Cryptographic methods:

PacNOG I Workshop

Presented by
Hervey Allen

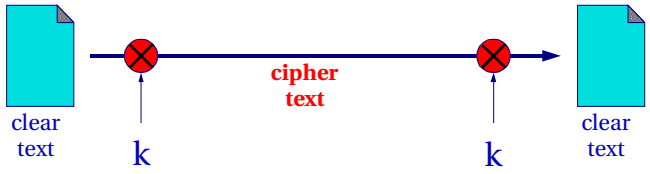
Materials originally by
Brian Candler

Recommended reading:
Applied Cryptography, Bruce Schneier

Why use cryptography?

- Can offer *genuinely secure* solutions to important security problems
- Some governments forbid it
- Confidentiality
  - Can I be sure no-one else can see my data? (e.g. sniffing)
  - Integrity
- Has my data been modified?
  - Authentication

1. "Private key" or "symmetric" ciphers



The same key is used to encrypt the document before sending and decrypt it at the far end

We assume an eavesdropper is able to intercept the ciphertext

- How can they recover the cleartext?

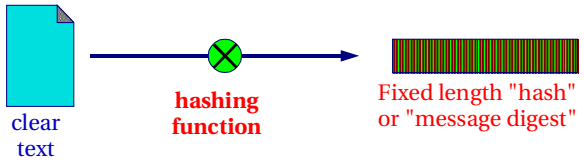
Examples of symmetric ciphers

- DES - 56 bit key length, designed by US security service
- 3DES - effective key length 112 bits
- AES (Advanced Encryption Standard) - 128 to 256 bit key length
- Blowfish - 128 bits, optimised for fast operation on 32-bit microprocessors
- IDEA - 128 bits, patented (requires a licence for commercial use)

Features of symmetric ciphers

- Fast to encrypt and decrypt, suitable for large volumes of data
- A well-designed cipher is only subject to brute-force attack; the strength is therefore directly related to the key length
- Current recommendation is a key length of at least 90 bits
  - i.e. to be fairly sure that your data will be safe for at least 20 years
  - http://www.schneier.com/paper-keylength.html
- Problem - how do you distribute the keys?

2. "Hashing" - one-way encryption



Munging the document gives a short "message digest" (checksum). Not possible to go back from the digest to the original document.

Examples

- Unix crypt() function, based on DES
- MD5 (Message Digest 5) - 128 bit hash
- SHA1 (Secure Hash Algorithm) - 160 bits
- Collisions have been found for SHA1/MD5. Some discussions here:
  http://en.wikipedia.org/wiki/SHA-1
  http://www.cits.rub.de/MD5Collisions/
  http://www.schneier.com/blog/archives/2005/03/more_hash_funct.html
- *Almost n*o two documents have been discovered which have the same MD5 digest
- So far, no feasible method to create any document which has a given MD5 digest

So what use is that? a. Integrity checks

- You can run many megabytes of data through MD5 and still get only 128 bits to check
- It is difficult for an attacker to modify your file and leave it with the same MD5 checksum
- Gives your document an almost unique "fingerprint"

Exercise

- Exercise: on your machine type
  - cat /etc/motd
- Look at your neighbour's machine. Is their file *exactly* the same as yours? Can you be sure?
- md5sum /etc/motd
- Compare the result with your neighbour
- Now change ONE character in /etc/motd and repeat the md5sum test

Software announcements often contain an MD5 checksum

- It's trivial to check
- Protects you against hacked FTP servers and download errors

```
$ md5 exim-4.50_1.tbz
MD5 (exim-4.50_1.tbz) = 1884ca8e48536a087b86c279de5c9e69
$
```

Two Considerations:
1. Could the attacker have modified the original email announcment?
2. You need to keep the md5sum file *on a separate server* from the software being downloaded.

So what use is that? b. Encrypted password storage

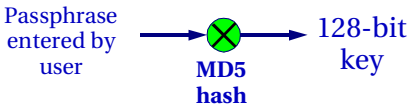
- We don't want to keep cleartext passwords if possible; the password file would be far too attractive a target
- Store hash(passwd) in /etc/master.passwd (shadow password in Linux)
- When user logs in, calculate the hash of the password they have given, and compare it to the hash in the password file
- If the two hashes match, the user must have entered the correct password
- *Can an attacker still recover the password?*

So what use is that?
c. Generating encryption keys

- Users cannot remember 128 bit binary encryption keys
- However they can remember "passphrases"
- A hash can be used to convert a passphrase into a fixed-length encryption key
- The longer the passphrase, the more "randomness" it contains and the harder to guess. English text is typically only 1.3 bits of randomness per character.

http://www.cranfield.ac.uk/docs/email/pgp/pgp-attack-faq.txt
http://www.schneier.com/paper-personal-entropy.html

Generating encryption keys for symmetric ciphers



Every passphrase generates a different 128-bit key

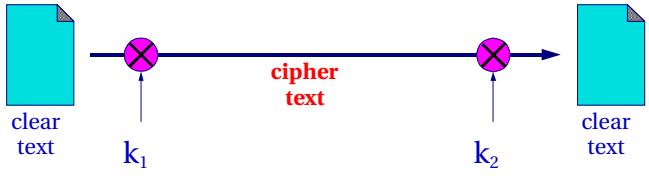
Example: GPG with symmetric cipher

```
# vi foobar.txt
# gpg -c foobar.txt
Enter passphrase: ding/dong 479 fruitbat
Repeat passphrase: ding/dong 479 fruitbat
# ls foobar.txt*
foobar.txt  foobar.txt.gpg
# rm foobar.txt
rm: remove regular file `foobar.txt'? y

# gpg foobar.txt.gpg
gpg: CAST5 encrypted data
Enter passphrase: ding/dong 479 fruitbat
# cat foobar.txt
```

("gpg --version" shows the ciphers available)

3. "Public key" ciphers

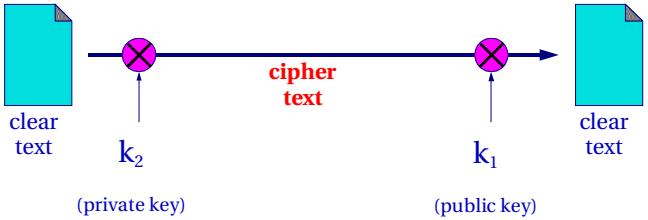


One key is used to encrypt the document, a different key is used to decrypt it

Public key and Private key

- The Public key and Private key are mathematically related (generated as a pair)
- It is easy to convert the Private key into the Public key. It is not easy to do the reverse.
- Key distribution problem is solved: you can post your public key anywhere. People can use it to encrypt messages to you, but only the holder of the private key can decrypt them.
- Examples: RSA, Elgamal (DSA)

Use for authentication: reverse the roles of the keys



If you can decrypt the document with the public key, it proves it was written by the owner of the private key (and was not changed)

Key lengths

- Attacks on public key systems involve mathematical attempts to convert the public key into the private key. This is more efficient than brute force.
- 512-bit has been broken
- Recent developments suggest that 1024-bit keys might not be secure for long
- Recommend using 2048-bit keys*

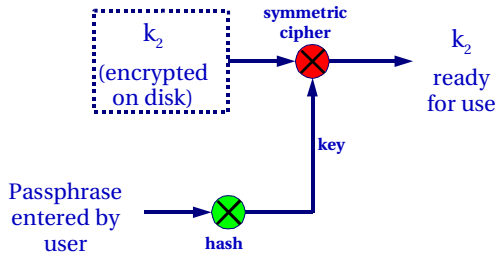
*http://www.rsasecurity.com/rsalabs/node.asp?id=2218

Protecting the private key*

- The security of the private key is paramount: keep it safe!
- Keep it on a floppy or a smartcard?
- Prefer to keep it *encrypted* if on a hard drive
- That means you have to decrypt it (using a passphrase) each time you use it
- An attacker would need to steal the file containing the private key, AND know or guess the passphrase.

*Some disagree with this notion...

Protecting the private key

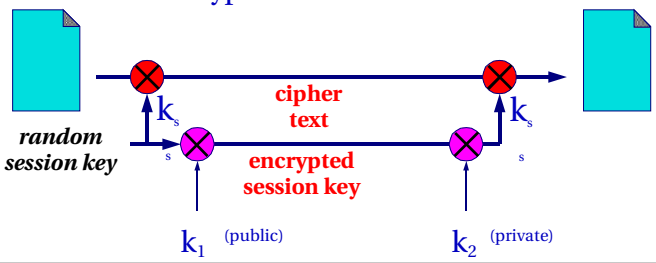


Public key cryptosystems are important

- But they require a lot of computation (expensive in CPU time)
- So we use some tricks to minimise the amount of data which is encrypted

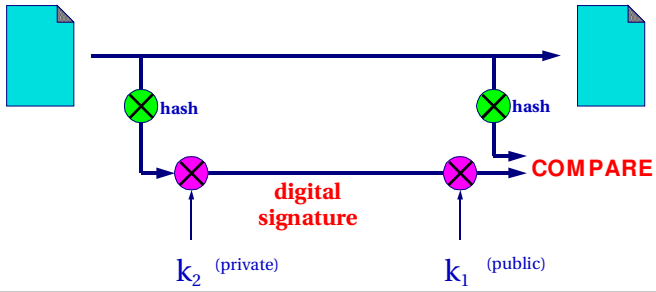
When encrypting:

- Use a symmetric cipher with a random key (the "session key"). Use a public key cipher to encrypt the session key and send it along with the encrypted document.



When authenticating:

- Take a hash of the document and encrypt only that. An encrypted hash is called a "digital signature"

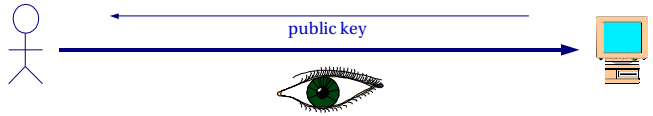


Digital Signatures have many uses, for example:

- E-commerce. An instruction to your bank to transfer money can be authenticated with a digital signature.
  - Legislative regimes are slow to catch up
- A trusted third party can issue declarations such as "the holder of this key is a person who is legally known as Alice Hacker"
  - like a passport binds your identity to your face
- Such a declaration is called a "certificate"
- You only need the third-party's public key to check the signature

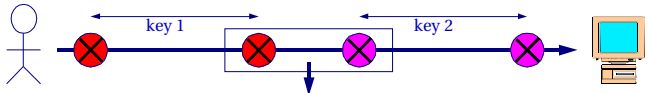
Do public keys *really* solve the key distribution problem?

- Often we want to communicate securely with a remote party whose key we don't know
- We can retrieve their public key over the network
- But what if there's someone in between intercepting our traffic?



The "man in the middle" attack

- Passive sniffing is no problem
- But if they can modify packets, they can substitute a different key
- The attacker uses separate encryption keys to talk to both sides
- You think your traffic is secure, but it isn't!



Attacker sees all traffic in plain text
- and can modify it!

Digital Certificates can solve the man-in-the-middle problem

- Problem: I have no prior knowledge of the remote side's key
- But someone I trust can check who they are
- The trusted third party can vouch for the remote side by signing a certificate which contains the remote side's name and public key
- I can check the validity of the certificate using the trusted third party's public key

Example: TLS (SSL) web server with digital certificate

- I generate a private key on my webserver
- I send my public key plus my identity (my webserver's domain name) to a certificate authority (CA)
- The CA manually checks that I am who I say I am, i.e. I own the domain
- They sign a certificate containing my public key, my domain name, and an expiration date (Q: why is an expiration date included?)
- I install the certificate on my web server

When a client's web browser connects to me with HTTPS:

- They negotiate an encrypted session with me, during which they learn my public key
- I send them the certificate
- They verify the certificate using the CA's public key, which is built-in to the browser
- If the signature is valid, the domain name in the URL matches the domain name in the certificate, and the expiration date has not passed, they know the connection is secure

The security of TLS depends on:

- Your webserver being secure
  - So nobody else can obtain your private key
- The CA's public key being in all browsers
- The CA being well managed
  - How carefully do they look after their own private keys?
- The CA being trustworthy
  - Do they vet all certificate requests properly?
  - Could a hacker persuade the CA to sign their key pretending to be someone else? What about a government?

PGP takes a different view

- We don't trust anyone except our friends (especially not big corporate monopolies)
- You sign your friends' keys to vouch for them
- Other people can choose to trust your signature as much as they trust you
- Generates a distributed "web of trust"
- Sign someone's key when you meet them face to face - "PGP key signing parties"

SSH uses a simple solution to man-in-the-middle

- The first time you connect to a remote host, remember its public key
  - Stored in ~/.ssh/known_hosts
- The next time you connect, if the remote key is different, then maybe an attacker is intercepting the connection!
  - Or maybe the remote host has just got a new key, e.g. after a reinstall. But it's up to you to resolve the problem
- Relies on there being no attack in progress the *first* time you connect to a machine

SSH can eliminate passwords

- Use public-key cryptography to prove who you are
- Generate a public/private key pair locally
  - ssh-keygen -t dsa
  - Private key is ~/.ssh/id_dsa
  - Public key is ~/.ssh/id_dsa.pub
- Install your PUBLIC key on remote hosts
  - mkdir .ssh
  - chmod 755 .ssh
  - Copy public key into ~/.ssh/authorized_keys
- Login!

Notes on SSH authentication

- Private key is protected by a passphrase
  - So you have to give it each time you log in
  - Or use "ssh-agent" which holds a copy of your passphrase in RAM
- No need to change passwords across dozens of machines
- Disable passwords entirely!
  - /etc/ssh/sshd_config
- Annoyingly, for historical reasons there are *three* different types of SSH keys
  - SSH1 RSA*, SSH2 DSA, SSH2 RSA (*largely gone)

Designing a good cryptosystem is very difficult

- Many possible weaknesses and types of attack, often not obvious
- DON'T design your own!
- DO use expertly-designed cryptosystems which have been subject to widespread scrutiny
- Understand how they work and where the potential weaknesses are
- Remember the other weaknesses in your systems, especially the human ones

Where can you apply these cryptographic methods?

- At the link layer
  - PPP encryption
- At the network layer
  - IPSEC
- At the transport layer
  - TLS (SSL): many applications support it
- At the application layer
  - SSH: system administration, file transfers
  - PGP/GPG: for securing E-mail messages, stand-alone documents, software packages etc.
  - Tripwire (and others): system integrity checks

Some Resources

Some interesting web links for further reading:

- Crypto FAQ from RSA Security
  `http://www.rsasecurity.com/rsalabs/node.asp?id=2152`
- Cryptography resources from Schneier.com
  `http://www.schneier.com/resources.html`
- Wikipedia SHA-1 collision discussion
  `http://en.wikipedia.org/wiki/SHA-1`