Cryptographic standards

The theoretical principles of cryptographic primitives are probably nothing new for you, but once it comes to implementation, more than the basic principles are necessary. It is necessary to know how the keys are stored, how the cryptographically processed data are stored. Even the cryptographic processing of data does not have to be necessarily unique. Just to mention data padding or metadata. Let’s look at the particular cryptographic primitives.

Hash functions

Hash functions receive the input of arbitrary length and give an output of a fixed length (e.g. 128 or 160 bits). The output is binary and is usually coded as a hexadecimal string. To store several hashes a format, where each line includes the hash and the filename, is used (e.g. md5sum). For example:

```
a94a2480d3289e625eea47cd1b285758 pozadavek.aux
17c2e99ef02927c48e01aeba4498ceb3 pozadavek.log
980d433060a664d2d5269edff8fbc8dc pozadavek.pdf
bbfa0f49df9f6a580bff445aa027c1b8 pozadavek.tex
```

Some important hash functions:

- MD2 – defined v RFC 1319, output 128 bits
- MD4 – defined v RFC 1320, output 128 bits
- MD5 – defined v RFC 1321, output 128 bits
- SHA-0 (originally named SHA) – defined by NIST in FIPS 180, output 160 bits
- SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 – defined by NIST in FIPS 180-2, output 160 bits for SHA-1, for other the output length corresponds with the name of the hash function.
- RIPEMD – designed by academic community, default output 160 bits, also versions with the output of 128, 256 a 320 bits.

Remind that the message for MD4/MD5 and SHA-1/SHA-224/SHA-256 is padded (extended) to ensure that its length in bits plus 64 is divisible by 512 (i.e., its bit-length is congruent to 448 modulo 512). Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended to the required length, and a 64-bit binary representation of the original length of the message is then concatenated to the message. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. The message x is then processed in 512-bit blocks ($x_1, x_2, \ldots, x_t$) in the Damgård/Merkle iterative structure as follows: $H_0 = IV$; $H_i = f(H_{i-1}; x_i)$; $1 \le i \le t$; $h(x) = g(H_t)$, where IV is initialization value and function g is used to mapping the result to the required length (often identity).

Note: in the summer of 2004, serious flaws were found in the design of hash functions MD5, SHA-1 and RIPEMD. These flaws lead to finding a collision algorithm for MD5 and the reduction of the space for collision searching for SHA-1 from the expected 2^{80} (birthday paradox consequence) to 2^{63}.

Symmetric encryption functions

Based on a key shared by the sender and the recipient (or other 2 parties) symmetric encryption algorithms transform the plaintext into the ciphertext (and vice versa for decryption). The key is binary data of (usually) fixed length. The plaintext is arbitrary data, its structure is usually not important (but typically the plaintext is compressed before encrypting), ciphertext is binary data that can be coded to use only “safe” and printable ASCII characters (e.g. it is possible to use base64 coding).

Symmetric keys can be stored directly in files without internal structure (i.e. only these 128 bits of data) or in files further protected by password (e.g. PKCS#5 defines a way to encrypt data (typically keys) with the help of user-supplied password). Symmetric keys should be random data; some keys can have parity bits (e.g., DES; the least significant bit in each octet should ensure odd parity). Due to quite short lengths of symmetric keys it is possible to specify them directly as a command line parameter or into a dialog box. In such cases hexadecimal values are used. It is also possible to derive cryptographic keys from passwords (or pass phrases); typically hash functions will do the task.
Encrypted data can be stored directly or with supplemental information, which may include the original length, padding, hash or MAC of the data to verify that the decryption has been done correctly. To store the data it is possible to use the above mentioned PKCS#5 format or PKCS#7 format (see later), but many of the encryption algorithms use their own format and the resulting file is not compatible with other encryption applications (e.g. gpg).

Symmetric encryption functions can be divided into block functions and stream functions. Block ciphers can only encrypt data with the length of a multiply of the block length, stream ciphers can encrypt data of any length. Symmetric ciphers can be used in a variety of modes (as specified in FIPS 81):

- ECB (Electronic Code Book) – a block of the plaintext is encrypted without respect to other blocks (blocks can be freely shifted/exchanged, this fact can be misused for an attack)
- CBC (Ciper Block Chaining) – the output is dependent on all previous data (XOR with previous block is used), initialization vector is used (IV)
- CFB (Cipher Feedback Mode) – creates a stream cipher from block cipher, key stream is obtained by encrypting previous ciphertext block
- OFB (Output Feedback Mode) – creates a stream cipher from block cipher, key stream is obtained by encrypting previous key block

And newly (as specified in FIPS SP 800-38A) also:

- CTR (Counter Mode) – for a given key different input blocks (called counters) are encrypted and produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext.

An initialization vector (IV) can be required (according to the used mode of operation). The cipher is allowed by using different IV to produce for the same input (plaintext) and secret key a different output without a complex process of re-keying. It can be viewed as randomization of the encryption process that is performed (in CBC mode) by XORing plaintext IV to the first input block or (in CFB and OFB mode) by XORing encrypted IV to the first input block.

Padding

Padding is used to adjust the length of the input data (typically to make it a multiply of the length of the block to be able to use a specific block algorithm). Stream ciphers usually do not need padding because the key stream can be used only partially to match the length of the input data. The exception is a padding of very short messages (consider, e.g., one bit), where it is necessary to obscure the length of message/communication to prevent attacker guess the value. Hash algorithms (as we discussed) use their own input data padding. Finally, block symmetric ciphers use the following methods:

- **ISO 9797 method 1** – the message is padded with values 0x00 to the multiply of the block length.
    - to remove the padding correctly, it is necessary to know the exact length of the original message
- **ISO 9797 method 2** (ISO 7816-4, EMV'96) – first the value 0x80 is added, then ((n-||*M*|| mod n) - 1) bytes of 0x00 are added
    - *e.g. PS = '80 00', if ||M||mod n = 2;*
    - *e.g. PS = '80 00 00 00 00 00 00 00', if ||M|| mod 8 = 0;*
- **PKCS#5** – the padding string is made from value n-(||*M*|| mod n)
    - *for DES n=8, AES n=16*
    - *e.g. PS = 02 02 - if ||M||mod n = 6 ;*
    - *e.g. PS = 0n 0n 0n 0n 0n 0n 0n 0n - if ||M||mod n = 0. (the message is extended by one block)*

Note: The indication of decryption status based on the verification of the MAC or the detection of wrong padding can be used for attacks which can lead up to recovering the plaintext (e.g. CBC padding attack by S. Vaudenay).

The most important symmetric encryption algorithms are:

- DES – defined in FIPS PUB 46 (-1 a -2), key 56 bits, block 64 bits
- 3DES – defined in FIPS PUB 46-3, key either 112 or 168 bits, block 64 bits
- AES – (Rijndael), defined v FIPS PUB 197, key 128, 192 or 256 bits, block 128 bits
- IDEA – block 64 bits, key 128 bits
- BLOWFISH – block 64 bits, key 32 up to 448 bits

Asymmetric encryption algorithms

Asymmetric encryption algorithms use private and public keys to digitally sign data or to encrypt data to achieve data confidentiality. Due to the importance of the integrity of the public keys we have to care also about the format of the public key certificates. Many standards in the area of asymmetric cryptography were prepared by the company RSA Security. The standards are called PKCS and some of them became RFCs or norms.

PKCS#1 – defines RSA encryption
PKCS#3 – defines Diffie-Hellman protocol
PKCS#5 – symmetric encryption based on a password

PKCS#7 – format for digital signatures and encryption (including certificate storage). So called *hybrid encryption* is used: random symmetric key is generated, is used to encrypt the message and the key itself is asymmetrically encrypted with the public key of the recipient.
PKCS#8 – defines the private key format
PKCS#10 – defines format for certificate requests
PKCS#11 – API for communication with cryptographic tokens
PKCS#12 – format for storing private keys including public key certificates, all protected by a password
PKCS#13 – defines encryption based on elliptic curves
PKCS#15 – defines cryptographic token information format

To store certificates (owner identification, public key including description of used cryptographic functions and the purpose of this key) the standard ITU-T X.509 is used. (Today it is also ISO/IEC 9594-8, full text is available only for paying customers). The version 1 of the standard included only the basic fields; soon there appeared efforts to extend the format. PKCS#6 was one of such efforts. Later version v3 of the X.509 was published and it includes quite general way to add extended attributes (it is possible to add an arbitrary attribute, for compatibility with application that do not recognize such an attribute it must be labeled as uncritical (can be ignored) or critical (such certificate cannot be processed in an application that does not recognize this attribute). Certificates and certificate revocation lists are coded by using ASN.1 and can be stored in the binary form as so called DER certificates or can be further base64 encoded and with appended headers stores as PEM certificates (PEM is short for Privacy-Enhanced Mail, a predecessor of S/MIME). Conversion between the formats is trivial and can be done by a number of programs (e.g. OpenSSL).

Example of the certificate headers:
```
-----BEGIN CERTIFICATE----
-----END CERTIFICATE----
```
And for CRLs:
```
-----BEGIN X509 CRL-----
-----END X509 CRL-----
```

The PKCS#7 standard describes the structure of the cryptographically processed data (digitally signed, asymmetrically encrypted (hybrid encryption, digital envelope), other encrypted data or hashed data), it also supports certificate storage (in X.509 format). The PKCS#7 data structure can be hierarchic and so it is possible to sign a message already signed by another subject etc.
Standard S/MIME is intended to secure electronic mail, it uses PKCS#7 data format, the message is base64 encoded and proper MIME-types (multipart/signed, application/pkcs7-mime) are appended. S/MIME supports digital signatures of data (transparent and nontransparent) and data encryption. Emails use the MIME structure which itself can be hierarchic and makes it possible to cryptographically secure only part of the document or secure different parts in a different way (which can be a source of problems). S/MIME in the version 2 only supported 40-bit encryption and included the following parts:

- S/MIME Version 2 Message Specification (RFC 2311)

- S/MIME Version 2 Certificate Handling (RFC 2312)
- PKCS #1: RSA Encryption Version 1.5 (RFC 2313)
- PKCS #10: Certification Request Syntax Version 1.5 (RFC 2314)
- PKCS #7: Cryptographic Message Syntax Version 1.5 (RFC 2315)
- Description of the RC2 Encryption Algorithm (RFC 2268)

Version 3 is not restricted in the encryption strength and covers the following parts:

- Cryptographic Message Syntax (RFC 3852)
- Cryptographic Message Syntax (CMS) Algorithms (RFC 3370)
- S/MIME Version 3.1 Message Specification (RFC 3851)
- S/MIME Version 3.1 Certificate Handling (RFC 3850)
- Diffie-Hellman Key Agreement Method (RFC 2631)

The message format is now called CMS (Cryptographic Message Syntax).
Important asymmetric encryption algorithms are:

- RSA – defined in PKCS#1
- DSS – defined by NIST in FIPS PUB 186-2

Padding in RSA:

- **ANSIX 9.31**
  - 6b bb … bb ba || Hash(M) || 3x cc (where x=3 for sha1, x=1 for ripemd160)
- **PKCS#1 v1.5**
  - 00 01 ff … ff 00 || HashAlgID || Hash(M)
- **PSS**
  - 00 || H || G(H) $\oplus$ [salt || 00 … 00] (where H = Hash(salt, M), salt is random, and G is a mask generation function)

Random number generation

Cryptographic random number generators are typically used to generate (secret) random data (e.g., cryptographic keys, initialization vectors, or random challenges) whose quality and unpredictability is critical for many cryptographic operations. In general, two kinds of generators can be distinguished – the true random number generator and the pseudorandom number generator. The former is typically based on nondeterministic physical process/phenomena (e.g., radioactive decay or thermal noise), while the latter is only a deterministic algorithm where all randomness of the output is fully dependent on the randomness of the input (often called seed). Getting truly random data in the deterministic environment of computer systems is extremely hard and slow, therefore we often restrict ourselves to the use of deterministically generated pseudorandom data. True random data then serves only as input to the faster pseudorandom number generator.
To use the physical processes to get random numbers special HW is required. Such a HW is not cheap, but the output generated is good quality (even so it is processed in SW to remove some regularity). Without special HW it is very difficult for deterministic computers to generate random numbers. We can use nondeterministic users and their input by keyboard or mouse. Without the help of users we have to rely on timing of some events like hard disk interrupt or packet arrival. Linux provides special device /dev/random for random data based on timing of events in the system (obtaining larger amounts of such data can be time consuming (if not enough data is

available the reading operation is blocked)) and the device /dev/urandom for pseudorandom data (obtained by hashing).
Important pseudorandom number generators are:

- ANSI X9.17 / ANSI X9.31 PRNG
- DSS / FIPS 186 PRNG
- Yarrow, Tiny, and Fortuna PRNG

Useful links

RFC documents:
http://www.ietf.org/rfc.html
NIST FIPS standards:
http://csrc.nist.gov/publications/fips/index.html
PKCS standards:
http://www.rsasecurity.com/rsalabs/node.asp?id=2124

Assignments

1. Write a program (in any programming language) that will prepare a padded block for RSA signature with ANSI X9.31 and PKCS#1 v1.5 padding. Input is a file, output are 2 padded octet strings. Use sha-1 as the hash function. [5 points]

2. Extract RSA public exponents from sample certificates (use any tool, no need to create you own program). Just list the files and relevant public exponents.
[5 points]