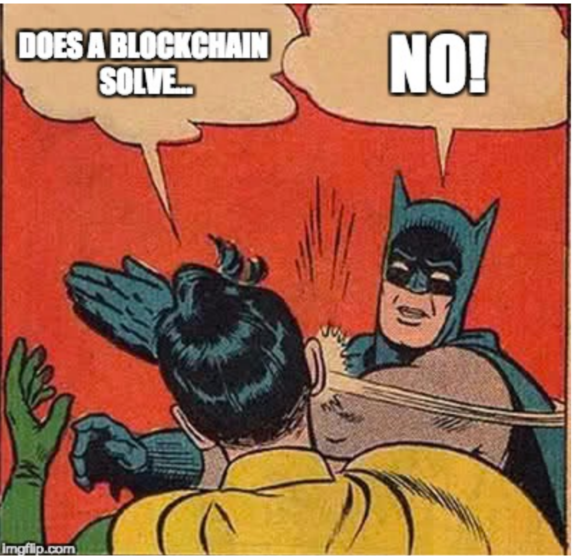
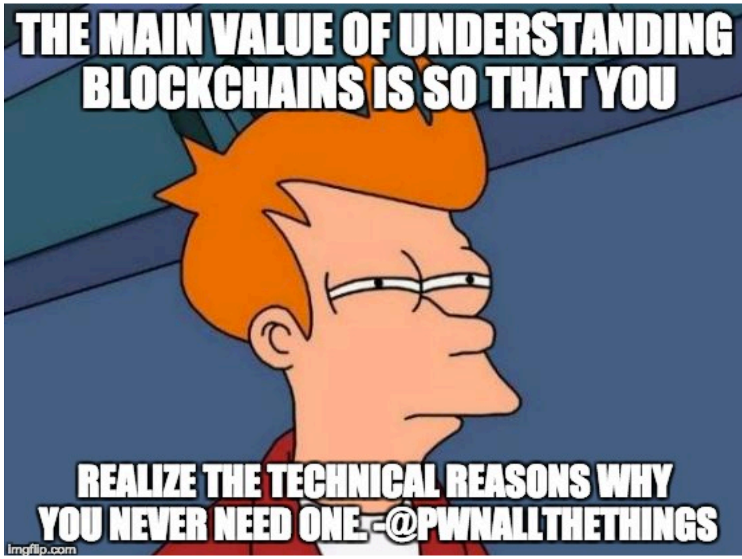
Crypto 3


DOES A BLOCKCHAIN SOLVE...
NO!
imgflip.com


NICK'S LAW OF BLOCKCHAIN: BLOCKCHAIN SOLVES ONE PROBLEM
IF ANYONE SAYS "BLOCKCHAIN CAN SOLVE X" THEY KNOW NOTHING ABOUT X AND SHOULD BE IGNORED
imgflip.com


THE MAIN VALUE OF UNDERSTANDING BLOCKCHAINS IS SO THAT YOU
REALIZE THE TECHNICAL REASONS WHY YOU NEVER NEED ONE. -@PWNALLTHETHINGS
imgflip.com

Administrivia...

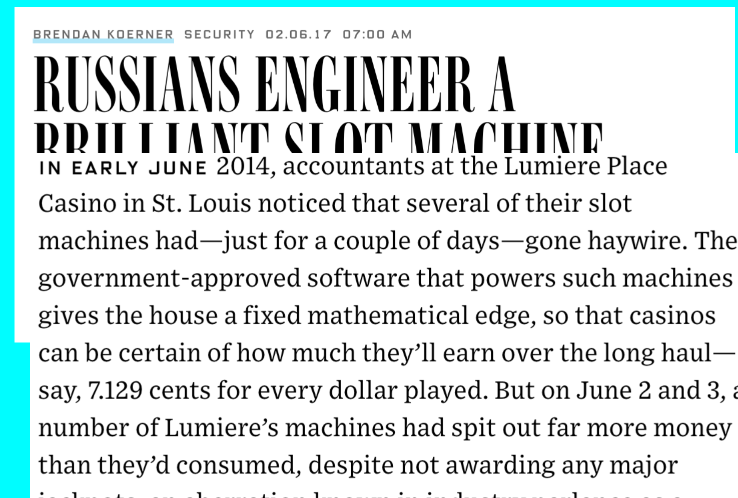
- Project 1 question 3 starter code has a bug
  - See Piazza
- HW2 due Friday
- Project 1 due Friday the 19th

A Lot Of 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



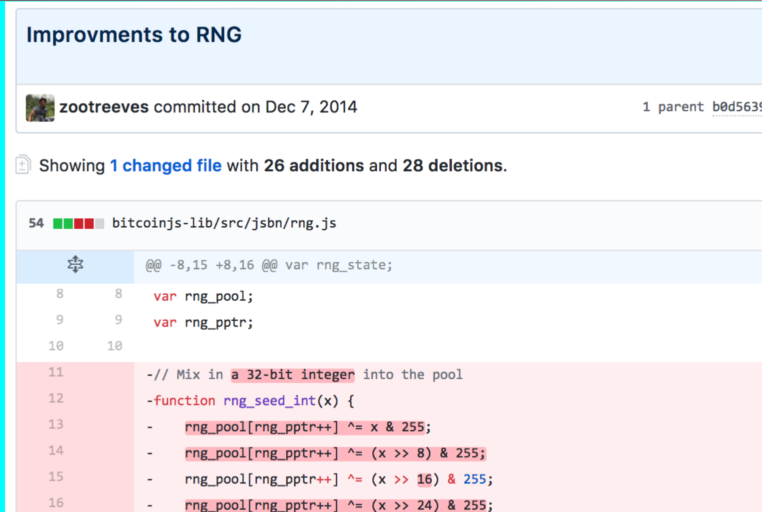
BRENDAN KOERNER SECURITY 02.06.17 07:00 AM

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: It is just one source of the randomness

Combining Entropy

- Many physical entropy sources are biased
  - Some have significant biases: e.g. a coin that flips "heads" 90% of the time!
  - Some aren't very good: e.g. keystroke timing at a microsecond granularity
- 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 fair 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 **2**^{s} distinct possible sequences.
- A cryptographically strong pRNG "looks" truly random to an attacker
  - Attacker ***cannot distinguish*** it from a random sequence:
    If the attacker can tell a sufficiently long bitstream was generated by the pRNG instead of a truly random source it isn't a good pRNG

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
  - Rollback resistance isn't specifically required in a pRNG...
    But it should be

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} = Random Session key
    T_0 = Nonce/IV
- 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

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
  - So as long as the security proof for HMAC is correct, the security proof for HMAC_DRBG is correct!

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 with 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 (keep **state.k** and **state.v**)
- 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)
}
```

Generating ***true*** random numbers

- Modern CPUs have true random number generators
  - Sample a noisy circuit at a low rate or similar tricks
- These sources are biased...
  - They are also ***slow***
- So use this as an entropy source to feed a pRNG on the chip
  - Now you can get random numbers quickly
- Very fast
- Vulnerable to tampering!
  - You can't actually test that the pRNG circuit is 100% correct without adding paths that could potentially sabotage the pRNG circuit
  - Sabotage that can reduce effective entropy to 32b are possible

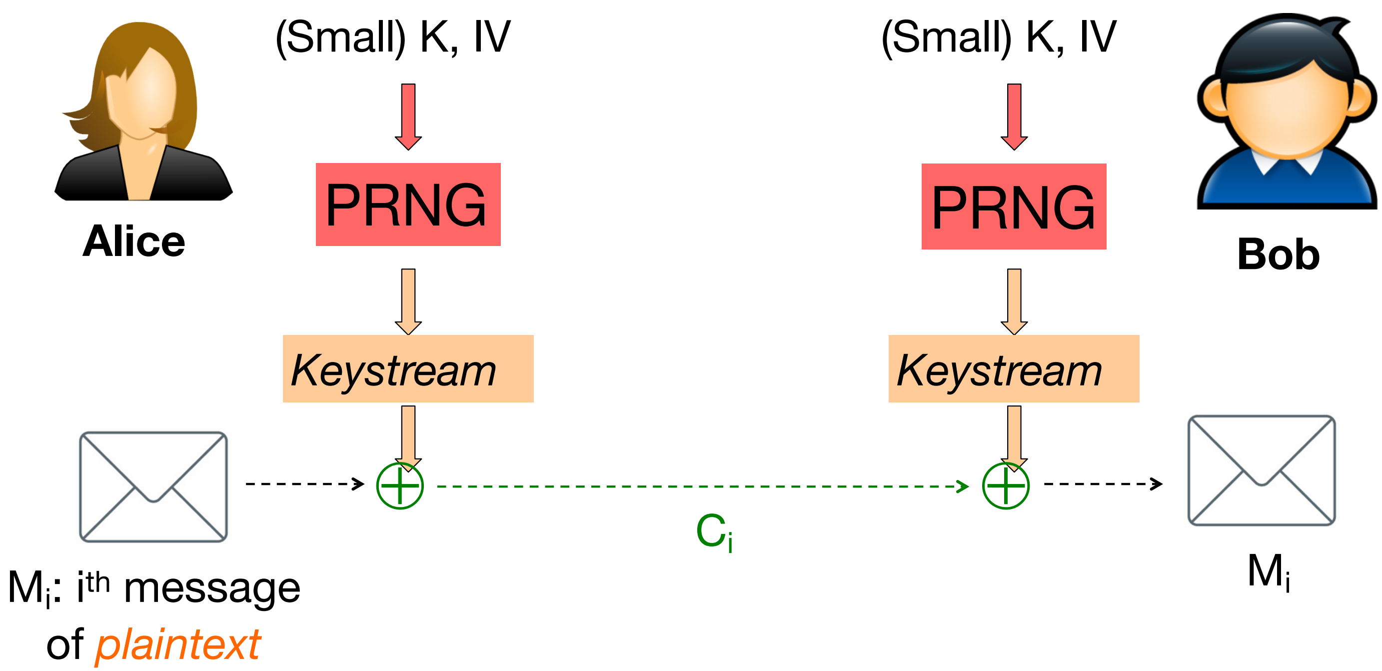
Stream ciphers

- 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 $2^{n/2}$. 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 use CTR mode in the first place
- Also very minor information leakage:
  - If $\mathbf{C_i = C_j}$, for $\mathbf{i \mathrel{!=} j}$, it follows that $\mathbf{M_i \mathrel{!=} M_j}$

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 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 What To Use?

- AES-128-CFB or AES-256-CFB:
  - Robust to screwups encryption
  - Alternately, AES-128-GCM (Galios Counter Mode):
    An AEAD mode, but is ***NOT resistant to screwups***
- SHA-2 or SHA-3 family (256b, 384b, or 512b):
  - Robust cryptographic hashes, SHA-1 and MD5 are broken
- HMAC-SHA256 or HMAC-SHA3:
  - Different function than the encryption:
    Prevents screwups on using the same key & is a hash if not using an AEAD mode
  - ***Always Encrypt Then MAC***!
- HMAC-SHA256-DRBG or HMAC-SHA3-DRBG:
  - The best pRNG available
  - Seed using ***both*** the processor random number generator AND other entropy sources!
    - Don't use the processor RNG bare when building a software cryptosystem:
      Those are potentially sabotage able and use designs without rollback resistance.

Public Key...

- All our previous primitives required a "miracle":
  - We somehow have to have Alice and Bob get a shared ***k***.
- Enter Public Key cryptography: the miracle of modern cryptography
  - How starting Friday, but ***what*** today
- Three primitives:
  - Public Key Agreement
  - Public Key Encryption
  - Public Key Signatures
- Based on some families of magic math...
  - For us, we will use some group-theory based primitives

Public Key Agreement

- Alice and Bob have a channel...
  - There may be an eavesdropper ***but not a manipulator***
- The goal: Alice & Bob agree on a ***random*** value
  - This will be ***k*** for all subsequent communication
- When done, the key is thrown away
  - Designed to prevent an attacker who later recovers Alice or Bob's long lived secrets from finding ***k***.

Public Key Encryption

- Alice has ***two*** keys:
  - K_{pub}: Her public key, anyone can know
  - K_{priv}: Her private key, a deep dark secret
- Anyone has access to Alice's public key
- For anyone to send a message to Alice:
  - Create a random session key ***k***
    - Used to encrypt the rest of the message
  - Encrypt ***k*** using Alice's K_{pub}.
- Only Alice can ***decrypt*** the message
  - The decryption function only works with K_{priv}!

Public Key Signatures

- Once again, Alice has ***two*** keys:
  - K_{pub}: Her public key, anyone can know
  - K_{priv}: Her private key, a deep dark secret
- She can sign a message
  - Calculate ***H(M)***
  - $S(K_{priv}, H(M))$: Sign ***H(M)*** with K_{priv}.
- Anyone can now verify
  - Recalculate ***H(M)***
  - $V(K_{pub}, S(K_{priv}, H(M), H(M))$: Verify that the signature was created with K_{priv}

Things To Remember...

- Public key is ***slow!***
  - Orders of magnitude slower than symmetric key
- Public key is based on delicate magic math
  - Discrete log in a group is the most common
  - RSA
  - Some new "post-quantum" magic...
- Some systems in particular are easy to get wrong
  - We will get to some of the epic crypto-fails later

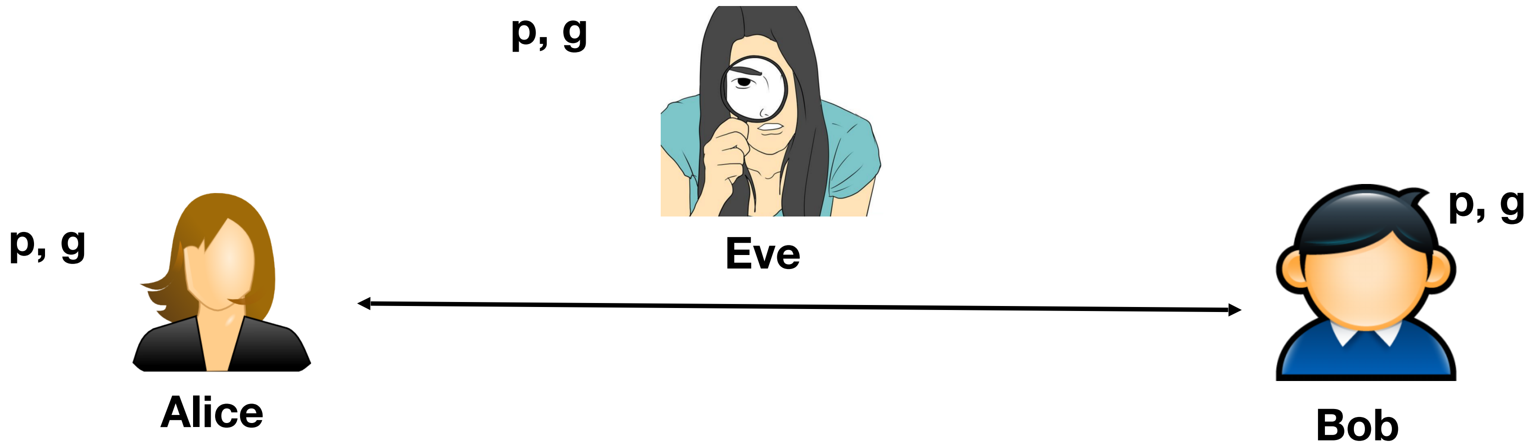
Our Roadmap For Public Key...

- 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

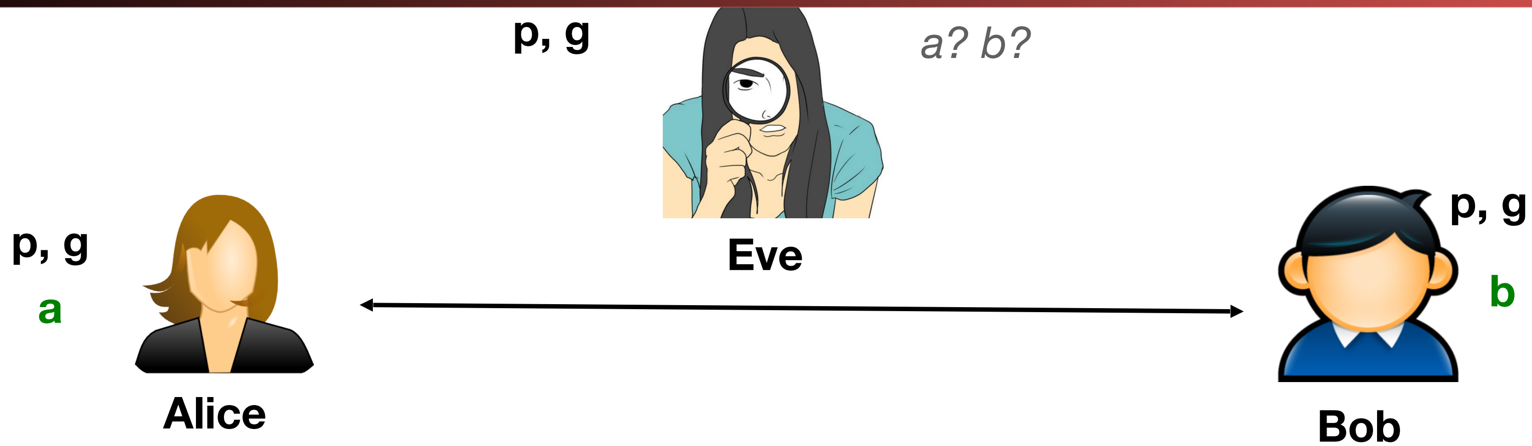
- What if instead they can somehow generate a random key when needed?
- Seems impossible in the presence of Eve observing all of their communication …
  - How can they exchange a key without her learning it?
- But: actually is possible using public-key technology
  - Requires that Alice & Bob know that 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**: $1 < g < p-1$

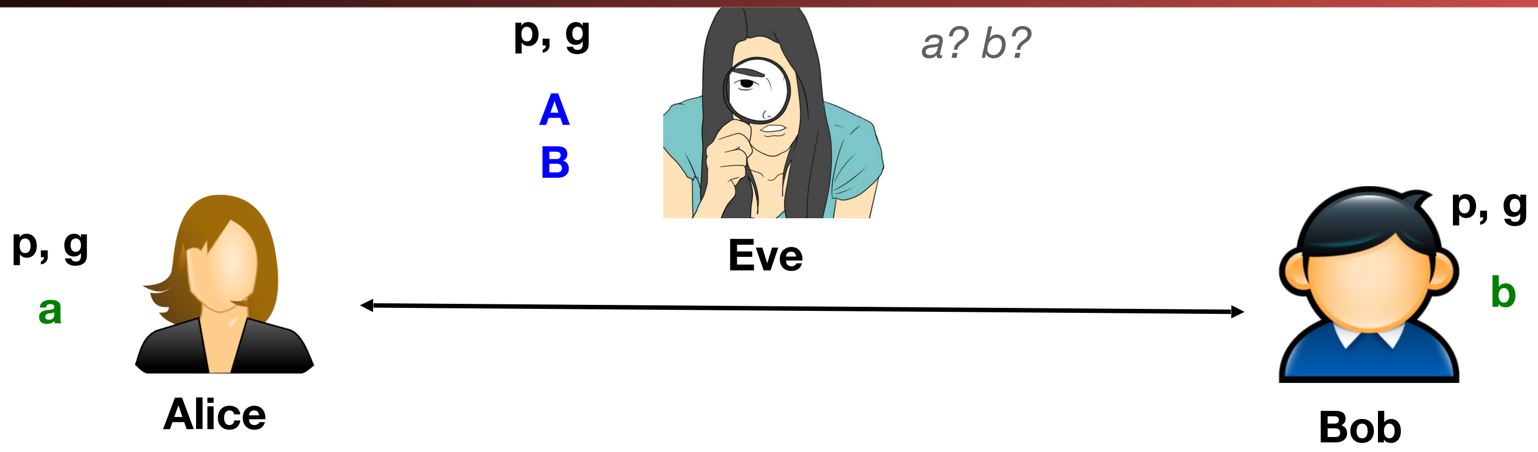
DHE



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

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

DHE

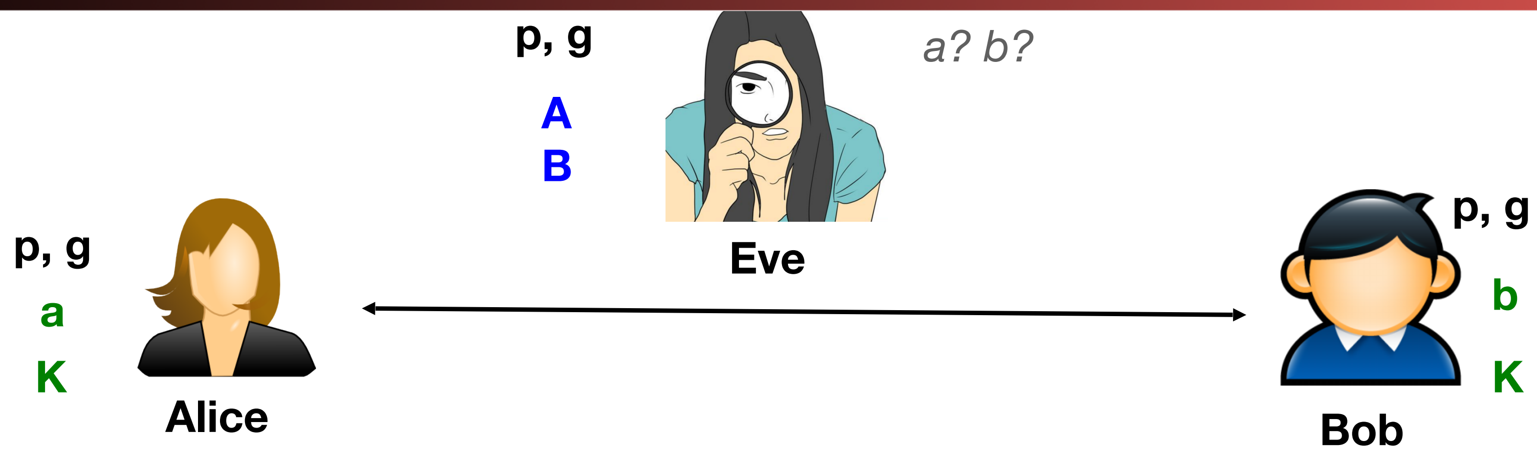


$A = g^a \bmod p$

$g^b \bmod p = B$

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

DHE



$A = g^a \bmod p$

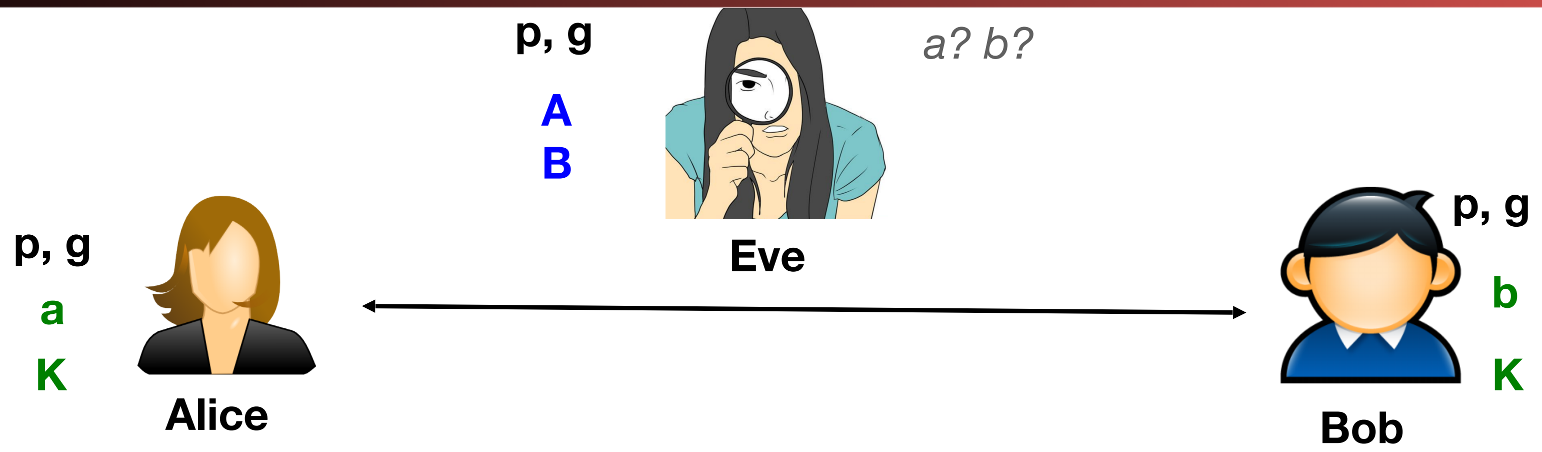
B

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 $A \cdot B = g^a \cdot g^b$ mod p = g^{a+b} mod p.
But computing g^{ab} requires ability to take *discrete logarithms* mod p.
Discrete log over the group defined by p and g ***presumed*** to be hard

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**.

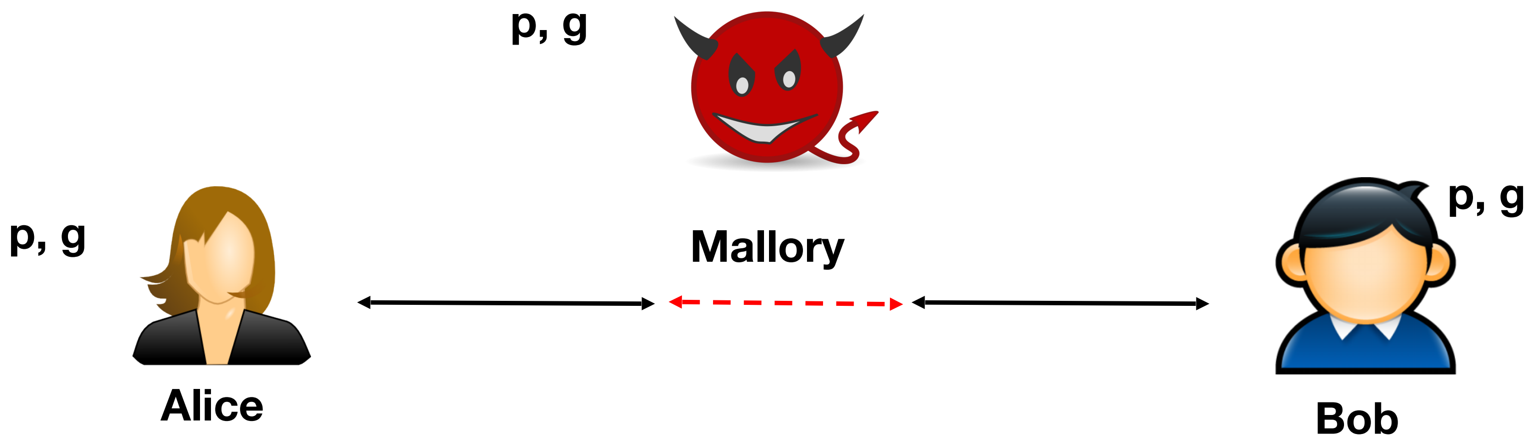
Diffie Hellman is part of more generic problem

- 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

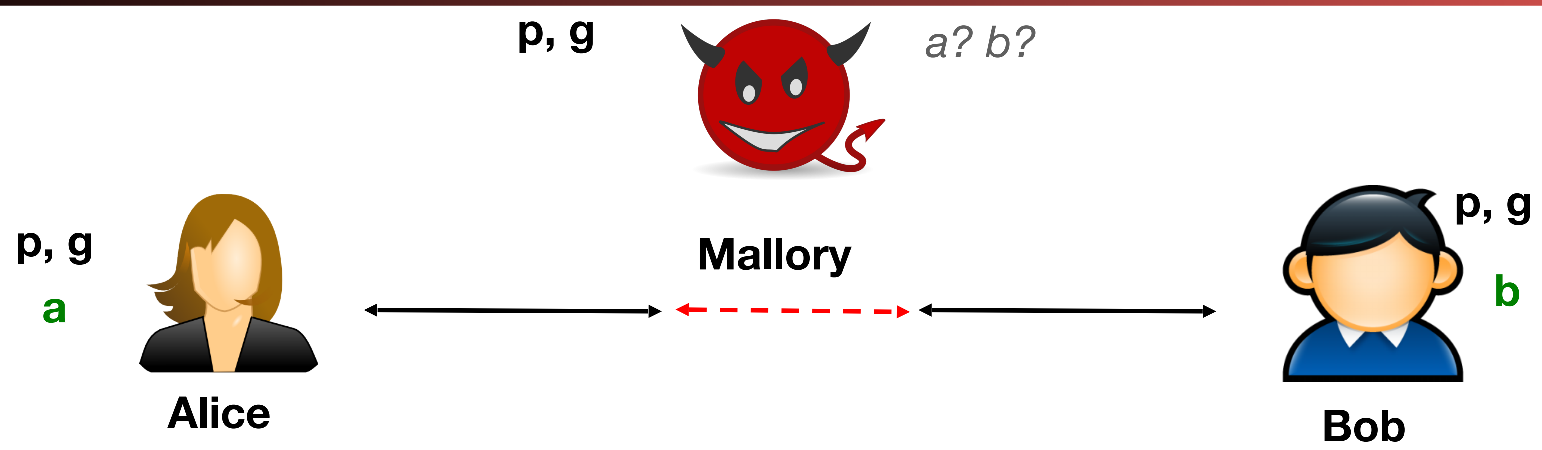
- What if Alice and Bob aren't facing a passive eavesdropper
  - But instead are facing Mallory, an ***active*** Man-in-the-Middle
- Mallory has the ability to change messages:
  - Can remove messages and add his own
- Lets see... Do you think DHE will still work as-is?

Attacking DHE as a MitM



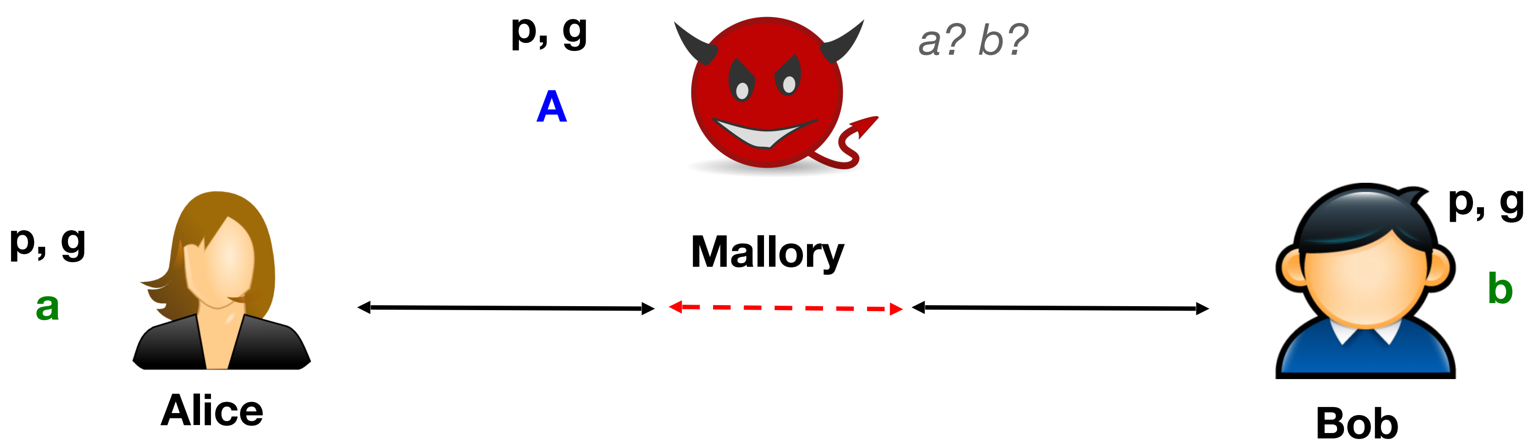
What happens if instead of Eve watching, Alice & Bob face the threat of a hidden Mallory (MITM)?

The MitM Key Exchange



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

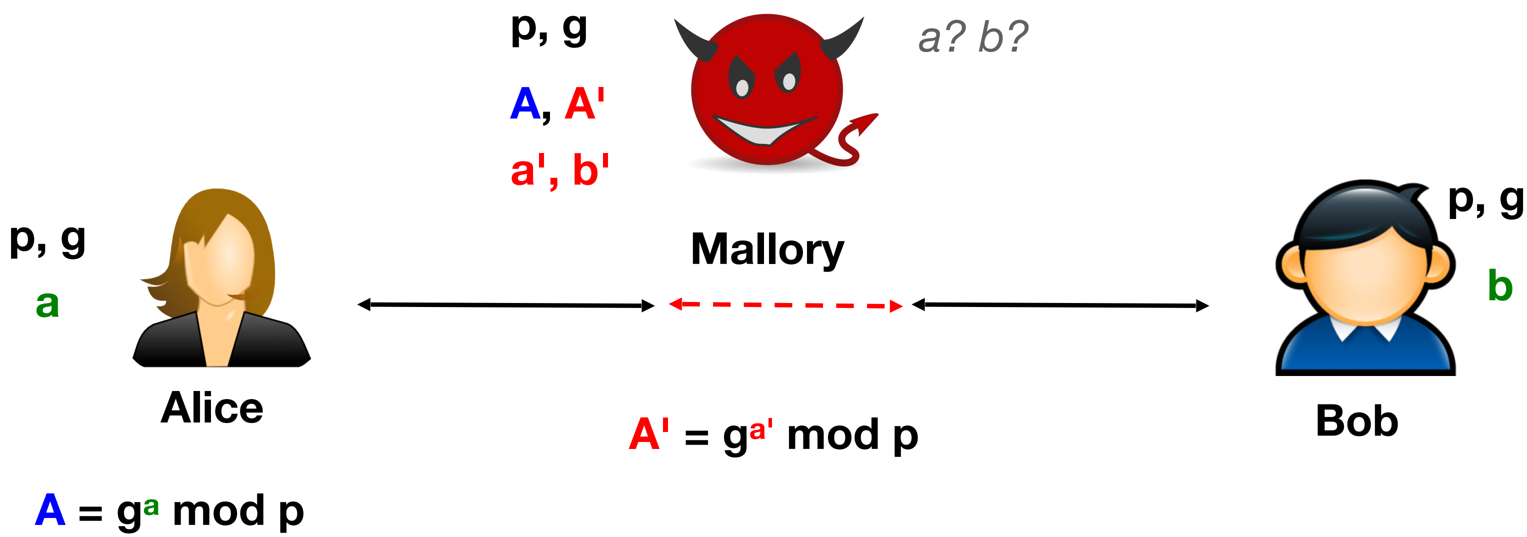
3. Bob picks random secret ‘**b**’: 1 < b < p-1



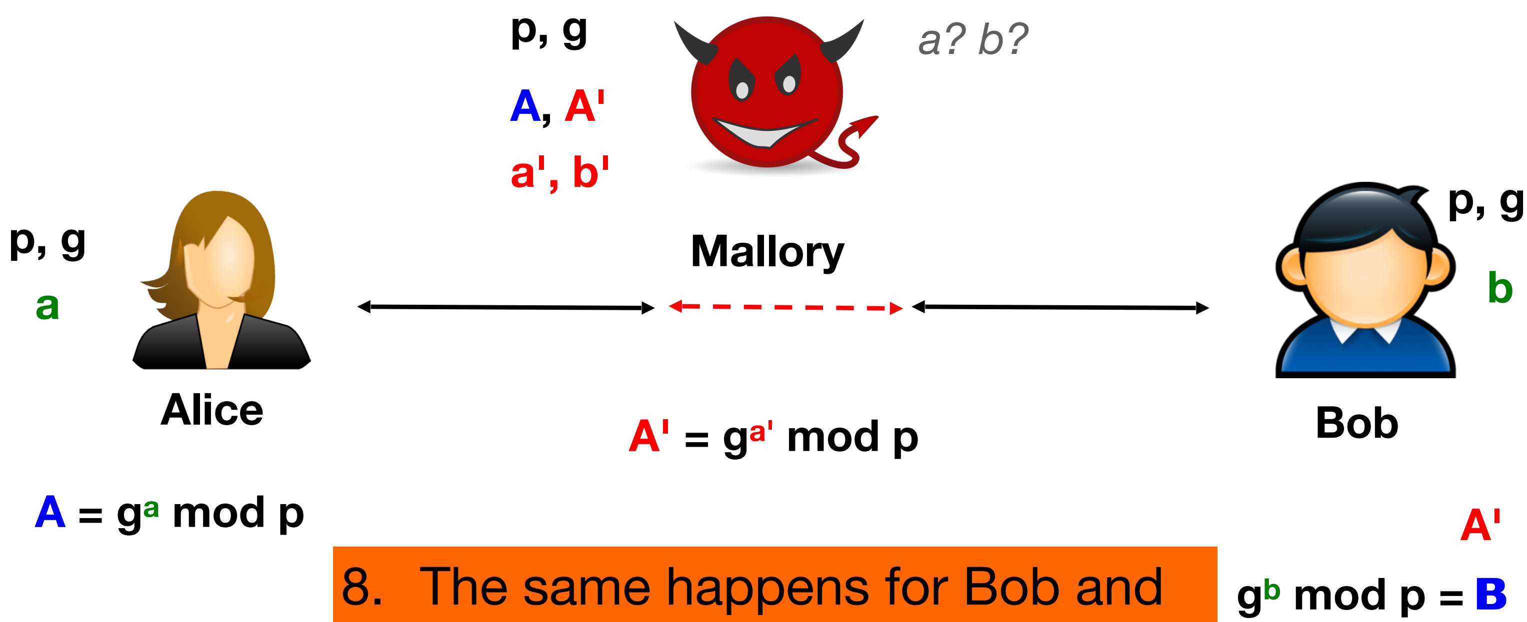
A = g^a mod p

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

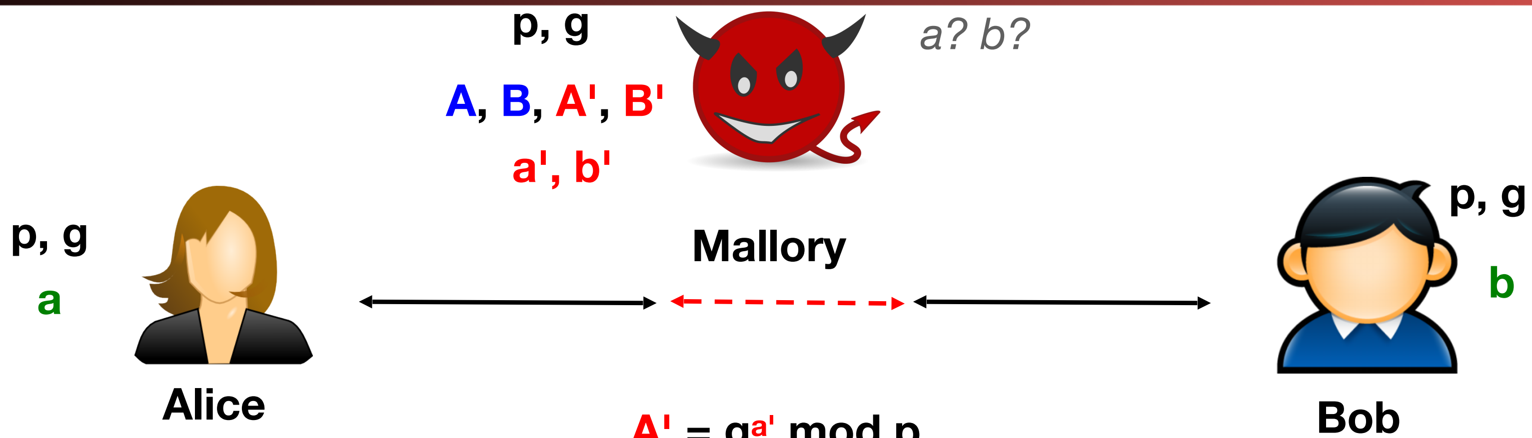
5. Mallory prevents Bob from receiving A



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



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



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

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

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

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

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

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

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

B'

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

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

A'

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

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

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

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'**.