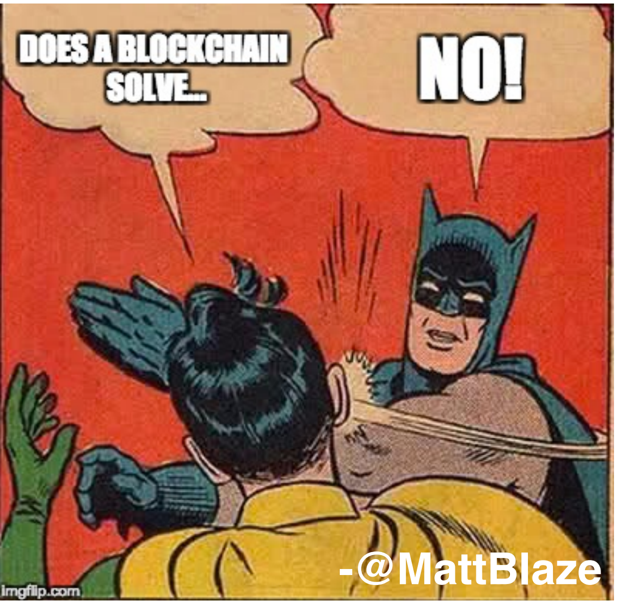
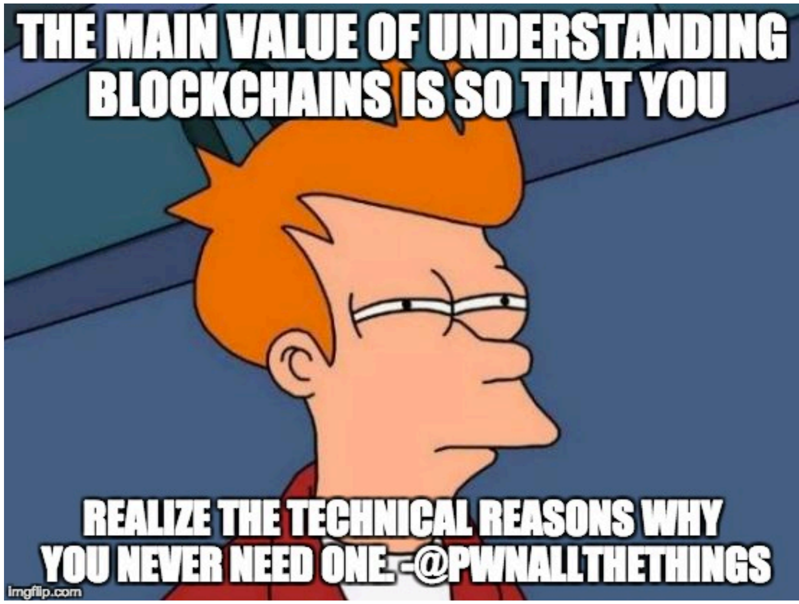
RNG & Public Key





Announcements:

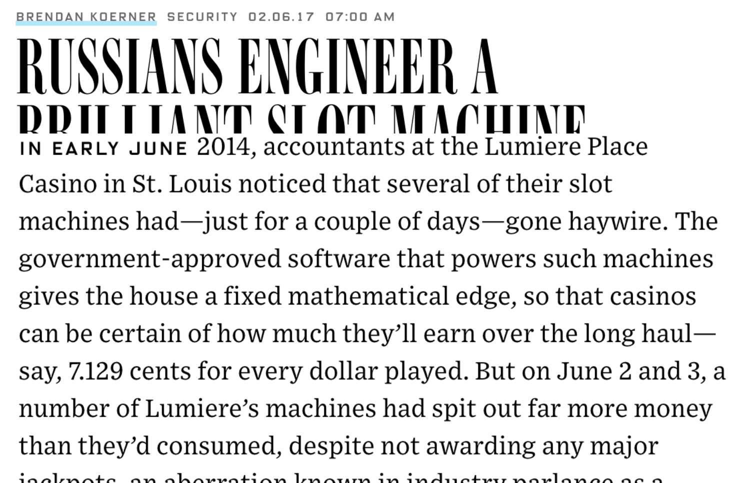
- Midterm 1: Sept 25, 5-6:30pm
  - Two rooms: 155 Dwinelle and 2050 VLSB
- Which room should you go to?
  - Take the last 3 digits of your student ID:
  - If more odd than even numbers, 155 Dwinelle
  - Otherwise, 2050 VLSB
- DSP students needing extra time etc, use the exam coordination Piazza folder
- GO! GO GO GO GO GO!!!
  - Project 2 will be in Go. We won't release it until at least the 25th
  - But start learning Go now.

But A Lot More Uses for Random Numbers...

- The key foundation for all modern cryptographic systems is often not encryption but these "random" numbers!
- So many times you need to get something random:
  - A random cryptographic key
  - A random initialization vector
  - A "nonce" (use-once item)
  - A unique identifier
  - Stream Ciphers
- If an attacker can ***predict*** a random number things can catastrophically fail

Breaking Slot Machines

- Some casinos experienced unusual bad "luck"
  - The suspicious players would wait and then all of a sudden try to play
- The slot machines have ***predictable*** pRNG
  - Which was based on the current time & a seed
- So play a little...
  - With a cellphone watching
  - And now you know when to press "spin" to be more likely to win
- Oh, and this ***never*** effected Vegas!
  - Evaluation standards for Nevada slot machines specifically designed to address this sort of issue



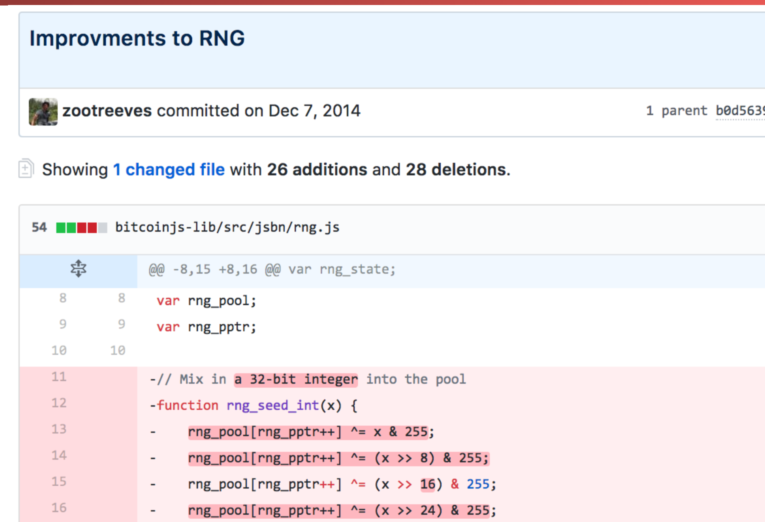
BRENDAN KOERNER SECURITY 02.06.17 07:00 AM

RUSSIANS ENGINEER A BRILLIANT SLOT MACHINE

IN EARLY JUNE 2014, accountants at the Lumiere Place Casino in St. Louis noticed that several of their slot machines had—just for a couple of days—gone haywire. The government-approved software that powers such machines gives the house a fixed mathematical edge, so that casinos can be certain of how much they'll earn over the long haul—say, 7.129 cents for every dollar played. But on June 2 and 3, a number of Lumiere's machines had spit out far more money than they'd consumed, despite not awarding any major

Breaking Bitcoin Wallets

- blockchain.info supports "web wallets"
  - Javascript that protects your Bitcoin
- The private key for Bitcoin needs to be random
  - Because otherwise an attacker can spend the money
- An "Improvment" [sic] to the RNG reduced the entropy (the actual randomness)
  - Any wallet created with this improvment was brute-forceable and could be stolen



TRUE Random Numbers

- True random numbers generally require a physical process
- Common circuit is an unusable ring oscillator built into the CPU
  - It is then sampled at a low rate to generate true random bits which are then fed into a pRNG on the CPU
- Other common sources are human activity measured at very fine time scales
  - Keystroke timing, mouse movements, etc
    - "Wiggle the mouse to generate entropy for a key"
  - Network/disk activity which is often human driven
- More exotic ones are possible:
  - Cloudflare has a wall of lava lamps that are recorded by a HD video camera which views the lamps through a rotating prism

Combining Entropy

- The general procedure is to combine various sources of entropy
- The goal is to be able to take multiple crappy sources of entropy
  - Measured in how many bits:
    A single flip of a coin is 1 bit of entropy
  - And combine into a value where the entropy is the minimum of the sum of all entropy sources (maxed out by the # of bits in the hash function itself)
  - **N-1** bad sources and **1** good source -> good pRNG state

Pseudo Random Number Generators (aka Deterministic Random Bit Generators)

- Unfortunately one needs a ***lot*** of random numbers in cryptography
  - More than one can generally get by just using the physical entropy source
- Enter the pRNG or DRBG
  - If one knows the state it is entirely predictable
  - If one doesn't know the state it should be indistinguishable from a random string
- Three operations
  - Instantiate: (aka Seed) Set the internal state based on the real entropy sources
  - Reseed: Update the internal state based on both the previous state and ***additional entropy***
    - The big different from a simple stream cipher
  - Generate: Generate a series of random bits based on the internal state
    - Generate can also optionally add in additional entropy
- **instantiate(entropy)**
  reseed(entropy)
  generate(bits, {optional entropy})

Properties for the pRNG

- Can a pRNG be truly random?
  - No. For seed length **s**, it can only generate at most $\mathbf{2^s}$ distinct possible sequences.
- A cryptographically strong pRNG "looks" truly random to an attacker
  - Attacker ***cannot distinguish*** it from a random sequence

Prediction and Rollback Resistance

- A pRNG should be predictable only if you know the internal state
  - It is this predictability which is why its called "pseudo"
- If the attacker does not know the internal state
  - The attacker should not be able to distinguish a truly random string from one generated by the pRNG
- It should also be rollback-resistant
  - Even if the attacker finds out the state at time T, they should not be able to determine what the state was at T-1
  - More precisely, if presented with two random strings, one truly random and one generated by the pRNG at time T-1, the attacker should not be able to distinguish between the two

Why "Rollback Resistance" is Essential

- Assume attacker, at time T, is able to obtain all the internal state of the pRNG
  - How? E.g. the pRNG screwed up and instead of an IV, released the internal state, or the pRNG is bad...
- Attacker observes how the pRNG was used
  - T_{-1} = Session key
    T_0 = Nonce
- Now if the pRNG doesn't resist rollback, and the attacker gets the state at T_0, attacker can know the session key! And we are back to...

More on Seeding and Reseeding

- Seeding should take all the different physical entropy sources available
  - If one source has 0 entropy, it ***must not*** reduce the entropy of the seed
  - We can shove a whole bunch of low-entropy sources together and create a high-entropy seed
- Reseeding ***adds*** in even more entropy
  - **F(internal_state, new material)**
  - Again, even if reseeding with 0 entropy, it ***must not*** reduce the entropy of the seed
- Entropy (most of the time) needs to be confidential

Probably the best pRNG/DRBG: HMAC_DRBG

- Generally believed to be the best
  - Accept no substitutes!
- Two internal state registers, ***V*** and ***K***
  - Each the same size as the hash function's output
- ***V*** is used as (part of) the data input into HMAC, while ***K*** is the key
- If you can break this pRNG you can ***either break the underlying hash function*** or ***break a significant assumption about how HMAC works***
  - Yes, security proofs sometimes are a very good thing and actually do work

HMAC_DRBG Generate

- The basic generation function
- Remarks:
  - It requires one HMAC call per blocksize-bits of state
  - Then two more HMAC calls to update the internal state
- Prediction resistance:
  - If you can distinguish new **K** from random when you don't know old **K**:
    You've distinguished HMAC from a random function!
    Which means you've either broken the hash or the HMAC construction
- Rollback resistance:
  - If you can learn old **K** from new **K** and **V**:
    You've reversed the hash function!

```
function hmac_drbg_generate (state, n) {
  tmp = ""
  while(len(tmp) < N){
     state.v = hmac(state.k,state.v)
     tmp = tmp || state.v
  }
  // Update state w no input
  state.k = hmac(state.k, state.v || 0x00)
  state.v = hmac(state.k, state.v)
  // Return the first N bits of tmp
  return tmp[0:N]
}
```

HMAC_DRBG Update

- Used instead of the "no-input update" when you have additional entropy on the generate call
- Used standalone for both instantiate (**state.k = state.v = 0**) and reseed
- Designed so that even if the attacker controls the input but doesn't know **k**:
  - The attacker should not be able to predict the new **k**

```
function hmac_drbg_update (state, input) {
  state.k = hmac(state.k, state.v || 0x00
                          || input)
  state.v = hmac(state.k, state.v)
  state.k = hmac(state.k, state.v || 0x01
                          || input)
  state.v = hmac(state.k, state.v)
}
```

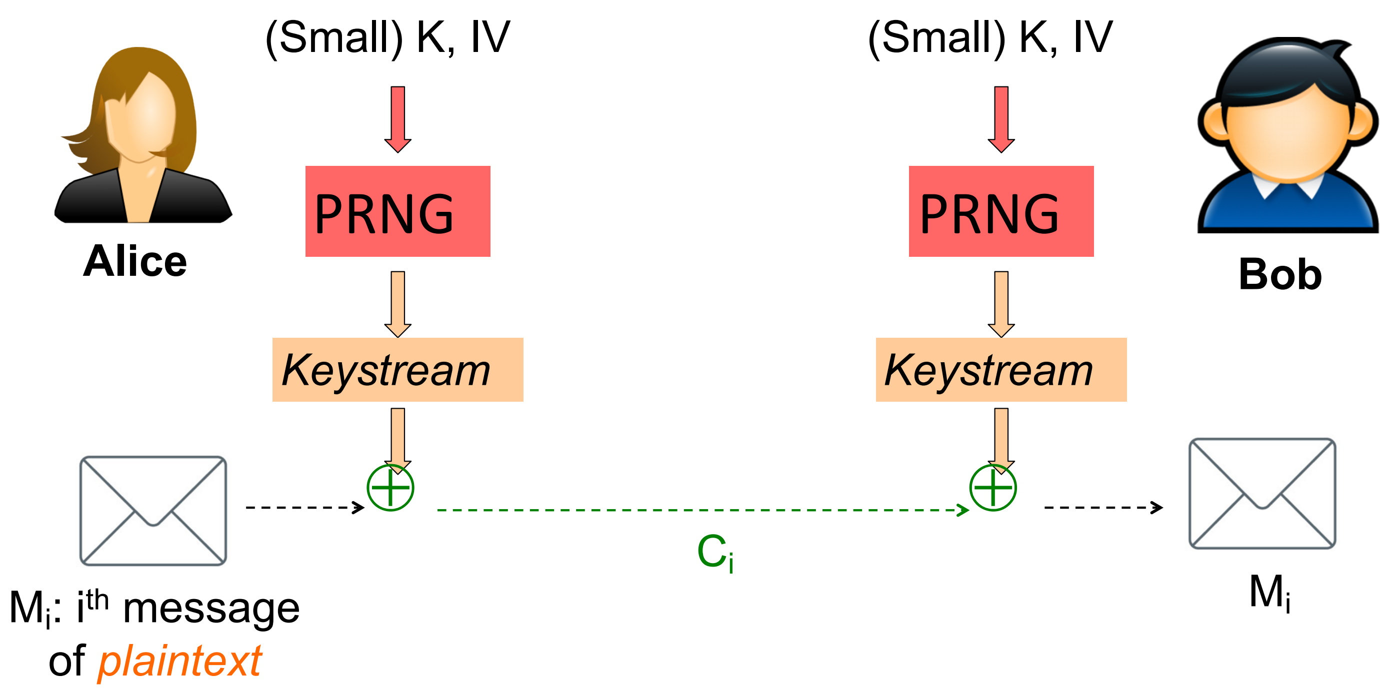
Stream ciphers

- Block cipher: fixed-size, stateless, requires “modes” to securely process longer messages
- Stream cipher: keeps state from processing past message elements, can continually process new elements
- Common approach: “one-time pad on the cheap”:
  - XORs the plaintext with some “random” bits
- But: random bits ≠ the key (as in one-time pad)
  - Instead: output from cryptographically strong pseudorandom number generator (pRNG)
  - Anyone who actually calls this a "One Time Pad" is selling snake oil!

Building Stream Ciphers

- Encryption, given key **K** and message **M**:
  - Choose a random value IV
  - **E(M, K)** = **pRNG(K, IV)** ⊕ **M**
- Decryption, given key **K**, ciphertext **C**, and initialization vector **IV**:
  - **D(C, K) = PRNG(K, IV)** ⊕ **C**
- Can encrypt message of any length because pRNG can produce any number of random bits...
  - But in practice, for an n-bit seed pRNG, stop at $\mathbf{2^{n/2}}$. Because, of course...

Using a PRNG to Build a Stream Cipher



CTR mode is (mostly) a stream cipher

- **E(ctr,K)** should look like a series of pseudo random numbers...
  - But after a large amount it is ***slightly*** distinguishable!
- Since it is actually a pseudo-random ***permutation***...
  - For a cipher using 128b blocks, you will never get the same 128b number until you go all the way through the 2^{128} possible entries on the counter
  - Reason why you want to stop after 2^{64}
    - if you are foolish enough to use CTR mode in the first place
- Also very minor information leakage:
  - If $\mathbf{C_i} = \mathbf{C_j}$, for $\mathbf{i} \mathrel{\mathbf{!=}} \mathbf{j}$, it follows that $\mathbf{M_i} \mathrel{\mathbf{!=}} \mathbf{M_j}$

UUID: Universally Unique Identifiers

- You got to have a "name" for something...
  - EG, to store a location in a filesystem
- Your name ***must*** be unique...
  - And your name ***must*** be unpredictable!
- Just chose a ***random*** value!
  - UUID: just chose a 128b random value
    - Well, it ends up being a 122b random value with some signaling information
  - A good UUID library uses a cryptographically-secure pRNG that is properly seeded
- Often written out in hex as:
  - 00112233-4455-6677-8899-aabbccddeeff

What Happens When The Random Numbers Goes Wrong...

- Insufficient Entropy:
  - Random number generator is seeded without enough entropy
- Debian OpenSSL CVE-2008-0166
  - In "cleaning up" OpenSSL (Debian 'bug' #363516), the author 'fixed' how OpenSSL seeds random numbers
    - Because the code, as written, caused Purify and Valgrind to complain about reading uninitialized memory
  - Unfortunate cleanup reduced the pRNG's seed to be ***just*** the process ID
    - So the pRNG would only start at one of ~30,000 starting points
- This made it easy to find private keys
  - Simply set to each possible starting point and generate a few private keys
  - See if you then find the corresponding public keys anywhere on the Internet

http://blog.dieweltistgarnichtso.net/Caprica,-2-years-ago

And Now Lets
Add Some RNG Sabotage...

- The Dual_EC_DRBG
  - A pRNG pushed by the NSA behind the scenes based on Elliptic Curves
- It relies on two parameters, ***P*** and ***Q*** on an elliptic curve
  - The person who generates ***P*** and selects ***Q=eP*** can predict the random number generator, regardless of the internal state
- It also ***sucked!***
  - It was horribly slow and even had subtle biases that shouldn't exist in a pRNG: You could distinguish the upper bits from random!
- Now this was spotted fairly early on...
  - Why should anyone use such a horrible random number generator?

Well, anyone not paid that is...

- RSA Data Security accepted ~~30 pieces of silver~~ $10M from the NSA to implement Dual_EC in their RSA BSAFE library
  - And silently make it the default pRNG
- Using RSA's support, it became a NIST standard
  - And inserted into other products...
- And then the Snowden revelations
  - The initial discussion of this sabotage in the NY Times just vaguely referred to a Crypto talk given by Microsoft people...
    - That everybody quickly realized referred to Dual_EC

But this is insanely powerful...

- It isn't just forward prediction but being able to run the generator backwards!
  - Which is why Dual_EC is so nasty:
    Even if you know the internal state of HMAC_DRBG it has rollback resistance!
- In TLS (HTTPS) and Virtual Private Networks you have a motif of:
  - Generate a random session key
  - Generate some other random data that's ***public visible***
    - EG, the IV in the encrypted channel, or the "random" nonce in TLS
    - Oh, and an NSA sponsored "standard" to spit out even more "random" bits!
- If you can run the random number generator ***backwards***, you can find the session key

It Got Worse: Sabotaging Juniper

- Juniper also used Dual_EC in their Virtual Private Networks
  - "But we did it safely, we used a different **Q**"
- Sometime later, someone else noticed this...
  - "Hmm, **P** and **Q** are the keys to the backdoor...
    Lets just hack Juniper and rekey the lock!"
    - And whoever put in the first Dual_EC then went "Oh crap, we got locked out but we can't do anything about it!"
- Sometime later, someone else goes...
  - "Hey, lets add an ssh backdoor"
- Sometime later, Juniper goes
  - "Whoops, someone added an ssh backdoor, lets see what else got F'ed with, oh, this # in the pRNG"
- And then everyone else went
  - "Ohh, patch for a backdoor. Lets see what got fixed. Oh, these look like Dual_EC parameters..."

Sabotaging "Magic Numbers" In General

- Many cryptographic implementations depend on "magic" numbers
  - Parameters of an Elliptic curve
  - Magic points like ***P*** and ***Q***
  - Particular prime ***p*** for Diffie/Hellman
  - The content of S-boxes in block cyphers
- Good systems should cleanly describe how they are generated
  - In some sound manner (e.g. AES's S-boxes)
  - In some "random" manner defined by a pRNG with a specific seed
    - Eg, seeded with "Nicholas Weaver Deserves Perfect Student Reviews"... Needs to be very low entropy so the designer can't try a gazillion seeds

Because Otherwise You Have Trouble...

- Not only Dual-EC's ***P*** and ***Q***
- Recent work: 1024b Diffie/Hellman moderately impractical...
  - But you can create a sabotaged prime that is 1/1,000,000 the work to crack!
    And the most often used "example" ***p***'s origin is lost in time!
- It can cast doubt ***even when a design is solid***:
  - The DES standard was developed by IBM but with input from the NSA
    - Everyone was suspicious about the NSA tampering with the S-boxes...
    - They did: The NSA made them ***stronger*** against an attack they knew but the public didn't
  - The NSA-defined elliptic curves P-256 and P-384
    - I trust them because they are in Suite-B/CNSA so the NSA uses them for TS communication:
      A backdoor here would be absolutely unacceptable...
      but ***only because I actually believe the NSA wouldn't go and try to shoot itself in the head!***

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

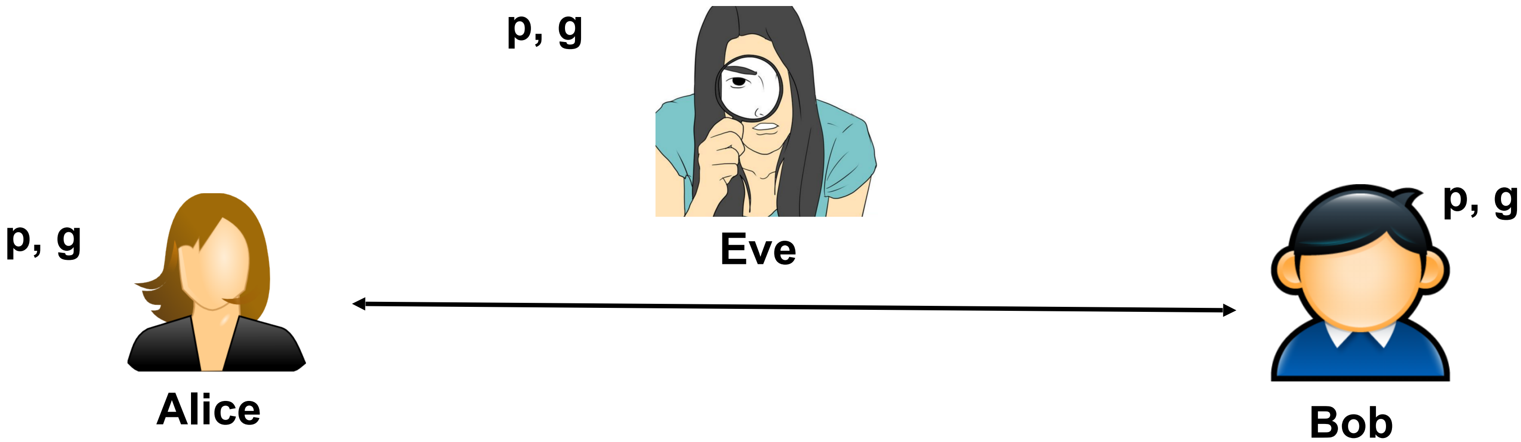
Our Roadmap...

- 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
- 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

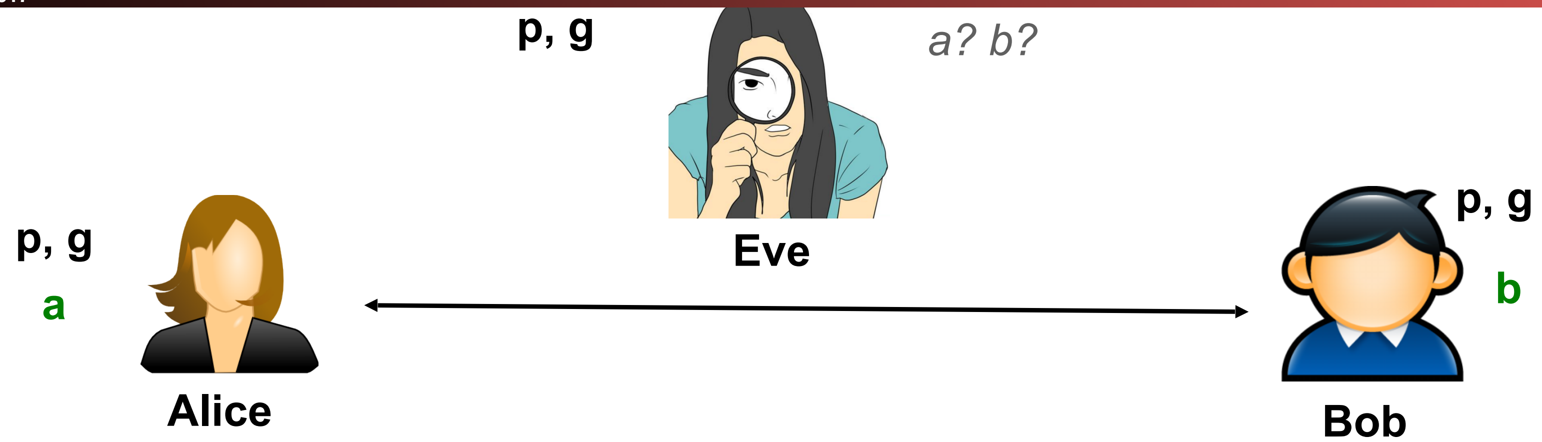
- What if instead they can somehow generate a random key when needed?
- Seems impossible in the presence of Eve observing all of their communication …
  - How can they exchange a key without her learning it?
- But: actually is possible using public-key technology
  - Requires that Alice & Bob know that 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

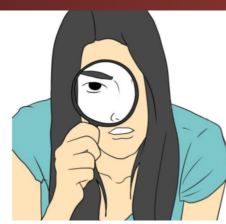
3. Bob picks random secret ‘**b**’: 1 < b < p-1

DHE

p, g

A

B



Eve

a? b?

p, g

a

Alice

p, g

b

Bob

$A = g^a \bmod p$

4. Alice sends $A = g^a \bmod p$ to Bob
5. Bob sends $B = g^b \bmod p$ to Alice

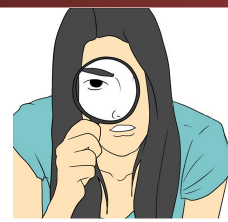
$g^b \bmod p = B$

DHE

p, g

A

B



a? b?

Eve

p, g

a

K

Alice

p, g

b

K

Bob

A = g^a mod p

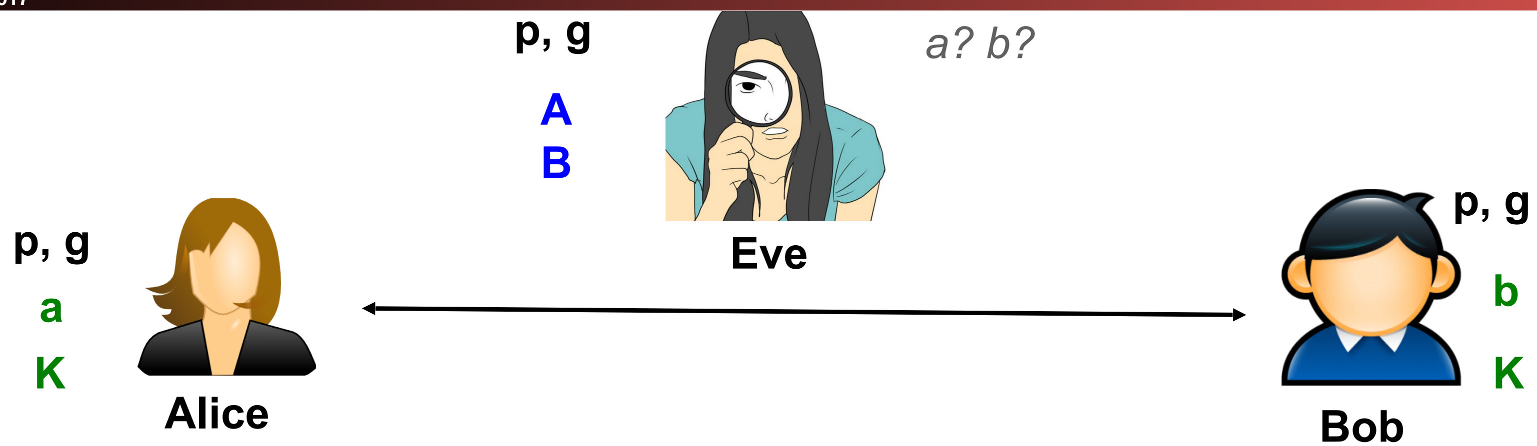
B

A

g^b mod p = B

6. Alice knows {a, A, B}, computes K = B^a mod p = $(g^b)^a$ = g^{ba} mod p
7. Bob knows {b, A, B}, computes K = A^b mod p = $(g^a)^b$ = g^{ab} mod 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.

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

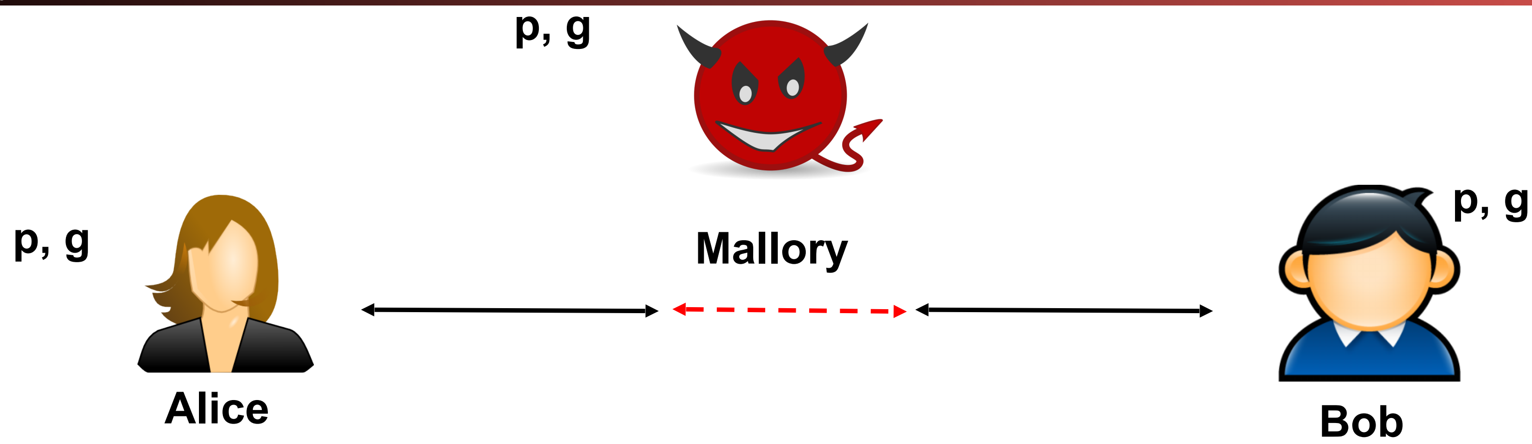
This is Ephemeral Diffie/Hellman

- **$K = g^{ab}$ mod 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**.

But Its Not That Simple

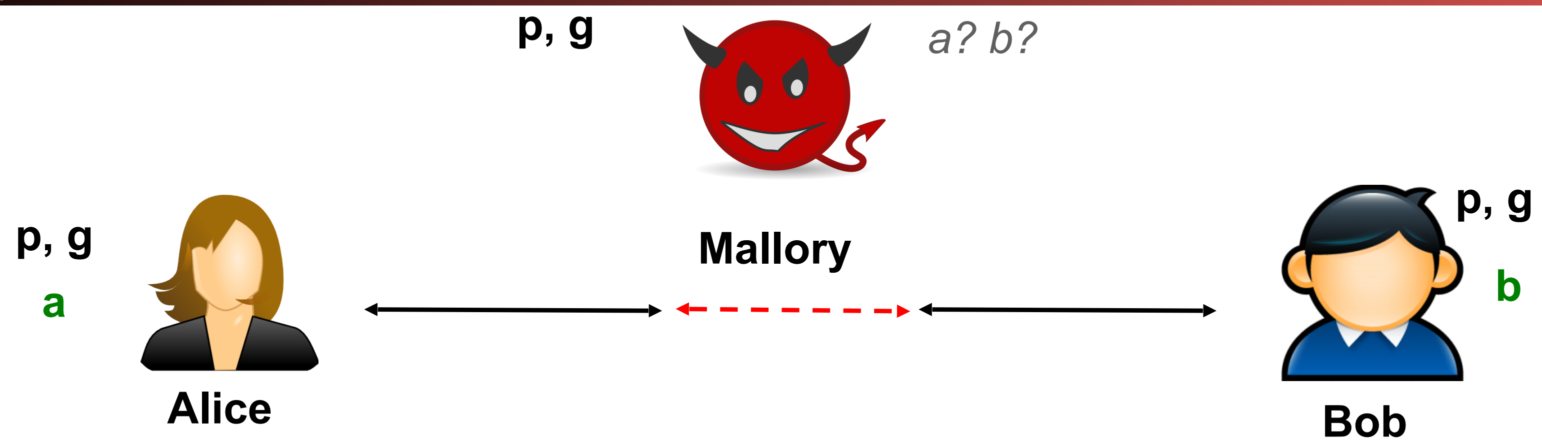
- What if Alice and Bob aren't facing a passive eavesdropper
  - But instead are facing Mallory, an ***active*** Man-in-the-Middle
- Mallory has the ability to change messages:
  - Can remove messages and add his own
- Lets see... Do you think DHE will still work as-is?

Attacking DHE as a MitM



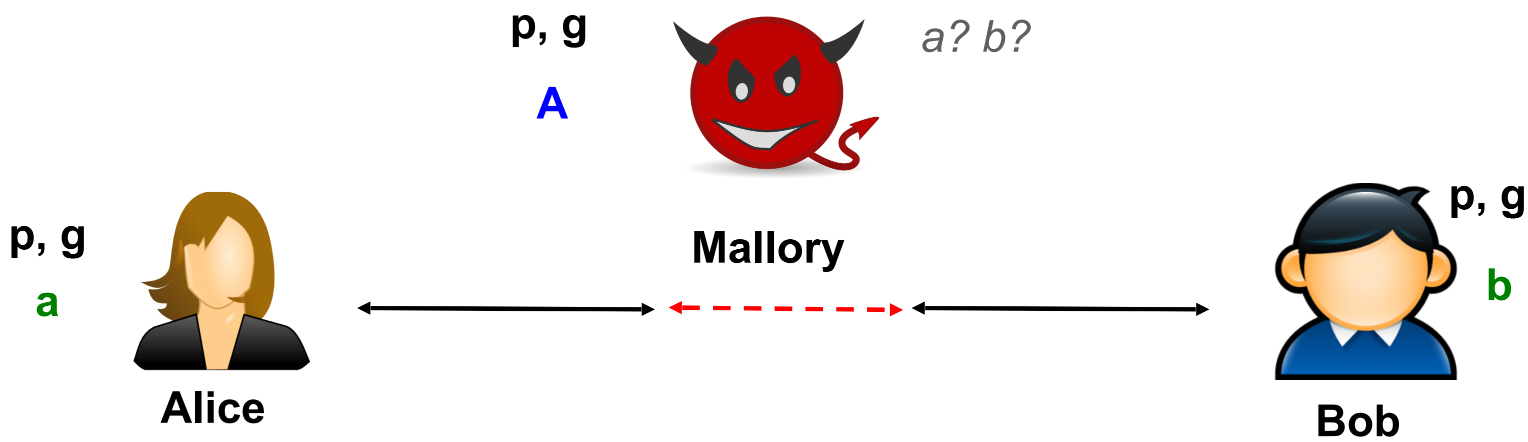
What happens if instead of Eve watching, Alice & Bob face the threat of a hidden Mallory (MITM)?

The MitM Key Exchange



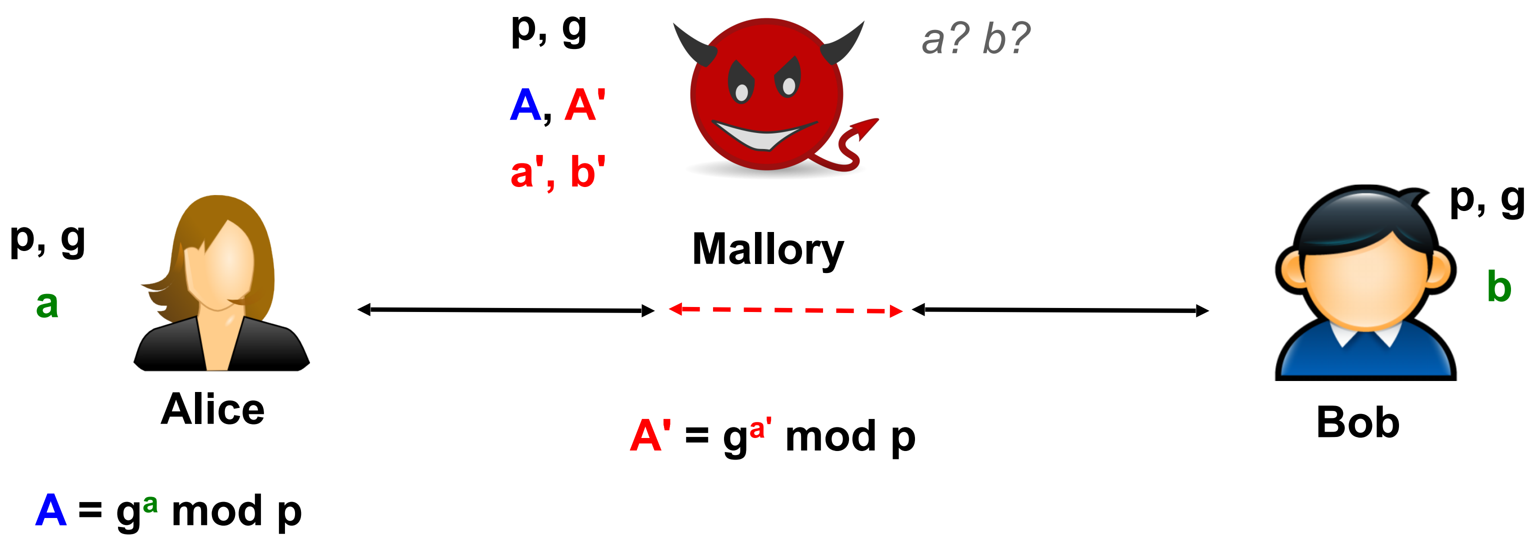
2. Alice picks random secret '**a**': 1 < a < p-1

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



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

4. Alice sends $A = g^a$ mod 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

a

Alice

p, g

A, A'

a', b'

a? b?

Mallory

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

p, g

b

Bob

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

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

p, g

b

Bob

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

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

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

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'

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

A'

$g^b \bmod p = B$

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

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 **d =** $\mathbf{e^{-1}}$ **mod φ(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: $D(C, K_D) = (M^{e \cdot d}) \bmod n$
- Now recall that **d** is the multiplicative inverse of **e**, modulo **φ(n)**, and thus:

  $e \cdot d = 1 \bmod \varphi(n)$ (by definition)

  $e \cdot d - 1 = k \cdot \varphi(n)$ for some **k**
- Therefore **$D(C, K_D) = M^{e \cdot d} \bmod n = (M^{e \cdot d-1}) \cdot M \bmod n$**

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

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

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

  $= M \bmod n = M$

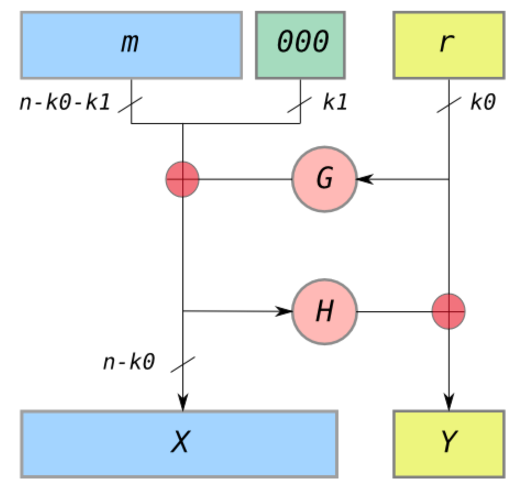
(believed) Eve can recover M from C *iff* Eve can factor n=p·q

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...] ⊕ G(r)**
    - **G** and **H** are hash functions (EG SHA-256)
      k_0 = # of bits of randomness, **len(m) +** k_1 **+** k_0 **= n**
  - Then replaces r with **Y = H(G(r) ⊕ [m,0...]) ⊕ 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...

$$
\begin{array}{l}
\boldsymbol{x} = C \\
\textbf{for } j = 1 \texttt{ to } n \\
\quad x = \texttt{mod}(x^2, N) \\
\quad \textbf{if } d_j == 1 \textbf{ then} \\
\quad\quad x = \texttt{mod}(xC, N) \\
\quad \textbf{end if} \\
\textbf{next } j \\
\textbf{return } x
\end{array}
$$

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
- 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 $\mathbf{m_1}$...

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

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 sa and sb
  - 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