73 F4 3C 89 35 AF
       :             65 FE B1 C0 5B 19 A3 60 78 80 DB 07 6D 36 28 C8
       :             A0 EB CA 2D 5C 1D B2 A0 9C 59 0F 6E E2 AA 9D B5
       :             27
291   3:           INTEGER 65537
       :           }
       :         }
       :       }
```

5.1.2 Signature

The signature is the bit string field at the end of the certificate. The length of the signature is the same as the unencoded RSA modulus. In this case (RSA 1024) the length is 128 bytes.

```
296  13:   SEQUENCE {
298   9:     OBJECT IDENTIFIER sha256WithRSAEncryption (1 2 840 113549 1 1 11)
309   0:     NULL
       :     }
311 129:   BIT STRING
       :     7E 97 8C BA 48 9D 89 C8 98 E4 77 27 E4 01 54 4D
       :     65 BB 96 04 46 43 C2 4A F4 6C B3 19 9A 4B 56 28
       :     10 99 4C 5F 12 D9 3D 36 5F BC C7 C9 F2 28 1A 24
       :     5D 09 05 AA 85 17 18 E6 B0 1D C9 C6 CD 8A AE 10
       :     6F AA 9E 6E 09 B6 C5 85 54 2D CF 7B A0 B8 72 67
       :     86 9C 51 35 8D ED 17 8A 48 23 A4 82 53 3C BE 29
       :     5A 7D DA 00 B5 D3 9F EC 9A 6F 3D 95 B7 B4 64 72
       :     56 D1 3C DA 62 BD 4B 45 81 C2 D4 35 E1 06 DB 24
```

5.2 prime256v1

5.2.1 Public key

Bit string data is the byte sequence (04 | X | Y). The length of X is 32 bytes and the length of Y is 32 bytes.

```
162  89:     SEQUENCE {
164  19:       SEQUENCE {
166   7:         OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
175   8:         OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
       :         }
185  66:       BIT STRING
       :         04 AB AA 1E 30 A0 41 00 05 C5 7F 32 E3 99 B8 BE
       :         3B 8A C1 4A A2 A3 4C CB 3C 44 97 04 4D D2 99 F1
       :         E9 CD FE 63 B3 C4 B7 05 99 1B 94 1B 87 3B 47 BA
       :         3A 76 AA 37 96 2F 89 47 31 53 EB 77 E6 43 17 D2
       :         3B
       :       }
```

5.2.2 Signature

The two integers in the encoding are R followed by S. The unencoded length of R is 32 bytes and the unencoded length of S is 32 bytes.

```
253  10:   SEQUENCE {
255   8:     OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
       :     }
265  72:   BIT STRING, encapsulates {
268  69:     SEQUENCE {
270  33:       INTEGER
       :         00 9F 3F A9 AE 97 A0 48 52 AA AA AF 3E CA BA 62
       :         5F 6C 2C 46 BB 29 D0 19 A6 14 EA C0 5D 0E B9 B8
       :         D5
305  32:       INTEGER
       :         16 76 0A FC 4D 9C 6F 65 BD D2 8B CA EF C5 6E 07
       :         76 46 13 1D CF 39 A8 E3 80 D8 BD 2E B2 F9 89 0D
       :       }
       :     }
```

5.3 secp384r1

5.3.1 Public key

Bit string data is the byte sequence (04 | X | Y). The length of X is 48 bytes and the length of Y is 48 bytes.

```
163 118:     SEQUENCE {
165  16:       SEQUENCE {
167   7:         OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
176   5:         OBJECT IDENTIFIER secp384r1 (1 3 132 0 34)
       :         }
183  98:       BIT STRING
       :         04 42 62 D9 F6 76 24 10 AE 1B 60 1F 59 45 C5 7D
       :         69 89 A3 A7 29 92 40 E6 BF FD F0 D0 20 55 BD 97
       :         5E 2B D8 BB 14 56 30 08 6E F0 02 A8 DB F4 DD C5
       :         BD DD 69 AB 39 B9 32 FC 55 D4 D5 8C 70 8E 27 3C
       :         AA A0 72 67 22 AB 1D DF 41 B5 D4 99 6D 32 7C 06
       :         ED 48 9F 31 E5 BD 10 AA 09 E7 5B 19 B6 8D 23 43
       :         27
       :       }
```

5.3.2 Signature

The two integers in the encoding are R followed by S. The unencoded length of R is 48 bytes and the unencoded length of S is 48 bytes.

```
283  10:   SEQUENCE {
285   8:     OBJECT IDENTIFIER ecdsaWithSHA256 (1 2 840 10045 4 3 2)
       :     }
295 104:   BIT STRING, encapsulates {
298 101:     SEQUENCE {
300  48:       INTEGER
       :         33 30 98 0F AA 4C 83 A1 0C 17 F9 3F 2F 05 F7 92
       :         2B 97 E9 2E E5 63 33 26 29 36 10 4F 65 2F E4 BA
       :         FF 14 09 0E 6B 07 BC 3D 8C 62 E0 4E 9C 4E B4 37
350  49:       INTEGER
       :         00 F6 6E EC 4F F9 5B 37 DC 8D E3 E9 B3 CA 13 0C
       :         5A BE F4 72 E4 4B 7A B4 BF C7 05 F1 71 83 77 68
       :         DF CF F3 CA B2 3E C5 8F E6 7E 34 B7 B4 AB 6F D5
       :         4F
       :       }
       :     }
```

6 References

Certicom Corp., “Standards for Efficient Cryptography 1 (SEC 1: Elliptic Curve Cryptography)”

FIPS Publication 180-4, Secure Hash Standard

Kaliski B., “A Layman’s Guide to a Subset of ASN.1, BER, and DER”

NIST, “Mathematical routines for the NIST prime elliptic curves”

RFC 1321 The MD5 Message-Digest Algorithm

RFC 3447 Public-Key Cryptography Standards (PKCS)