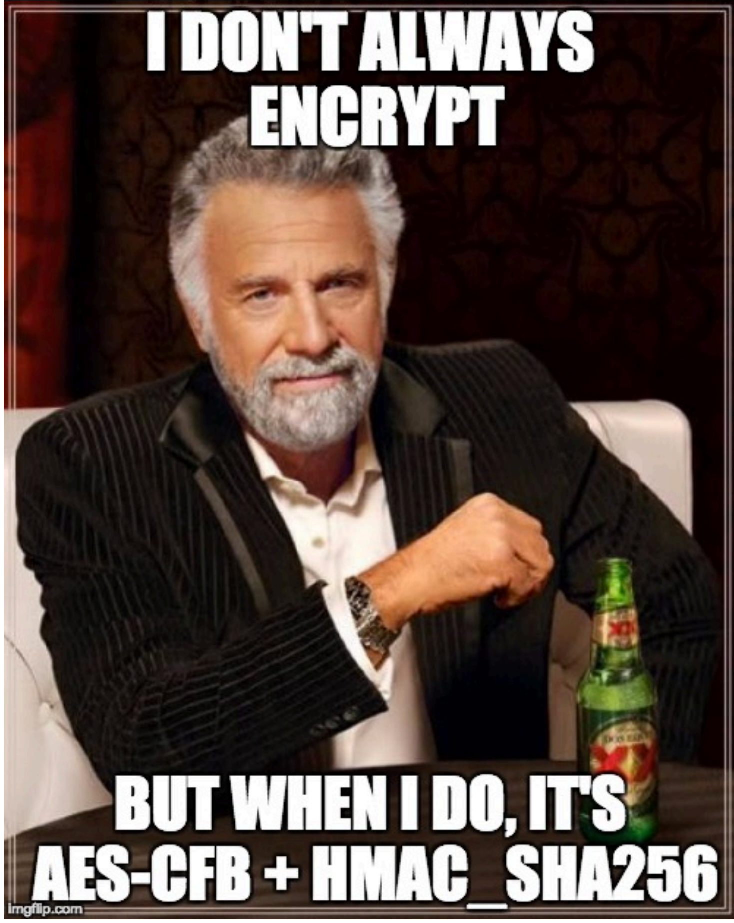
Integrity
&
Signatures



The Next Two Lectures...

- This Lecture: (Will be on MT1)
  - MACs
    - Message Authentication Codes:
      How to insure integrity with a shared secret
  - Public Key Signatures
    - How to insure integrity and authenticity using public key cryptography
- Next Lecture: (Will ***not*** be on MT1)
  - "Random" Numbers
  - Crypto-Fails
  - Crypto Successes!

Mallory the Manipulator

- Mallory is an active attacker
  - Can introduce new messages (ciphertext)
  - Can "replay" previous ciphertexts
  - Can cause messages to be reordered or discarded
- A "Man in the Middle" (MITM) attacker
  - Can be much more powerful than just eavesdropping

Encryption Does Not Provide Integrity

- Simple example: Consider a block cipher in CTR mode...
- Suppose Mallory knows that Alice sends to Bob “Pay Mal $0100”. Mallory intercepts corresponding C
  - M = “Pay Mal $0100”. C = “r4ZC#jj8qThMK”
  - $M_{10..13}$ = “0100”. $C_{10..13}$ = “ThMK”
- Mallory wants to replace some bits of C...

Encryption Does Not Provide Integrity

- Mallory computes
  - “0100” $\oplus$ $C_{10..13}$
    - Tells Mallory that section of the counter XOR:
      Remember that CTR mode computes $E_k(IV||CTR)$ and XORs it with the corresponding part of the message
  - $C'_{10..13}$ = "9999" $\oplus$ “0100” $\oplus$ $C_{10..13}$
- Mallory now forwards to Bob a full $C' = C_{0..9}||C'_{10..13}||C_{14...}$
- Bob will decrypt the message as "Pay Mal $9999"...
  - For a CTR mode cipher, Mallory can in general replace any ***known*** message M with a message M' of equal length!

Integrity and Authentication

- Integrity: Bob can confirm that what he's received is exactly the message M that was originally sent
- Authentication: Bob can confirm that what he's received was indeed generated by Alice
- Reminder: for either, confidentiality may-or-may-not matter
  - E.g. conf. not needed when Mozilla distributes a new Firefox binary
- Approach using symmetric-key cryptography:
  - Integrity via MACs (which use a shared secret key **K**)
  - Authentication arises due to confidence that only Alice & Bob have **K**
- Approach using public-key cryptography:
  - "Digital signatures" provide both integrity & authentication together
- Key building block: cryptographically strong hash functions

Cryptographically Strong Hash Functions

- A collision occurs if **x≠y** but **Hash(x) = Hash(y)**
  - Since input size > output size, collisions do happen
- A cryptographically strong **Hash(x)** provides three properties:
  - One-way: **h = Hash(x)** easy to compute, but not to invert.
    - Intractable to find ***any*** **x'** s.t. **Hash(x') = h**, for a given **h**
    - Also termed “preimage resistant”

H(🐮) =

Cryptographically Strong Hash Functions

- The other two properties of a cryptographically strong **Hash(x)**:
  - Second preimage resistant: given **x**, intractable to find **x'** s.t. **Hash(x) = Hash(x')**
  - Collision resistant: intractable to find any **x**, **y** s.t. **Hash(x)** = **Hash(y)**
- Collision resistant $\Longrightarrow$ Second preimage resistant
  - We consider them separately because given Hash might differ in how well it resists each
  - Also, the Birthday Paradox means that for n-bit Hash, finding **x-y** pair takes only $\approx 2^{n/2}$ pairs
    - Vs. potentially 2^n tries for **x'**: **Hash(x) = Hash(x')** for given **x**

SHA-256...

- SHA-256/SHA-384 are two parameters for the SHA-2 hash algorithm, returning 256b or 384b hashes
  - Works on blocks with a truncation routine to make it act on sequences of arbitrary length
  - Rough security equivalent of AES-128 and AES-256 respectively
- Is vulnerable to a ***length-extension attack***: **s** is secret
  - Mallory knows **len(s)**, **H(s)**
  - Mallory can use this to calculate **H(s||M)** for an **M** of Mallory's construction
    - Works because ***all the internal state*** at the point of calculating **H(s||...)** is derivable from **H(s)** and **len(s)**
- New SHA-3 standard (Keccak) does not have this property

Stupid Hash Tricks: Sample A File...

- BlackHat Dude claims to have 150M records stolen from Equifax...
  - How can I as a reporter verify this?
- Idea: If I can have the hacker select 10 ***random*** lines...
  - All lines are ***properly and consistently formatted***
  - And in selecting them also say something about the size of the file...
  - Voila! Verify those lines and I now know he's not full of BS
- Can I use hashing to write a small script which the BlackHat Dude can run?
  - Where I can easily verify that the 10 lines were sampled at random, and can't be faked?

Sample a File

```
#!/usr/bin/env python
import hashlib, sys
hashes = {}

for line in sys.stdin:
    line = line.strip()
    for x in range(10):
        tmp = "%s-%i" % (line, x)
        hashval = hashlib.sha256(tmp)
        h = hashval.digest()
        if x not in hashes or hashes[x][0] > h:
            hashes[x] = (h, hashval, tmp)

for x in range(10):
    h, hashval, val = hashes[x]
    print "%s=\"%s\"" % (hashval.hexdigest(), val)
```

Why does this work?

- For each x in range 0-9...
  - Calculates **H(line||x)**
  - Stores the lowest hash matching so far
- Since the hash appears random...
  - Each iteration is an independent sample from the file
  - The expected value of H(line||x) is a function of the size of the file:
    More lines, and the value is smaller
- To fake it...
  - Would need to generate fake lines, ***and see if the hash is suitably low***
  - Yet would need to make sure these fake lines semantically match!
    - Thus you can't just go "John Q Fake", "John Q Fakke", "Fake, John Q", etc...
  - And every potential fake line selected needs to check out when the reporter checks them!

Message Authentication Codes (MACs)

- Symmetric-key approach for integrity
  - Uses a shared (secret) key **K**
- Goal: when Bob receives a message, can confidently determine it hasn't been altered
  - In addition, whomever sent it must have possessed **K**
    (⇒ message authentication, sorta...)
- Conceptual approach:
  - Alice sends **{M, T}** to Bob, with tag **T = MAC(K, M)**
    - Note, **M** could instead be **C = E_K'(M)**, but not required
  - When Bob receives **{M', T'}**, Bob checks whether **T' = MAC(K, M')**
    - If so, Bob concludes message untampered, came from Alice
    - If not, Bob discards message as tampered/corrupted

Requirements for Secure MAC Functions

- Suppose MITM attacker Mallory intercepts Alice's **{M, T}** transmission …
  - … and wants to replace **M** with altered **M***
  - … but doesn't know shared secret key **K**
- We have secure integrity if MAC function **T = MAC(M, K)** has two properties:
  - Mallory can't compute **T* = MAC(M*, K)**
    - Otherwise, could send Bob **{M*, T*}** and fool him
  - Mallory can't find **M**** such that **MAC(M**, K) = T**
    - Otherwise, could send Bob **{M**, T}** and fool him
- These need to hold even if Mallory can observe many **{M_i, T_i}** pairs, including for **M_i**'s she chose

The best MAC construction: HMAC

- Idea is to turn a hash function into a MAC
  - Since hash functions are often much faster than encryption
  - While still maintaining the properties of being a cryptographic hash
- Reduce/expand the key to a single hash block
- XOR the key with the i_pad
  - 0x363636... (one hash block long)
- Hash ((K ⊕ i_pad) || message)
- XOR the key with the o_pad
  - 0x5c5c5c...
- Hash ((K ⊕ o_pad) || first hash)

```
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
       key = key || 0x00
    }
    o_key_pad = 0x5c5c... ⊕ key
    i_key_pad = 0x3636... ⊕ key
    return hash(o_key_pad ||
                hash(i_key_pad || message))
}
```

Why This Structure?

- i_pad and o_pad are slightly arbitrary
    - But it is necessary for security for the two values to be different
        - So for paranoia chose very different bit patterns
- Second hash prevents appending data
    - Otherwise attacker could add more to the message and the HMAC and it would still be a valid HMAC for the key if the underlying hash is vulnerable to length extension attacks
        - Wouldn't be a problem with the key at the ***end*** but at the start makes it easier to capture intermediate HMACs on partial files
- Is a Pseudo Random Function if the underlying hash is a PRF
    - AKA if you can break this, ***you can break the hash***!

```
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
       key = key || 0x00
    }
    o_key_pad = 0x5c5c... ⊕ key
    i_key_pad = 0x3636... ⊕ key
    return hash(o_key_pad ||
                hash(i_key_pad || message))
}
```

Great Properties of HMAC...

- It is still a hash function!
  - So all the good things of a cryptographic hash:
    An attacker or ***even the recipient*** shouldn't be able to calculate **M** given **HMAC(M,K)**
  - An attacker who doesn't know **K** can't even verify if **HMAC(M,K) == M**
    - Very different from the hash alone, and potentially very useful:
      Attacker can't even brute force try to find **M** based on **HMAC(M,K)**!
- Its probably safe if you screw up and use the same key for both MAC and Encrypt
  - Since it is a different algorithm than the encryption function...
  - ***But you shouldn't do this anyway***!

Considerations when using MACs

- Along with messages, can use for data at rest
  - E.g. laptop left in hotel, providing you don't store the key on the laptop
  - Can build an efficient data structure for this that doesn't require re-MAC'ing over entire disk image when just a few files change
- MACs in general provide ***no promise*** not to leak info about message
  - Compute MAC on ciphertext if this matters
  - Or just use HMAC, which ***does*** promise not to leak info if the underlying hash function doesn't
- ***NEVER*** use the same key for MAC and Encryption...
  - Known "FU-this-is-crypto" scenarios reusing an encryption key for MAC in some algorithms when its the same underlying block cipher for both

AEAD Encryption Modes

- New Modern Encryption Modes:
  Authenticated Encryption with Additional Data
- These modes provide confidentiality and integrity
  - Effectively including a MAC
- Can also provide integrity over additional unencrypted data
- Warning, however:
  - These modes tend to include CTR mode as the base encryption mode...
    Which ***catastrophically*** fails if you ever reuse an IV

Passwords

- The password problem:
  - User Alice authenticates herself with a password ***P***
- How does the site verify later that Alice knows ***P***?
- Classic:
  - Just store {**Alice, *P***} in a file...
- But what happens when the site is hacked?
  - The attacker now knows Alice's password!
- Enter "Password Hashing"

Password Hashing

- Instead of storing {**Alice, *P***}...
  - Store {**Alice, H(*P*)**}
- To verify Alice, when she presents ***P***
  - Compute **H(*P*)** and compare it with the stored value
- Problem: Brute Force tables...
  - Most people chose bad passwords...
    And these passwords are known
  - Bad guy has a huge file...
    - **H(P1), P1**
      H(P2), P2
      H(P3), P3...
  - Ways to make this more efficient ("Rainbow Tables")

A Sprinkle of Salt...

- Instead of storing {**Alice, H(P)**}, also have a user-specific string, the "Salt"
  - Now store {**Alice, Salt, H(P||Salt)**}
  - The salt ideally should be both long and random, but it isn't considered "secret"
- As long as the salt is unique...
  - An attacker who captures the password file has to ***brute force*** Alice's password on its own
- Its still an "off-line attack" (Attacker can do all the computation he wants) but...
  - At least the attacker can't ***precompute*** possible solutions

Slower Hashes...

- Most cryptographic hashes are designed to be ***fast***
  - After all, that is the point: they should not only turn **H(🐮)** to hamburger... they do it with the speed of a woodchipper
- But for password hashes, we ***want*** it to be slow!
  - Its OK if it takes a good fraction of a second to ***check*** a password
    - Since you only need to do it once for each legitimate usage of that password
  - But the attacker needs to do it for each password he wants to try
- Slower hashes don't change the ***asymptotic difficulty*** of password cracking but can have huge practical impact
  - Slow rate by a factor of 10,000 or more!

PBKDF2

- "Password Based Key Derivation Function 2"
  - Designed to produce a long "random" bitstream derived from the password
- Used for both a password hash and to generate keys derived from a user's password
  - PKBDF(PRF, P, S, c, len):
    - **PRF** == Pseudo Random Function (e.g. HMAC-SHA256)
    - **P** == Password
    - **S** == Salt
    - **c** == Iteration count
    - **len** == Number of bits/bytes requested
    - **DK** == Derived Key

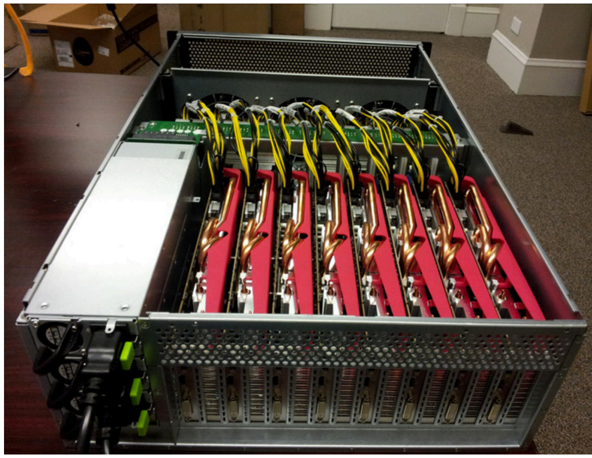
```
PKBDF(PRF,P,S,c,len){
  DK = ""
  for i = 1,range(len/blocksize)+1){
    DK = DK || F(PRF,P,S,c,i)
  }
  return DK[0:len]
}

F(PRF,P,S,c,i){
  UR = U = PRF(P, S||INT_32(i))
  for j = 2; j <= c; ++j {
    U = PRF(P, U)
    UR = UR ^ U
  }
  return UR
}
```

Comments on PBKDF2

- Allows you to get effectively an arbitrary long string from a password
  - ***Assuming*** the user's password is strong/high entropy
- Very good for getting a bunch of symmetric keys from a single password
  - You can also use this to seed a pRNG for generating a "random" public/private key pair
- Designed to be slow in computation...
  - But it does ***not*** require a lot of memory:
    Other functions are also expensive in memory as well, e.g. scrypt and argon2

Passwords...

- If an attacker can do an ***offline*** attack, your password must be ***really good***
  - Attacker simply tries a huge number of passwords in parallel using a GPU-based computer
  - So you need a ***high entropy*** password:
    - Even xkcd-style is only 10b/word, so need a 7 or more ***random word*** passphrase to resist a determined attacker
- Life is far better is if the attacker can only do ***online*** attacks:
  - Query the device and see if it works
  - Now limited to a few tries per second and ***no parallelism!***



... and iPhones

- Apple's security philosophy:
  - In your hands, the phone should be everything
  - In anybody else's, it should (ideally) be an inert "brick"
- Apple uses a small co-processor in the phone to handle the cryptography
  - The "Secure Enclave"
- The rest of the phone is untrusted
  - Notably the memory: ***All*** data must be encrypted:
    The CPU requests that the Secure Enclave unencrypt data and some data (e.g., your credit card for ApplePay) is only readable by the Secure Enclave
- They also have an ability to effectively erase a small piece of memory
  - "Effaceable Storage": this takes a good amount of EE trickery

Crypto and the iPhone Filesystem

- A lot of keys encrypted by keys...
  - But there is a random master key, $\mathbf{k}_{\mathbf{phone}}$, that is the root of all the other keys
- Need to store $\mathbf{k}_{\mathbf{phone}}$ encrypted by the user's password in the flash memory
  - PBKDF2(P,...) = $\mathbf{k}_{\mathbf{user}}$
- But how to prevent an off-line brute-force attack?
  - Also have a 256b ***random*** secret burned into the Secure Enclave
    - Need to take apart the chip to get this!
- Now the user key is not just a function of P, but P||secret
  - Without the secret, ***can not*** do an offline attack
- All ***online*** attacks have to go through the secure enclave
  - After 5 tries, starts to slow down
  - After 10 tries, can (optionally) nuke $\mathbf{k}_{\mathbf{phone}}$!
    - Erase just that part of memory -> effectively erases the entire phone!

Backups...

- Of course there is a ***necessary*** weakness:
  - Backing up the phone copies all the data off in a form not encrypted using the in-chip secret
    - After all, you need to be able to recover it onto a new phone!
- So someone who can get your phone...
  And can somehow managed to have it unlocked
  - Thief, abusive boyfriend, cop...
    - Hold it up to your face (iPhone X) or Fingerprint (5s or beyond)
    - And then sync it with a new computer
- Change of policy for iOS-11:
  - Now you also need to put in the passcode to trust a new computer:
    Can't create a backup without knowing the passcode

So Far...

- We have ***symmetric*** key encryption...
  - But that requires Alice and Bob knowing a key in advance
- We have ***symmetric*** integrity with MACs...
  - But anyone who can ***verify*** the integrity can also modify the message
- Goal of public key is to change that
  - Allows creation of a symmetric key in the presence of an adversary
  - Allows creation of a message to Alice by anybody but only Alice can decrypt
  - Allows creation of a message exclusively by Alice than anybody can verify

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
- RSA:
  - Provide a message to a recipient only knowing the recipient's ***public key***
- RSA signatures:
  - Provide a message that anyone can prove was generated with a ***private key***

Reminder:
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

Ephemeral Diffie/Hellman

- **$K = g^{ab} \bmod p$** 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.**

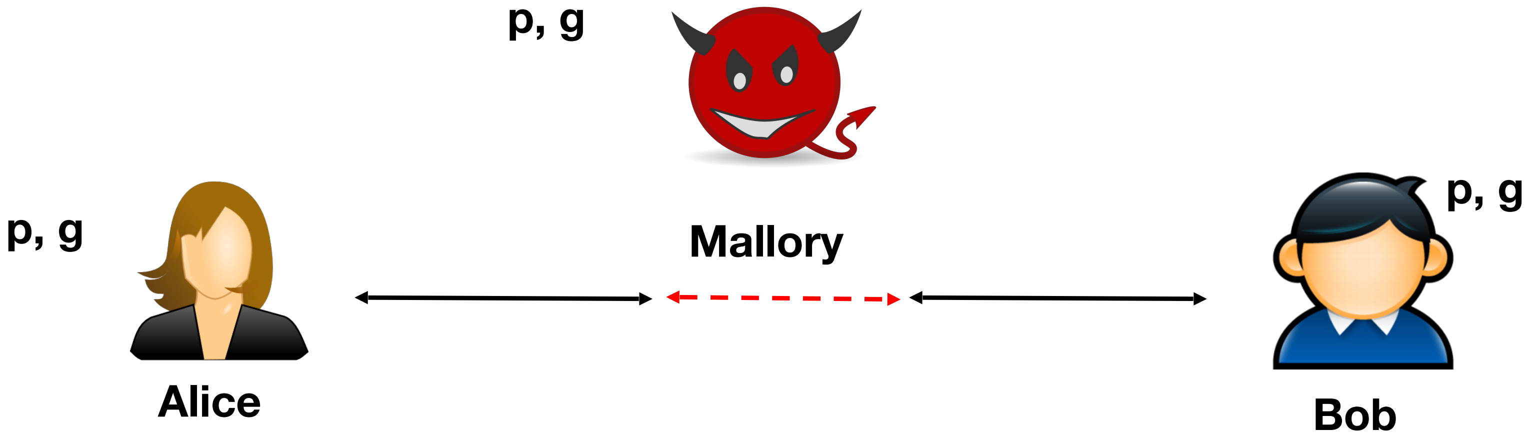
Diffie Hellman is part of more generic problem

- 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

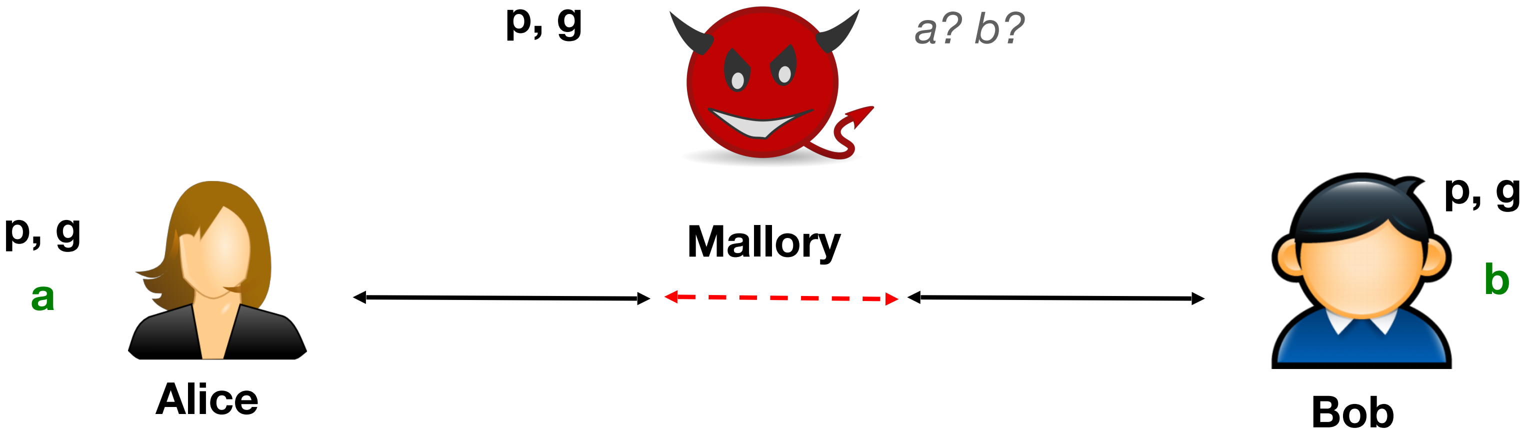
- What if Alice and Bob aren't facing a passive eavesdropper
  - But instead are facing Mallory, an ***active*** Man-in-the-Middle
- Mallory has the ability to change messages:
  - Can remove messages and add his own
- Lets see... Do you think DHE will still work as-is?

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



p, g

a? b?

Mallory

p, g

a

Alice

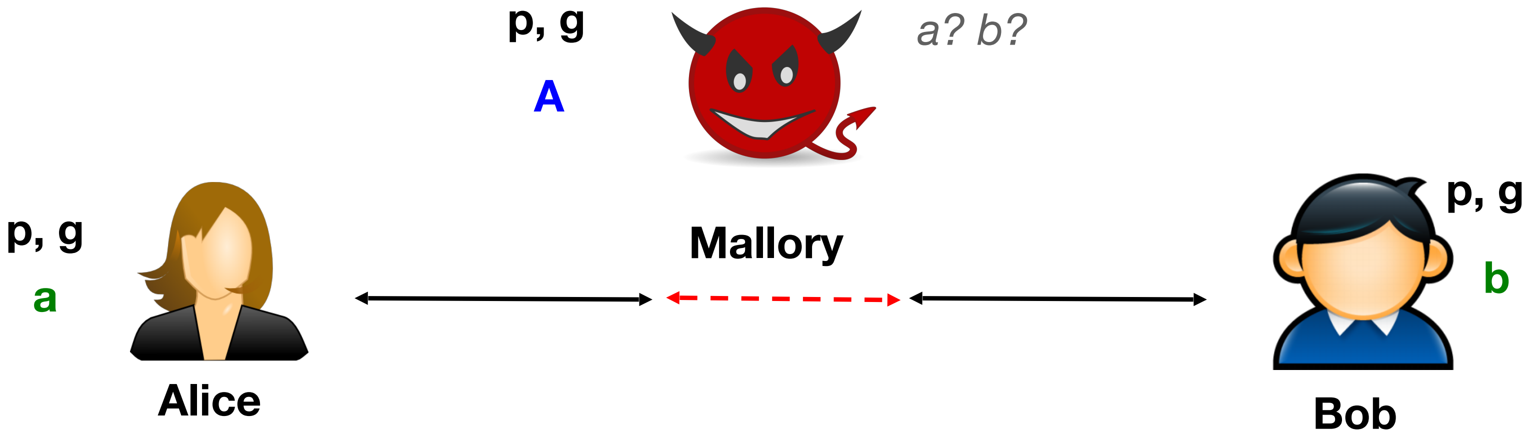
p, g

b

Bob

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

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



p, g

A

Mallory

a? b?

p, g

a

Alice

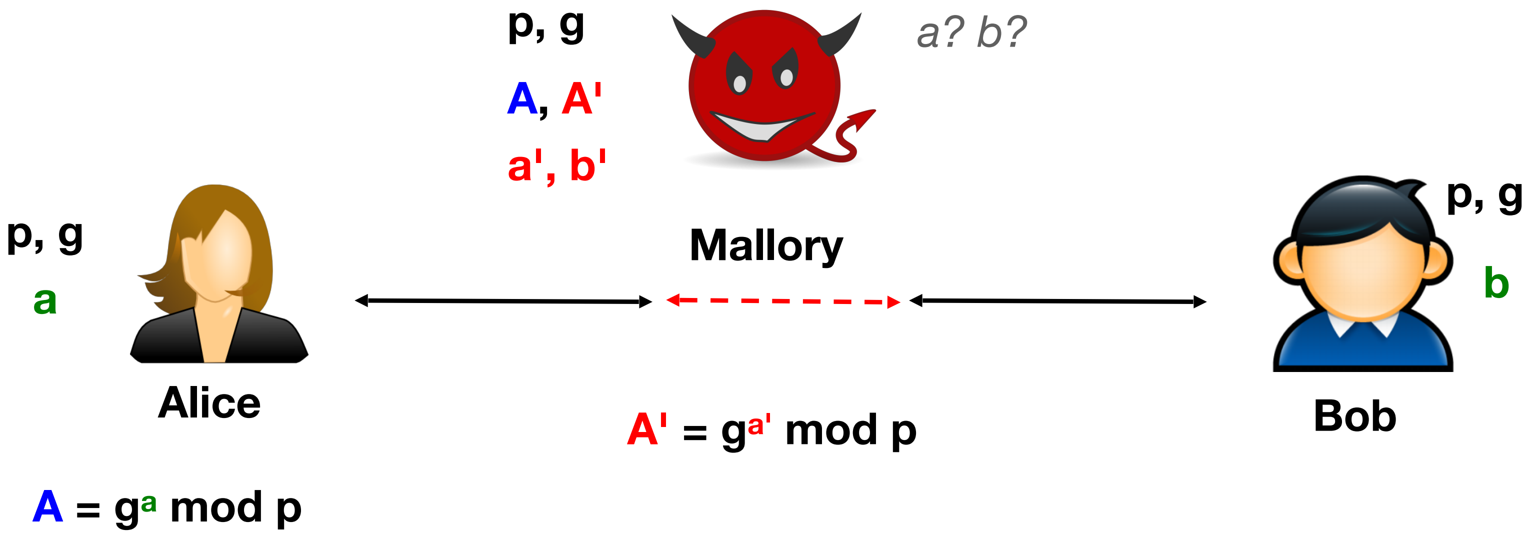
p, g

b

Bob

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

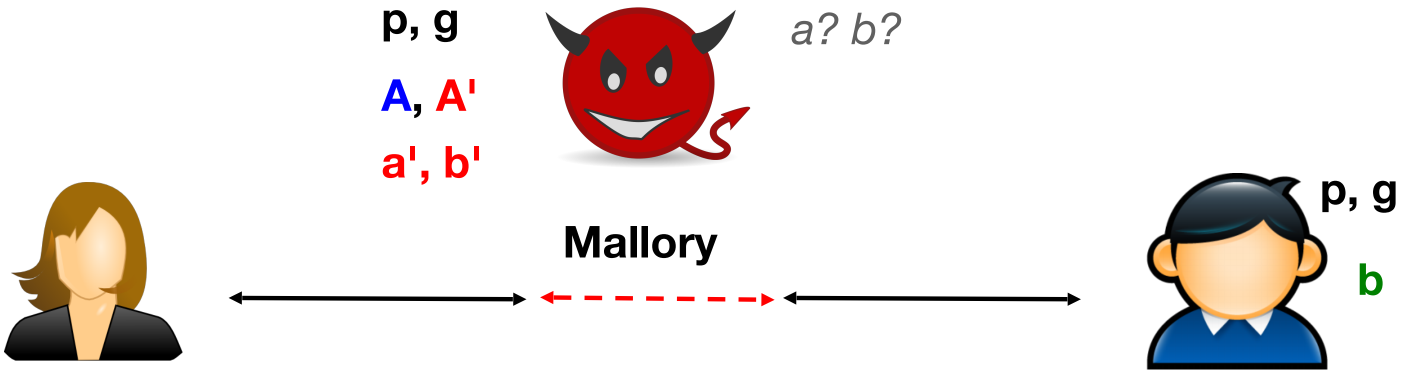
4. Alice sends A = g^{a} mod p to Bob
5. Mallory prevents Bob from receiving A



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

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

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



p, g

A, A'

a', b'

a? b?

Mallory

p, g

a

Alice

p, g

b

Bob

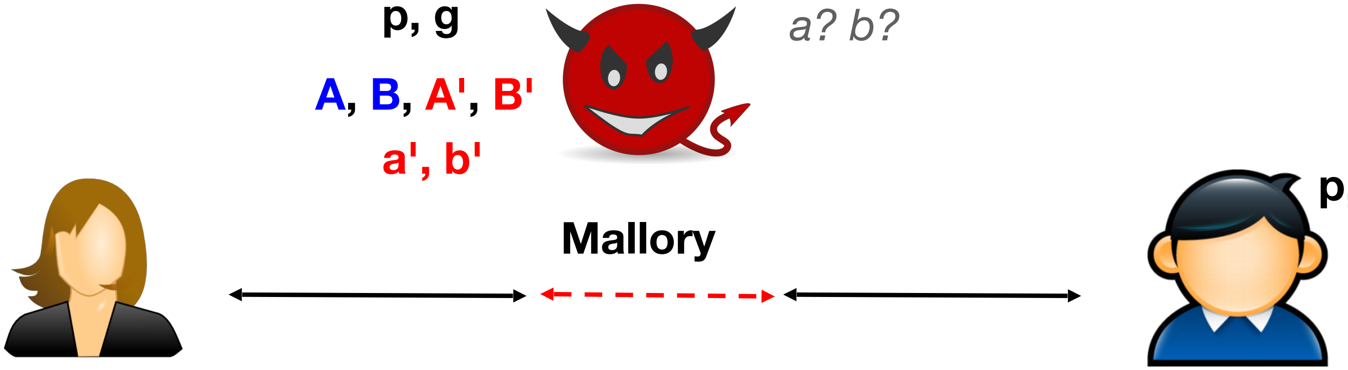
$A' = g^{a'} \bmod p$

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

A'

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

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



p, g

A, B, A', B'

a', b'

a? b?

Mallory

p, g

a

Alice

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

p, g

b

Bob

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

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

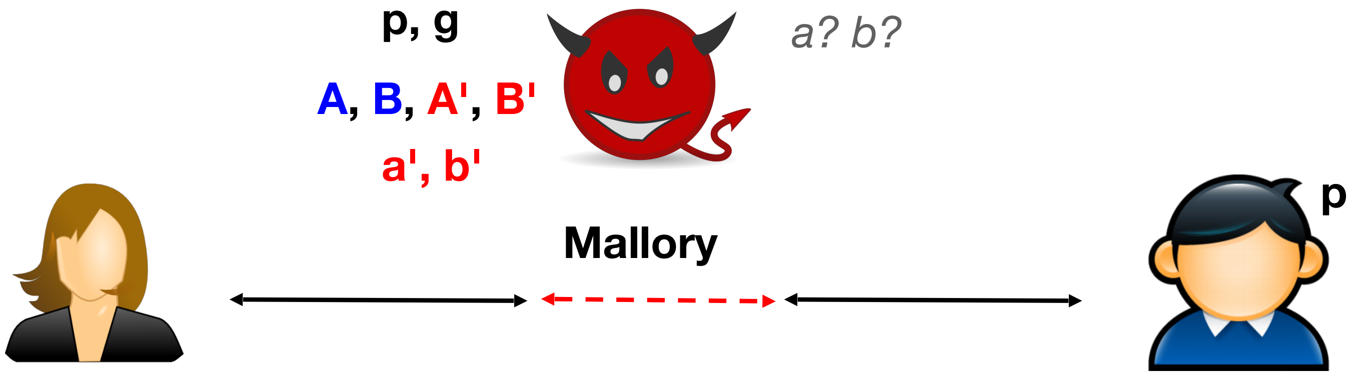
A'

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

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*



p, g

A, B, A', B'

a', b'

a? b?

Mallory

p, g

a

Alice

p, g

b

Bob

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

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

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

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

$A = g^a \bmod p$

B'

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

A'

$g^b \bmod p = B$

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

Public Key Cryptography: 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 $d = e^{-1} \bmod \phi(n)$
  - 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 $\mathbf{c = m^e \bmod n}$
  - 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 $\mathbf{m = c^d \bmod n = m^{ed} \bmod n}$
- 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:* $a^{\phi(n)} \bmod n = 1$

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

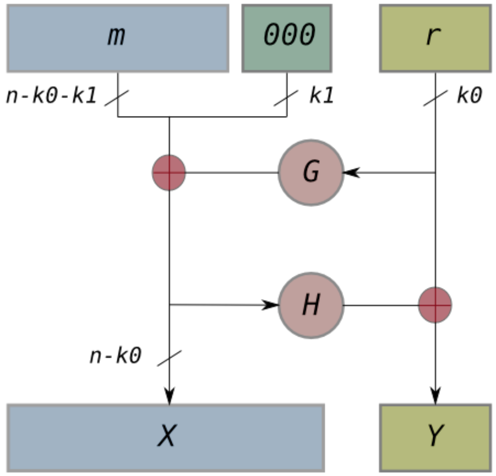
(believed) Eve can recover M from C *iff* Eve can factor n=p·q

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 **$X = [m,0\ldots] \oplus G(r)$**
    - **G** and **H** are hash functions (EG SHA-256)
      k_0 = # of bits of randomness, **$\mathrm{len}(m) + k_1 + k_0 = n$**
  - Then replaces r with **$Y = H(G(r) \oplus [m,0\ldots]) \oplus R$**
  - 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"
- The RSA mode we provide in the project uses this mode



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

RSA Signatures... Just Run RSA Backwards!

- Alice computes a hash of the message **H(m)**
  - Alice then computes **$s = (H(m))^d \bmod n$**
- Anyone can then verify
  - **$v = s^e \bmod m = ((H(m))^d)^e \bmod n = H(m)$**
- 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")

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!

Signatures Enable Ephemeral Diffie/Hellman

- Bob knows (somehow) Alice's public key...
  - We will find out how later when we talk about ***certificates***
  - Or, as in the project, the "trusted keystore" can tell you Alice's public key
- Now Alice doesn't just send $\mathbf{g^a}$, but also $\mathbf{sign(g^a,K_{alice})}$
  - As a consequence, now Mallory can't play the MitM!
- And yet we have "forward secrecy"
  - Even if Eve gets Alice's private key, she can't decrypt old messages or new messages
  - Even if Malory gets Alice's private key, he can only intercept new messages as a man-in-the-middle