Public Key

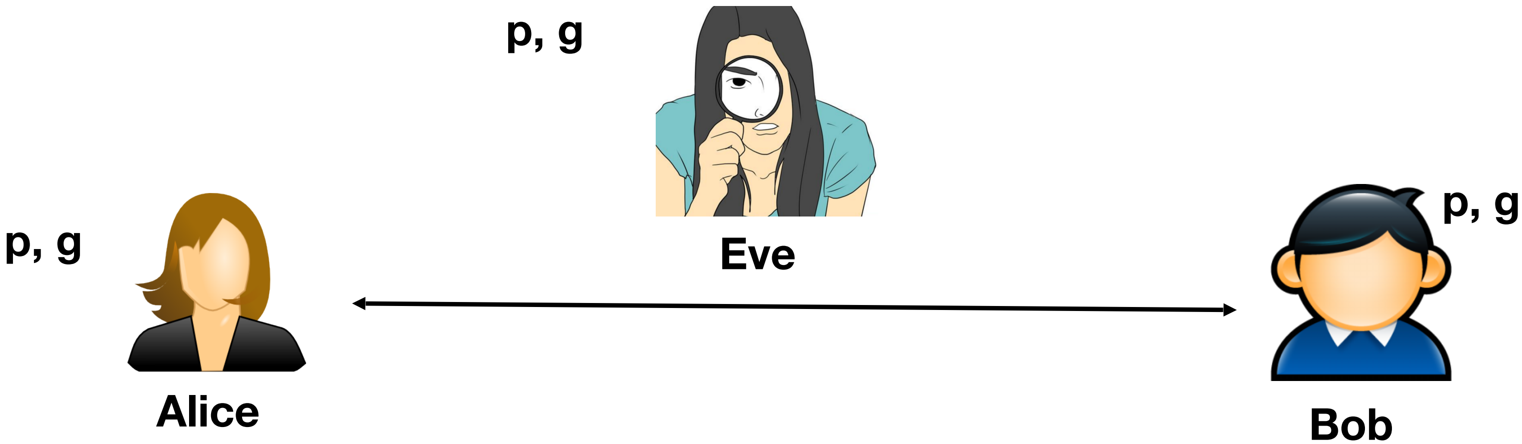
Our Roadmap...

- Public Key:
  - Something ***everyone*** can know
- Private Key:
  - The secret belonging to a specific person
- Diffie/Hellman:
  - Provides key exchange with no pre-shared secret
- ElGamal & RSA:
  - Provide a message to a recipient only knowing the recipient's ***public key***
- DSA & RSA signatures:
  - Provide a message that anyone can prove was generated with a ***private key***

Diffie-Hellman Key Exchange

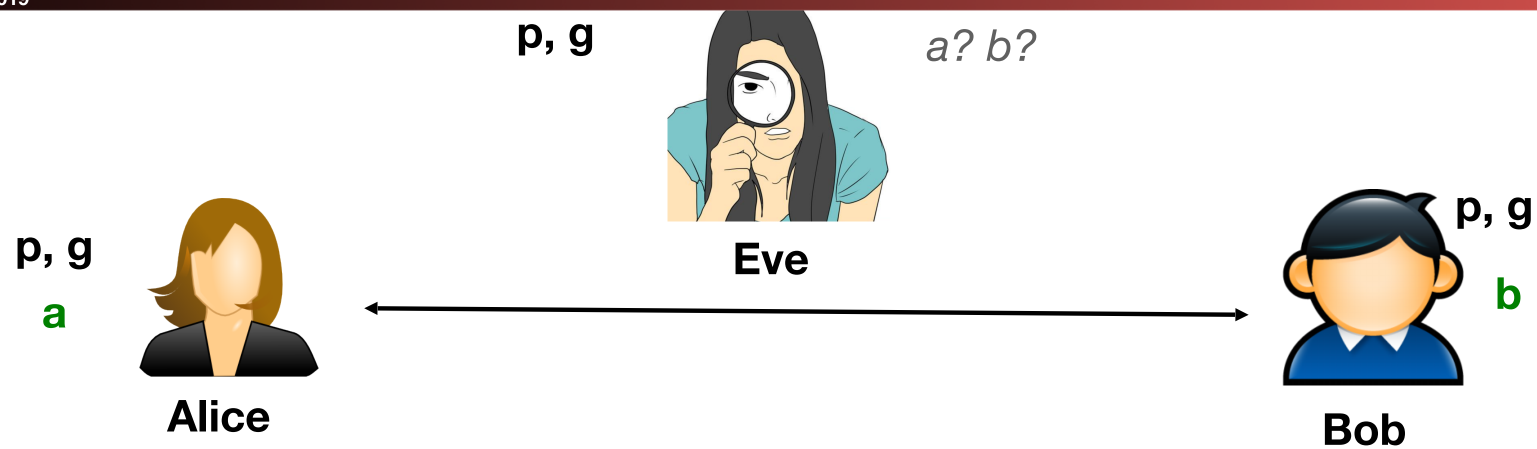
- What if instead they can somehow generate a random key when needed?
- Seems impossible in the presence of Eve observing all of their communication …
  - How can they exchange a key without her learning it?
- But: actually is possible using public-key technology
  - Requires that Alice & Bob know that their messages will reach one another without any meddling
- Protocol: Diffie-Hellman Key Exchange (DHE)
  - The E is "Ephemeral", we use this to create a temporary key for other uses and then forget about it

Diffie-Hellman Key Exchange



1. Everyone agrees in advance on a well-known (large) prime **p** and a corresponding **g**: 1 < g < p-1

DHE



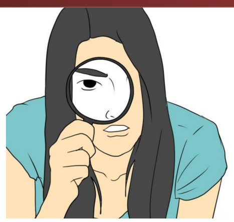
2. Alice picks random secret '**a**': 1 < a < p-1
3. Bob picks random secret '**b**': 1 < b < p-1

DHE

p, g

A

B



Eve

a? b?

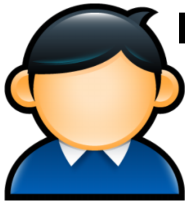
p, g

a

Alice

p, g

b



Bob

A = g^a mod p

4. Alice sends A = g^a mod p to Bob

5. Bob sends B = g^b mod p to Alice

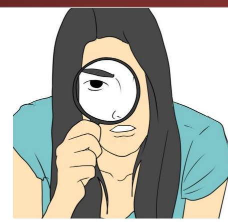
g^b mod p = **B**

DHE

p, g

A

B



Eve

a? b?

p, g

a

K

Alice

p, g

b

K

Bob

$A = g^a \bmod p$

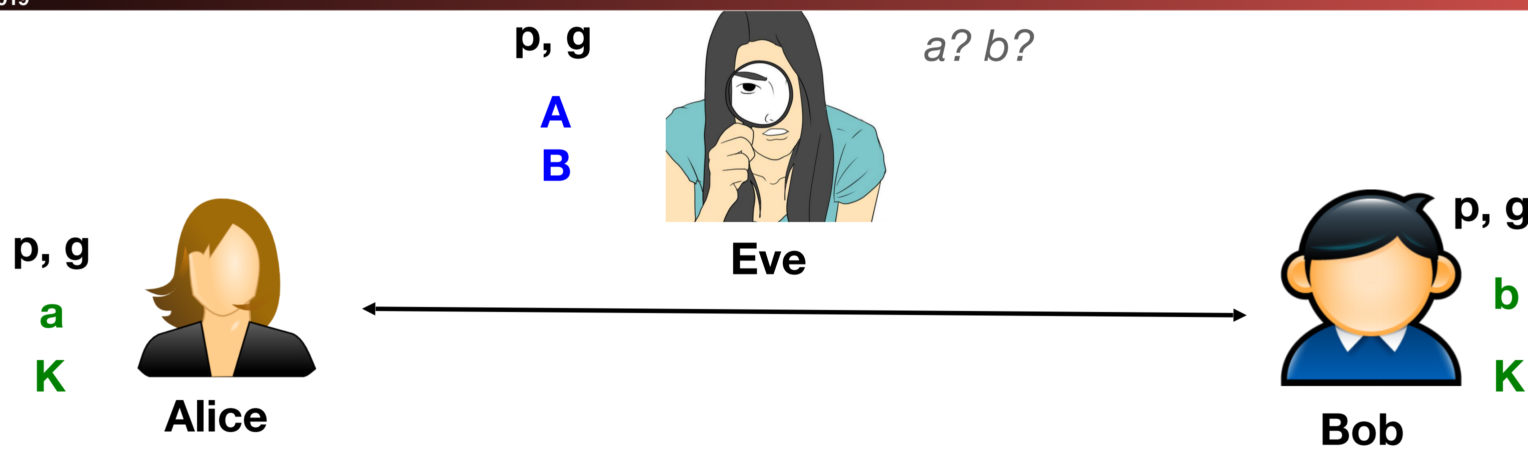
B

A

$g^b \bmod p = B$

6. Alice knows {a, A, B}, computes
   $K = B^a \bmod p = (g^b)^a = g^{ba} \bmod p$
7. Bob knows {b, A, B}, computes
   $K = A^b \bmod p = (g^a)^b = g^{ab} \bmod p$
8. K is now the shared secret key.

DHE



While Eve knows {p, g, g^a mod p, g^b mod p}, believed to be ***computationally infeasible*** for her to then deduce K = g^{ab} mod p.

She can easily construct A·B = $g^a \cdot g^b$ mod p = g^{a+b} mod p.
But computing g^{ab} requires ability to take *discrete logarithms* mod p.

This is Ephemeral Diffie/Hellman

- $\mathbf{K = g^{ab} \bmod p}$ is used as the basis for a "session key"
  - A symmetric key used to protect subsequent communication between Alice and Bob
    - In general, public key operations are vastly more expensive than symmetric key, so it is mostly used just to agree on secret keys, transmit secret keys, or sign hashes
  - If either **a** or **b** is random, **K** is random
- When Alice and Bob are done, they discard **K**, **a**, **b**
  - This provides ***forward secrecy***: Alice and Bob don't retain any information that a later attacker who can compromise Alice or Bob's secrets could use to decrypt the messages exchanged with **K**.

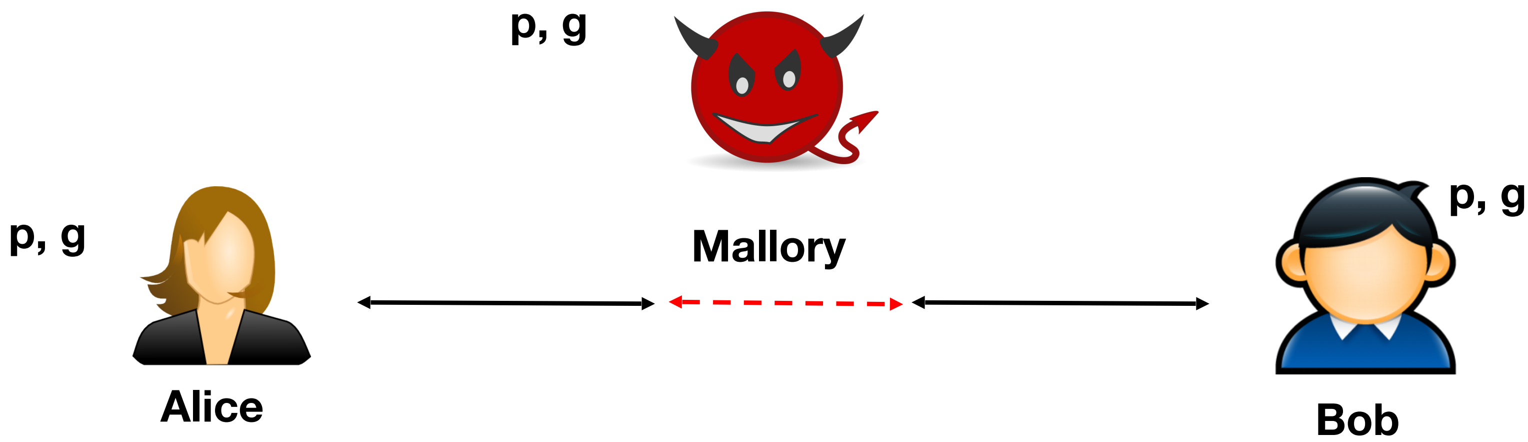
Diffie Hellman is part of more generic problem

- This involved deep mathematical voodoo called "Group Theory"
  - Its actually done under a group G
- Two main groups of note:
  - Numbers mod ***p*** with generator ***g***
  - Point addition in an elliptic curve C
    - Usually identified by number, eg. p256, p384 (NSA-developed curves) or Curve25519 (developed by Dan Bernstein, also 256b long)
- So EC (Elliptic Curve) == different group
  - Thought to be harder so fewer bits: 384b ECDHE ?= 3096b DHE
  - But otherwise, its "add EC to the name" for something built on discrete log

But Its Not That Simple

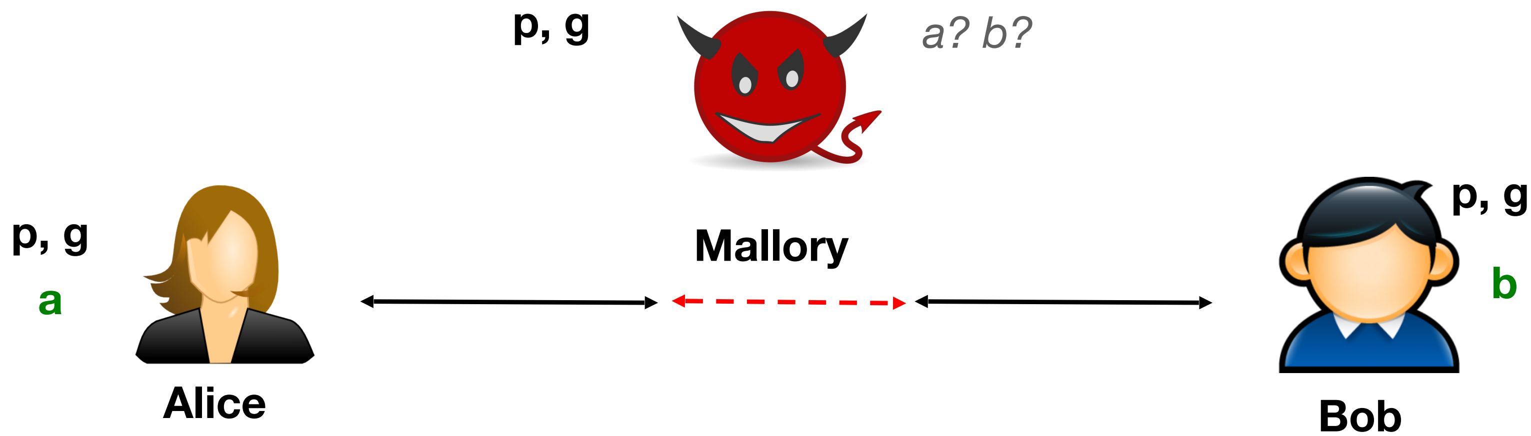
- What if Alice and Bob aren't facing a passive eavesdropper
  - But instead are facing Mallory, an ***active*** Man-in-the-Middle
- Mallory has the ability to change messages:
  - Can remove messages and add his own
- Lets see... Do you think DHE will still work as-is?

Attacking DHE as a MitM



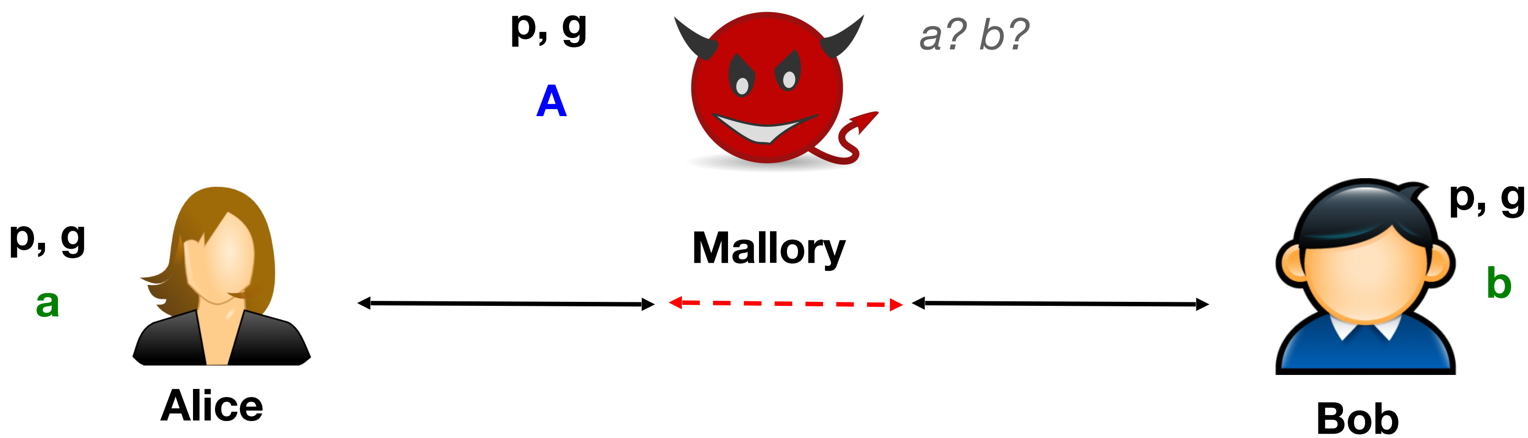
What happens if instead of Eve watching, Alice & Bob face the threat of a hidden Mallory (MITM)?

The MitM Key Exchange



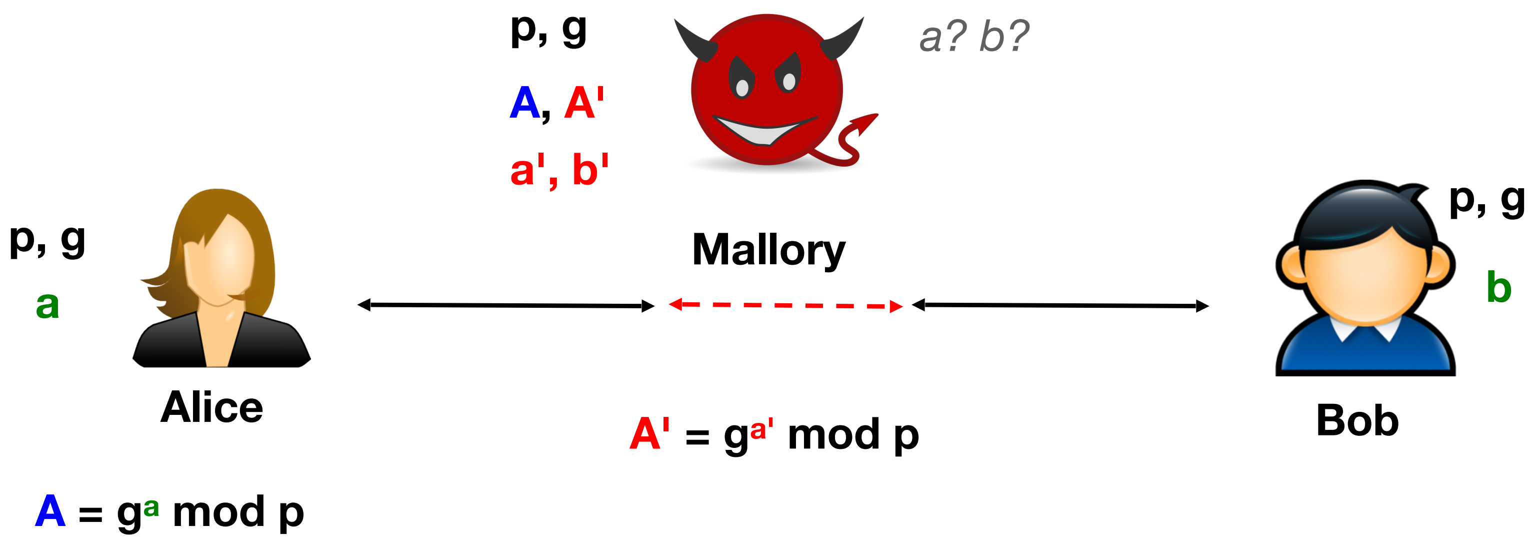
2. Alice picks random secret '**a**': 1 < a < p-1

3. Bob picks random secret '**b**': 1 < b < p-1



$A = g^a \bmod p$

4. Alice sends $A = g^a \bmod p$ to Bob
5. Mallory prevents Bob from receiving A



6. Mallory generates her own a', b'
7. Mallory sends $A' = g^{a'} \bmod p$ to Bob

p, g

$\color{blue}{A}$, $\color{red}{A'}$

$\color{red}{a', b'}$

a? b?

Mallory

p, g

$\color{green}{a}$

Alice

p, g

$\color{green}{b}$

Bob

$\color{red}{A'} = g^{\color{red}{a'}} \bmod p$

$\color{blue}{A} = g^{\color{green}{a}} \bmod p$

$\color{red}{A'}$

$g^{\color{green}{b}} \bmod p = \color{blue}{B}$

8. The same happens for Bob and B/B'

p, g

A, B, A', B'

a', b'

a? b?

Mallory

p, g

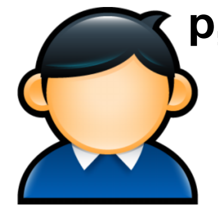
a

Alice

$A = g^{a} \bmod p$

$A' = g^{a'} \bmod p$

$B' = g^{b'} \bmod p$



p, g

b

Bob

A'

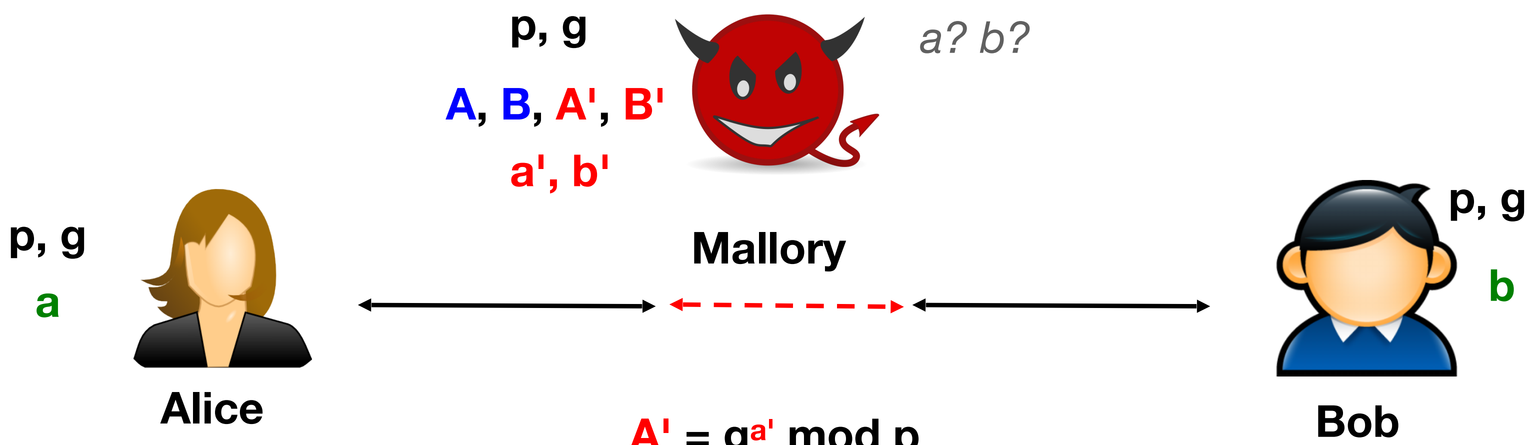
$g^{b} \bmod p = B$

8. The same happens for Bob and B/B'

9. Alice and Bob now compute keys they share with … Mallory!

10. Mallory can relay encrypted traffic between the two ...

 10'. Modifying it or making stuff up *however she wishes*



$A' = g^{a'} \bmod p$

$B' = g^{b'} \bmod p$

$A = g^{a} \bmod p$

B'

A'

$g^{b} \bmod p = B$

$K'_1 = A^{b'} \bmod p = g^{ab'} \bmod p$

$K'_2 = B^{a'} \bmod p = g^{ba'} \bmod p$

$\boxed{K'_1 = (B')^{a} \bmod p = (g^{b'})^{a} = g^{b'a} \bmod p}$

$\boxed{K'_2 = (A')^{b} \bmod p = (g^{a'})^{b} = g^{a'b} \bmod p}$

So We Will Want More...

- This is online:
  - Alice and Bob actually need to be active for this to work...
- So we want offline encryption:
  - Bob can send a message to Alice that Alice can read at a later date
- And authentication:
  - Alice can publish a message that Bob can verify was created by Alice later
  - Can also be used as a building-block for eliminating the MitM in the DHE key exchange:
    Alice authenticates **A**, Bob verifies that he receives **A** not **A'**.

Public Key Cryptography #1: RSA

- Alice generates two ***large*** primes, **p** and **q**
  - They should be generated randomly:
    Generate a large random number and then use a "primality test":
    A ***probabilistic*** algorithm that checks if the number is prime
- Alice then computes **n = p*q** and **ϕ(n) = (p-1)(q-1)**
  - **ϕ(n)** is Euler's totient function, in this case for a composite of two primes
- Chose random **2 < e < ϕ(n)**
  - **e** also needs to be relatively prime to **ϕ(n)** but it can be small
- Solve for $\mathbf{d = e^{-1} \bmod \phi(n)}$
  - You can't solve for **d** without knowing **ϕ(n)**, which requires knowing **p** and **q**
- **n**, **e** are public, **d**, **p**, **q**, and **ϕ(n)** are secret

RSA Encryption

- Bob can easily send a message m to Alice:
  - Bob computes **$c = m^e \bmod n$**
  - Without knowing **d**, it is believed to be intractable to compute **m** given **c**, **e**, and **n**
    - But if you can get **p** and **q**, you can get **d**:
      It is ***not known*** if there is a way to compute **d** without also being able to factor **n**, but it is known that if you can factor **n**, you can get **d**.
    - And factoring is ***believed*** to be hard to do
- Alice computes **$m = c^d \bmod n = m^{ed} \bmod n$**
- Time for some math magic...

RSA Encryption/Decryption, con't

- So we have: $\mathbf{D(C, K_D) = (M^{e\cdot d}) \bmod n}$
- Now recall that **d** is the multiplicative inverse of **e**, modulo **ϕ(n)**, and thus:

  $\mathbf{e\cdot d = 1 \bmod \phi(n)}$ (by definition)

  $\mathbf{e\cdot d - 1 = k\cdot\phi(n)}$ for some **k**

- Therefore $\mathbf{D(C, K_D) = M^{e\cdot d} \bmod n = (M^{e\cdot d-1})\cdot M \bmod n}$

  $\mathbf{= (M^{k\phi(n)})\cdot M \bmod n}$

  $\mathbf{= [(M^{\phi(n)})^k]\cdot M \bmod n}$

  $\mathbf{= (1^k)\cdot M \bmod n}$ *by Euler's Theorem: $a^{\phi(n)} \bmod n = 1$*

  $\mathbf{= M \bmod n = M}$

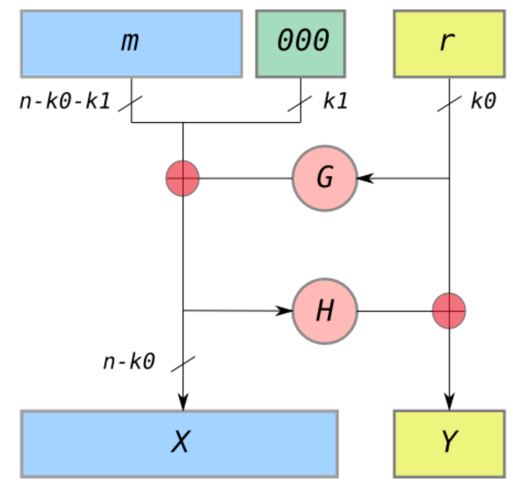
(believed) Eve can recover M from C *iff* Eve can factor n=p·q

But It Is Not That Simple...

- What if Bob wants to send the same message to Alice twice?
  - Sends $\mathbf{m^{e_a} \bmod n_a}$ and then $\mathbf{m^{e_a} \bmod n_a}$
  - Oops, not IND-CPA!
- What if Bob wants to send a message to Alice, Carol, and Dave:
  - $\mathbf{m^{e_a} \bmod n_a}$
    $\mathbf{m^{e_b} \bmod n_b}$
    $\mathbf{m^{e_c} \bmod n_c}$
  - This ends up leaking information an eavesdropper can use ***especially*** if $3 = e_a = e_b = e_c$!
- Oh, and problems if both **e** and **m** are small...
- As a result, you ***can not*** just use plain RSA:
  - You need to use a "padding" scheme that makes the input random but reversible

RSA-OAEP
(Optimal asymmetric encryption padding)

- A way of processing m with a hash function & random bits
  - Effectively "encrypts" **m** replacing it with $\mathbf{X = [m,0...] \oplus G(r)}$
    - **G** and **H** are hash functions (EG SHA-256)
      $\mathbf{k_0}$ = # of bits of randomness, $\mathbf{len(m) + k_1 + k_0 = n}$
  - Then replaces r with $\mathbf{Y = H(G(r) \oplus [m,0...]) \oplus R}$
  - This structure is called a "Feistel network":
    - It is always designed to be reversible.
      Many block ciphers are based on this concept applied multiple times with **G** and **H** being functions of **k** rather than just fixed operations



- This is more than just block-cipher padding (which involves just adding simple patterns)
  - Instead it serves to both pad the bits and make the data to be encrypted "random"

But Its Not That Simple... Timing Attacks

- Using normal math, the ***time*** it takes for Alice to decrypt **c** depends on **c** and **d**
  - Ruh roh, this can leak information...
  - More complex RSA implementations take advantage of knowing **p** and **q** directly...
    but also leak timing
- People have used this to guess and then check the bits of **q** on OpenSSL
  - http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf
- And even more subtle things are possible...

```
x = C
for j = 1 to n
    x = mod(x², N)
    if dⱼ == 1 then
        x = mod(xC, N)
    end if
next j
return x
```

So How to Find Bob's Key?

- Lots of stuff later, but for now...
  The Leap of Faith!
- Alice wants to talk to Bob:
  - "Hey, Bob, tell me your public key!"
- Now on all subsequent times...
  - "Hey, Bob, tell me your public key", and check to see if it is different from what Alice remembers
- Works assuming the ***first time*** Alice talks to Bob there isn't a Man-in-the-Middle
  - ssh uses this

RSA Signatures...

- Alice computes a hash of the message **H(m)**
  - Alice then computes $\mathbf{s = (H(m))^d \bmod n}$
- Anyone can then verify
  - $\mathbf{v = s^e \bmod m = ((H(m))^d)^e \bmod n = H(m)}$
- Once again, there are "F-U"s...
  - Have to use a proper encoding scheme to do this properly and all sort of other traps
  - One particular trap: a scenario where the attacker can get Alice to repeatedly sign things (an "oracle")

But Signatures Are Super Valuable...

- They are how we can prevent a MitM!
- If Bob knows Alice's key, and Alice knows Bob's...
  - How will be "next time"
- Alice doesn't just send a message to Bob...
  - But creates a random key **k**...
  - Sends $\mathbf{E(M,K_{sess})}$, $\mathbf{E(K_{sess},B_{pub})}$, $\mathbf{S(H(M),A_{priv})}$
- Only Bob can decrypt the message, and Bob can verify the message came from Alice
  - So Mallory is SOL!

RSA Isn't The Only Public Key Algorithm

- Isn't RSA enough?
  - RSA isn't particularly compact or efficient: dealing with 2000b (comfortably secure) or 3000b (NSA-paranoia) bit operations
  - Can we get away with fewer bits?
    - Well, Diffie-Hellman isn't any better...
    - But ***elliptic curve*** Diffie-Hellman is
- RSA also had some patent issues
  - So an attempt to build public key algorithms around the Diffie-Hellman problem

El-Gamal

- Just like Diffie-Hellman...
  - Select **p** and **g**
    - These are public and can be shared:
      Note, they need to be carefully considered how to create p and g...
      Math beyond the level of this class
- Alice choses **x** randomly as her private key
  - And publishes $\mathbf{h = g^x \bmod p}$ as her public key
- Bob, to encrypt m to Alice...
  - Selects a *random* **y**, calculates $\mathbf{c_1 = g^y \bmod p}$, $\mathbf{s = h^y \bmod p = g^{xy} \bmod p}$
    - **s** becomes a shared secret between Alice and Bob
  - Maps message **m** to create **m'**, calculates $\mathbf{c_2 = m' * s \bmod p}$
- Bob then sends $\{\mathbf{c_1, c_2}\}$

El-Gamal Decryption

- Alice first calculates $\mathbf{s = c_1^x \bmod p}$
  - Then Alice calculates $\mathbf{m' = c_2 * s^{-1} \bmod p}$
  - Then Alice calculates the inverse of the mapping to get **m**
- Of course, there are problems...
  - Attacker can always change **m'** to **2m'**
  - What if Bob screws up and reuses y?
  - $\mathbf{c_2 = m_1' * s \bmod p}$
    $\mathbf{c_2' = m_2' * s \bmod p}$
  - Ruh roh, this leaks information:
    $\mathbf{c_2 / c_2' = m_1' / m_2'}$
    - So if you know $\mathbf{m_1}$...

In Practice: Session Keys...

- You use the public key algorithm to encrypt/agree on a session key..
  - And then encrypt the real message with the session key
  - You ***never*** actually encrypt the message itself with the public key algorithm
- Why?
  - Public key is ***slow***... Orders of magnitude slower than symmetric key
  - Public key may cause weird effects:
    - EG, El Gamal where an attacker can change the message to ***2m***...
      - If ***m*** had meaning, this would be a problem
      - But if it just changes the encryption and MAC keys, the main message won't decrypt

DSA Signatures...

- Again, based on Diffie-Hellman
  - Two initial parameters, **L** and **N**, and a hash function **H**
    - **L** == key length, eg 2048
      N <= len(H), e.g. 256
    - An N-bit prime **q**, an L-bit prime **p** such that **p - 1** is a multiple of **q**, and $\mathbf{g = h^{(p-1)/q} \bmod p}$ for some arbitrary **h** $(1 < h < p - 1)$
    - {**p, q, g**} are public parameters
- Alice creates her own random private key $\mathbf{x < q}$
  - Public key $\mathbf{y = g^x \bmod p}$

Alice's Signature...

- Create a random value **k** < **q**
  - Calculate $\mathbf{r = (g^k \bmod p) \bmod q}$
    - If **r** = 0, start again
  - Calculate $\mathbf{s = k^{-1} (H(m) + xr) \bmod q}$
    - If **s** = 0, start again
  - Signature is {**r**, **s**} (Advantage over an El-Gamal signature variation: Smaller signatures)
- Verification
  - $\mathbf{w = s^{-1} \bmod q}$
  - $\mathbf{u_1 = H(m) * w \bmod q}$
  - $\mathbf{u_2 = r * w \bmod q}$
  - $\mathbf{v = (g^{u_1} y^{u_2} \bmod p) \bmod q}$
  - Validate that $\mathbf{v = r}$

But Easy To Screw Up...

- **k** is not just a nonce... It must be random and ***secret***
  - If you know **k**, you can calculate **x**
- And even if you just reuse a random **k**... for two signatures $\mathbf{s_a}$ and $\mathbf{s_b}$
  - A bit of algebra proves that $\mathbf{k = (H_A - H_B) / (s_a - s_b)}$
- A good reference:
  - How knowing k tells you x: https://rdist.root.org/2009/05/17/the-debian-pgp-disaster-that-almost-was/
  - How two signatures tells you k: https://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/

And ***NOT*** theoretical: Sony Playstation 3 DRM

- The PS3 was designed to only run ***signed*** code
  - They used ECDSA as the signature algorithm
  - This prevents unauthorized code from running
  - They had an ***option*** to run alternate operating systems (Linux) that they then removed
- Of course this was catnip to reverse engineers
  - Best way to get people interested: ***remove*** Linux from a device...
- It turns for out one of the key authentication keys used to sign the firmware...
  - Ended up reusing the same k for multiple signatures!

And ***NOT*** Theoretical: Android RNG Bug + Bitcoin

- OS Vulnerability in 2013
  Android "SecureRandom" wasn't actually secure!
  - Not only was it low entropy, it would occasionally return the ***same value multiple times***
- Multiple Bitcoin wallet apps on Android were affected
  - "Pay B Bitcoin to Bob" is signed by Alice's public key using ECDSA
    - Message is broadcast publicly for all to see
  - So you'd have cases where "Pay B to Bob" and "Pay C to Carol" were signed with the same **k**
- So ***of course*** someone scanned for ***all*** such Bitcoin transactions

And ***Still*** Happens! Chromebook

- Chromebooks have a built in "Security key"
  - Enables signatures using 256b ECDSA to validate to particular websites
- There was a bug in the secure hardware!
  - Instead of using a random ***k*** that was 256b long, a bug caused it to be 32b long!
  - So an attacker who had a signature could simply try all possible ***k*** values!
- Fortunately in this case the damage was slight: this is for authenticating to a single website: each site used its own private key
- But still...
- https://www.chromium.org/chromium-os/u2f-ecdsa-vulnerability

So What To Use?

- Paranoids like me:
  Good libraries and use the parameters from NSA's CNSA suite
  - Open algorithms approved for Top Secret communication
  - Better yet, libraries that implement full protocols that use these under the hood!
- Symmetric cipher: AES: 256b
  - CFB mode, thankyouverymuch. Counter mode and modes which include counter mode can DIAF...
- Hash function: SHA-384
  - Use HMAC for MAC
- RSA: 3072b
- Diffie/Hellman: 3072b
- ECDH/ECDSA: P-384
  - But really, this is extra paranoid, 2048b RSA/DH, 256b EC, 128b AES, SHA-256 excellent in practice