Cryptography in the Real World

Diffie-Hellman Key Exchange

RSA Analysis

RSA Performance

SSH Protocol

Diffie-Hellman Key Exchange

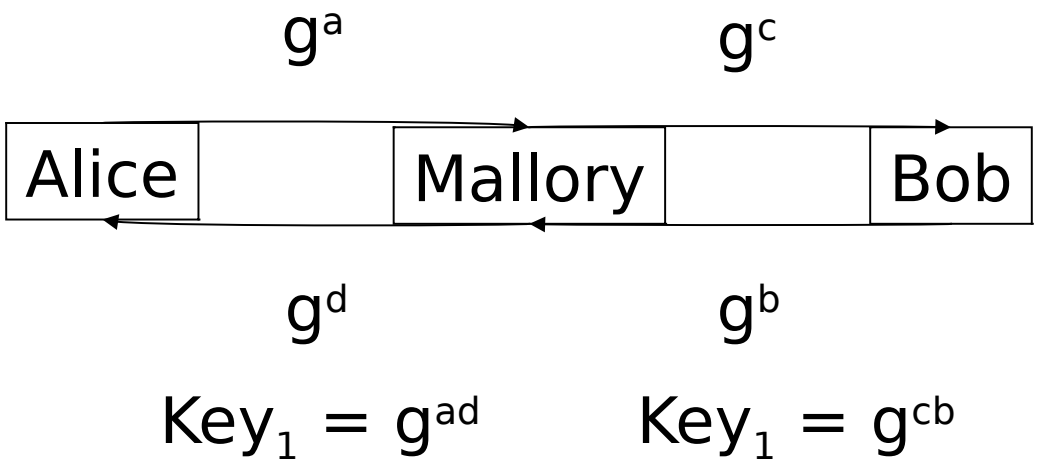
A multiplicative group, e.g., Z_P-{0}, and a generator g is made public.

- Alice picks a, and sends g^a to Bob in the clear
- Bob picks b and sends g^b to Alice in the clear
- The shared key is g^{ab}

Why is this secure? Because discrete logs seem to be hard to compute: i.e., given g^a, finding a is difficult (and similarly for g^b and b).

Note that $g^a g^b = g^{a+b}$, not g^{ab}.

Person-in-the-middle attack



Mallory gets to listen to everything.

RSA

Invented by Rivest, Shamir and Adleman in 1978

Based on **difficulty of factoring**.

Used to **hide** the size of a group Z_n^* since:

$$\left|Z_n^*\right| = \phi(n) = n\prod_{p|n}(1-1/p)$$

Factoring has not been reduced to RSA

- an algorithm that generates m from c does not give an efficient algorithm for factoring

On the other hand, factoring has been reduced to finding the private-key.

- there is an efficient algorithm for factoring given one that can find the private key.

RSA Public-key Cryptosystem

What we need:

- p and q, primes of approximately the same size
- $n = pq$
  $\phi(n) = (p-1)(q-1)$
- $e \in Z_{\phi(n)}$
- d = inv. of e in $Z_{\phi(n)}$
  i.e., $d = e^{-1} \bmod \phi(n)$

Public Key: (e,n)
Private Key: d

Encode:
$m \in Z_n$
$E(m) = m^e \bmod n$

Decode:
$D(c) = c^d \bmod n$

RSA continued

Why it works:

$D(c) = c^d \bmod n$

$= m^{ed} \bmod n$

$= m^{1 + k(p-1)(q-1)} \bmod n$

$= m^{1 + k\,\phi(n)} \bmod n$

$= m(m^{\phi(n)})^k \bmod n$

$= m$ (by Euler's Theorem, $m^{k\phi(n)} \bmod n = m^0 \bmod n$, if m and n are relatively prime.)

Note that in general $m^a \neq m^{a \bmod n} \bmod n$, but by Euler's Theorem $m^a = m^{a \bmod \phi(n)} \bmod n$, where $\phi(n) = |Z_n^*|$, and $Z_n^* = \{s \in Z_n$ such that s and n are relatively prime$\}$, and $m \in Z_n^*$.

What if m and n share a factor?

Euler's theorem doesn't guarantee that $m^{k\phi(n)} = 1 \mod n$

Answer 1: Special case, still works. By the Chinese Remainder Theorem, if $m^{ed}=m \mod p$ and $m^{ed}=m \mod q$, then $m^{ed}=m \mod pq$, where p and q are relatively prime.

If $m = 0 \mod p$, then $m^{ed} = 0^{ed} = 0 = m \mod p$. Otherwise

$$m^{ed} \equiv m^{(ed-1)}m = m^{h(p-1)(q-1)}m = \left(m^{p-1}\right)^{h(q-1)} m \equiv 1^{h(q-1)}m \equiv m \pmod p$$

where by Fermat's Little Theorem $m^{p-1} = 1 \mod p$

Answer 2: jackpot! You can factor n using Euclid's alg.

RSA computations

To **generate the keys**, we need to

- **Find two primes p and q**. Generate candidates and use primality testing to filter them.
- **Find e^{-1} mod (p-1)(q-1).** Use Euclid's algorithm. Takes time $\log^2(n)$

To **encode and decode**

- **Take m^e or c^d**. Use the power method. Takes time $\log(e)\ \log^2(n)$ and $\log(d)\ \log^2(n)$.

In practice e is selected to be small so that encoding is fast.

Security of RSA

Warning:

- Do not use this or any other algorithm naively!

Possible security holes:

- Need to use “safe” primes p and q. In particular p-1 and q-1 should have large prime factors.
- p and q should not have the same number of digits. Can use a middle attack starting at sqrt(n).
- e cannot be too small
- Don’t use same n for different e’s.
- You should always “pad”

RSA Performance

Performance: (600Mhz PIII) (from: ssh toolkit):

Algorithm	**Bits/key**		**Mbits/sec**
RSA Keygen	1024	.35sec/key	
	2048	2.83sec/key	
RSA Encrypt	1024	1786/sec	3.5
	2048	672/sec	1.2
RSA Decrypt	1024	74/sec	.074
	2048	12/sec	.024
ElGamal Enc.	1024	31/sec	.031
ElGamal Dec.	1024	61/sec	.061
DES-cbc	56		95
twofish-cbc	128		140
Rijndael	128		180

RSA in the "Real World"

Part of many standards: PKCS, ITU X.509, ANSI X9.31, IEEE P1363

Used by: SSL, PEM, PGP, Entrust, …

The standards specify many details on the implementation, e.g.

- e should be selected to be small, but not too small
- "multi prime" versions make use of n = pqr… this makes it cheaper to decode especially in parallel (uses Chinese remainder theorem).

Factoring in the Real World

Quadratic Sieve (QS):

$$T(n) = e^{(1+o(n))(\ln n)^{1/2}(\ln(\ln n))^{1/2}}$$

- Used in 1994 to factor a 129 digit (428-bit) number. 1600 Machines, 8 months.

Number field Sieve (NFS):

$$T(n) = e^{(1.923+o(1))(\ln n)^{1/3}(\ln(\ln n))^{2/3}}$$

- Used in 1999 to factor 155 digit (512-bit) number. 35 CPU years. At least 4x faster than QS
- Used in 2003-2005 to factor 200 digits (663 bits) 75 CPU years ($20K prize)

SSH v2

- Server has a permanent “host” public-private key pair (RSA or DSA) . Public key typically NOT signed by a certificate authority. Client warns if public host key changes.
- Diffie-Hellman used to exchange session key.
  - Server selects g and p (group size) and sends to client.
  - Client and server create DH private keys a and b. Client sends public DH key g^a.
  - Server sends public DH key g^b and signs hash of DH shared secret g^{ab} and 12 other values with its private “host” key.
  - Client verifies signed shared secret using public key.
- Symmetric encryption using 3DES, Blowfish, AES, or Arcfour begins.
- User can authenticate by sending password or using public-private key pair. Private key has optional passphrase.
- If using keys, server sends “challenge” signed with users public key for user to decode with private key.

Why Combine RSA and Diffie-Hellman?

Why doesn't the client just send a symmetric key to the server, encrypted with the server's public key?

Because if the server's private key is later compromised, previous communications encrypted with the public key can be decrypted, revealing the symmetric key. Then all communications encrypted with the symmetric key can also be decrypted!

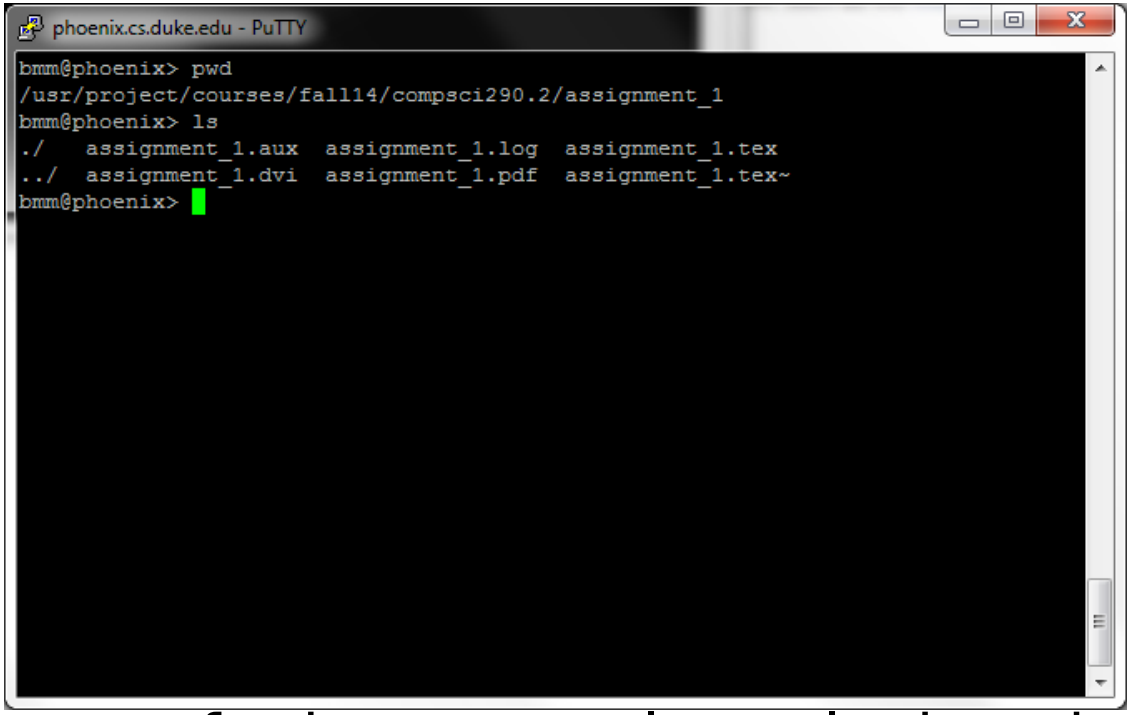
To prevent this attack, Diffie-Hellman ensures that the symmetric key is never transmitted, even in encrypted form, and the client and server discard the symmetric key after the session is over.

SSL/TLS provides this option too: DHE_RSA key exchange

"Perfect forward secrecy"

SSH Applications

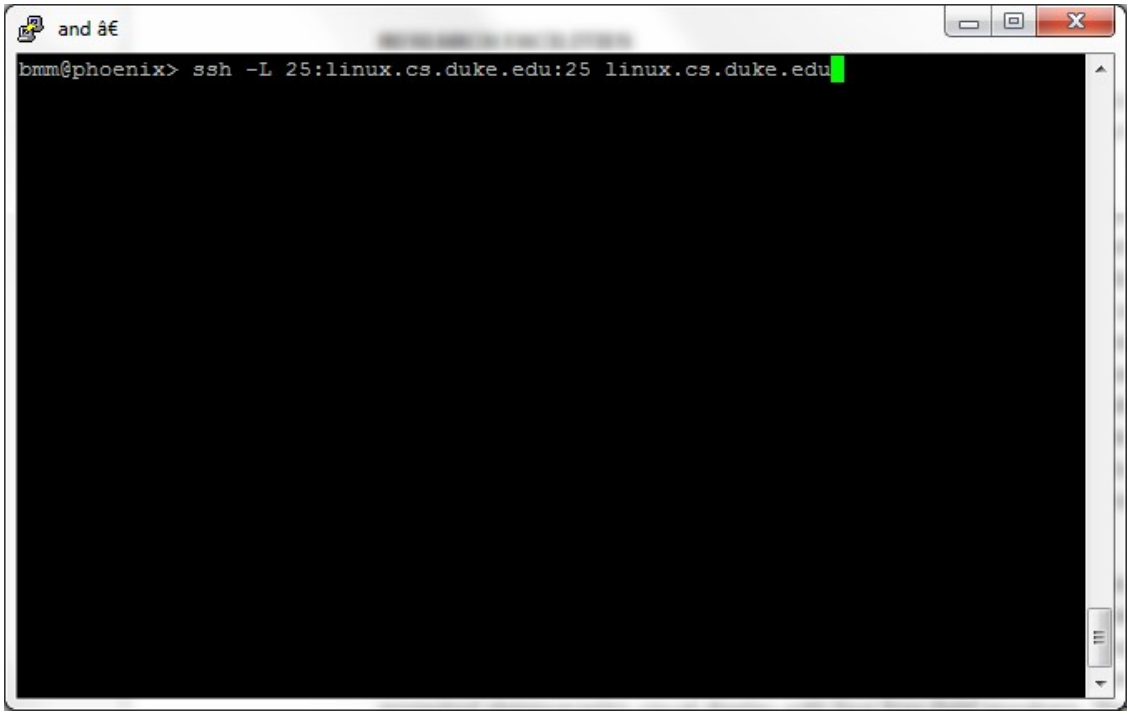
Secure Shell (SSH):



Replacement for insecure telnet, rlogin, rsh, rexec, which sent plaintext passwords over the network!

SSH Applications

Port forwarding (email example):



Log in to linux.cs.duke.edu. Forward anything received locally (phoenix) on port 25 to linux.cs.duke.edu on port25.

Useful if “phoenix” is not a trusted email relayer but “linux” is.

“phoenix” email program configured to use phoenix as relayer