Cryptographic Security

Cryptographic methods

- Private-key encryption
- One-way encryption
- Public-key encryption

Private-key encryption

"Standard" encryption. A key is used to encrypt the message, and the same key is used to decrypt it.

```
                key                                    key
                 |                                      |
                 v          Ciphertext                  v
Message  ---> encrypt ---------------------------> decrypt -->  Message
```

The ciphertext is only readable by someone who has a copy of the key.

Examples:

- DES (Data Encryption Standard) - 56 bit key. Free.
- IDEA - 128 bit key. Patented but free for non-commercial use.

Problem: how do you distribute keys securely? If users choose their own keys, they may be easy to guess.

One-way encryption ("secure hashing")

- hash function takes an input string (variable length) and converts it into an output (fixed length)
- Not possible to reverse the process - except by trying every possible input.

Examples:

- Unix 'crypt' (based on DES)
- MD4, MD5 (Message Digest)
- SHA (Secure Hash Algorithm)

```
    md5("My name is Brian\n") = 0134b6e9397f36c7c9f78fe17d2d7d8e
    md5("Test123\n")          = 36a03a8a4c00e81f03d62d8b04bbbf4d
```

Applications:

- Storing passwords

  Unix password database contains `hash(password)`. When the user types a password, the hash function is applied and compared to the stored hashed password.

In theory, password cannot be recovered from the hashed version. In practice, users tend to choose ’bad’ passwords (easily guessed).

Some possible solutions:

- Machine-generated passwords (but users write these down)
- Password complexity tests
- Periodically scan password file with ’crack’
- Keep hashed passwords away from prying eyes - "shadow passwords"

- One-time authentication

Can check whether other end knows a password ("shared secret") without sending that secret over the network, and without allowing a sniffer to reuse the result.

How it works: server generates a unique ’challenge’ (e.g. date and time). Client concatenates the challenge and secret, performs hash function, and returns the result. Server compares the answer with the expected value.

```
                                  Challenge
[Secret]   Generate -----------------------------------> -+       [Secret]
   |       Challenge                                      |          |
   |          |                                           |          |
   +-- hash --+                                           +-- hash --+
        |                                                      |
        v                   hash(Challenge+Secret)             |
     Compare <----------------------------------------- <----+
```

Examples: APOP (for POP3 - see RFC1939), CHAP (for PPP)

Disadvantage: Secret is stored in plain text on server

- S/Key authentication

Clever way of creating a sequence of one-time passwords *without* keeping a shared secret on the server.

Generate a sequence of passwords like this:

```
Seed:             hash(hash(hash(hash(hash(secret)))))
First password:   hash(hash(hash(hash(secret))))
Second password:  hash(hash(hash(secret)))
Third password:   hash(hash(secret))
Fourth password:  hash(secret)
```

The seed is stored on the server. When a user enters a password, `hash(password)` is compared with the seed. If it is correct, the user is permitted access, and the stored seed is replaced with the password typed by the user.

The nth password can be generated when required using a program, or a sequence of passwords can be printed out in advance.

Problem: when all the passwords are used up, the seed needs to be manually reset using a new secret value.

- Generating encryption keys from pass phrases

  Private-key encryption systems usually require a fixed-length key. A hash function can be used to convert an easily-remembered "pass phrase" into a suitable key. For maximum security, the pass phrase should be long (10 characters absolute minimum)

Public key encryption

A two-way encryption system which uses one key to encrypt and a different one to decrypt ("public" key and "private" key). The keys are related, but designed so that the private key cannot be derived from the public key.

Use for message encryption:

```
               public                                 private
                key                                     key
                 |                                      |
                 v           Ciphertext                 v
Message  ---> encrypt ---------------------------> decrypt -->  Message
```

Key distribution is very simple - the public key can be sent by any insecure means. Only the person with the private key can decipher the message.

Use for authentication - here the roles of the private and public keys are reversed:

```
              private                                  public
                key                                     key
                 |                                      |
                 v           Ciphertext                 v
Message  ---> encrypt ---------------------------> decrypt -->  Message
```

If you are able to decipher the text with the public key, this proves that it was generated by the person who has the private key.

Problems:

- Computation is slow. Hence we limit its use to small messages:
  - Securely transmit a key for use with private encryption (called a "session key")
  - For authentication, encode only a hash of the message; this hash can be recalculated at the far end and compared with the decrypted hash. The encrypted hash is known as a "digital signature"
- Subject to RSA patent in the United States [only]

Last updated 8 October 1996