Digital Signatures, Certificates and Password Hashing

CS 161 Spring 2024 - Lecture 10

Last Time: Public-Key Encryption

- Public-key cryptography: Two keys: private and public
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - ElGamal: Based on Diffie-Hellman
    - The public key is g^b, and C_1 is g^r.
    - Not semantically secure on its own but can be made so
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - Not semantically secure on its own but can be made so
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message

Digital Signatures

Textbook Chapter 12

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

Digital Signatures

- Asymmetric cryptography is good because we don’t need to share a secret key
- Digital signatures are the asymmetric way of providing integrity/authenticity to data
- Assume that Alice and Bob can communicate public keys without Mallory changing them
  - We will see how to fix this limitation later using certificates

Digital Signatures

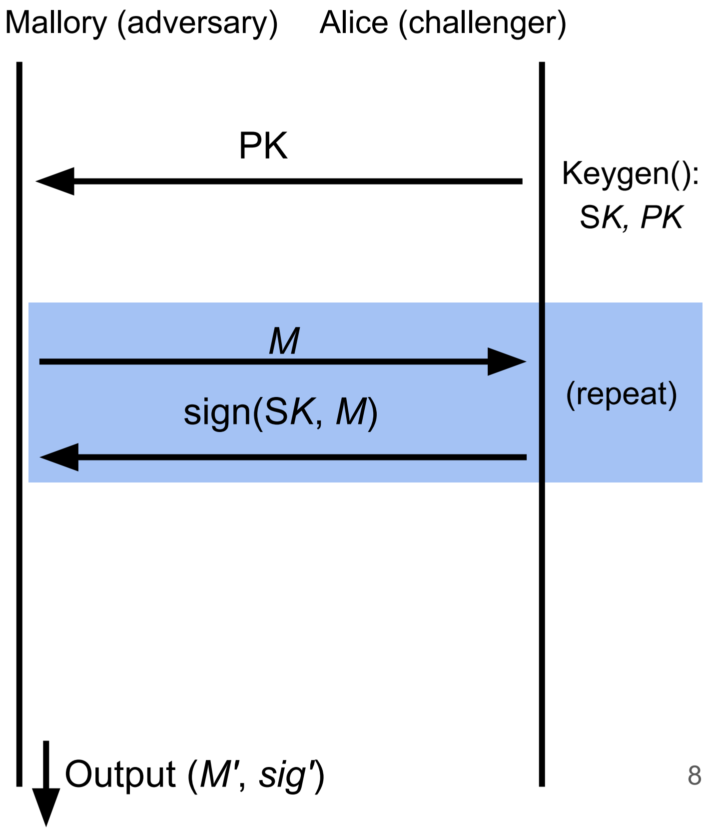
- Only the owner of the private key can sign messages with the private key
- Everybody can verify the signature with the public key

Digital Signatures: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify the signature *sig* on message *M* using the verify key *PK* and output 1 if valid and 0 if invalid
- Properties
  - **Correctness**: Verification should be successful for a signature generated over any message
    - Verify(*PK*, *M*, Sign(*SK*, *M*)) = 1 for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Signing/verifying should be fast
  - **Security**: EU-CPA, same as for MACs

Defining Integrity: EU-CPA

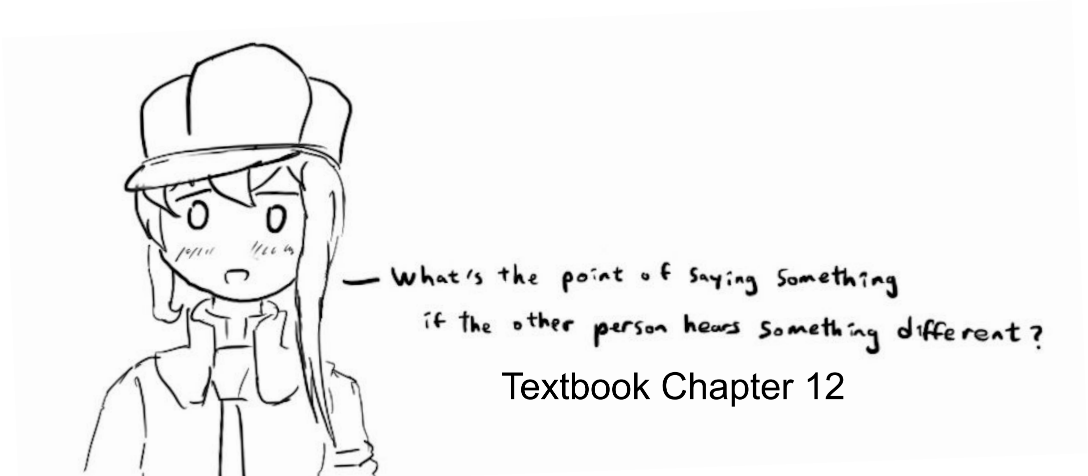
1. Mallory may send messages to Alice and receive their tags
2. Eventually, Mallory creates a message-signature pair (*M'*, *sig'*)
   - *M'* cannot be a message that Mallory requested earlier
   - If *sig'* is verifies with PK for *M'*, then Mallory wins. Otherwise, she loses.
3. A scheme is EU-CPA secure if for *all* polynomial time adversaries, the probability of winning is 0 or negligible



Digital Signatures in Practice

- If you want to sign message *M*:
  - First hash *M*
  - Then sign H(*M*)
- Why do digital signatures use a hash?
  - Allows signing arbitrarily long messages
- Digital signatures provide integrity *and authenticity* for *M*
  - The digital signature acts as proof that the private key holder signed H(*M*), so you know that *M* is authentically endorsed by the private key holder

RSA Signatures



Textbook Chapter 12

RSA Signatures

- Recall RSA encryption: $M^{ed} \equiv M \bmod N$
  - There is nothing special about using *e* first or using *d* first!
  - If we encrypt using *d*, then anyone can "decrypt" using *e*
    - Given *x* and $x^d \bmod N$, can't recover *d* because of discrete-log problem, so *d* is safe

RSA Signatures: Definition

- KeyGen():
  - Same as RSA encryption:
    - **Public key**: N and e
    - **Private key:** d
- Sign(d, M):
  - Compute $H(M)^d \bmod N$
- Verify(e, N, M, sig)
  - Verify that $H(M) \equiv sig^e \bmod N$

Correctness: $sig^e \bmod N \equiv H(M)^{de} \bmod N \equiv H(M) \bmod N$

(because recall $x^{de} \bmod N \equiv x \bmod N$ for all x)

Summary: Public-Key Cryptography

- Public-key cryptography: Two keys, private and public
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - ElGamal: Based on Diffie-Hellman
    - The public key is g^b, and C_1 is g^r.
    - Not IND-CPA secure on its own
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - Not IND-CPA secure on its own
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- Digital signatures: Integrity and authenticity for asymmetric schemes
  - RSA: Same as RSA encryption, but sign the hash with the *private* key

How do we distribute public keys securely?

Certificates

Textbook Chapter 13

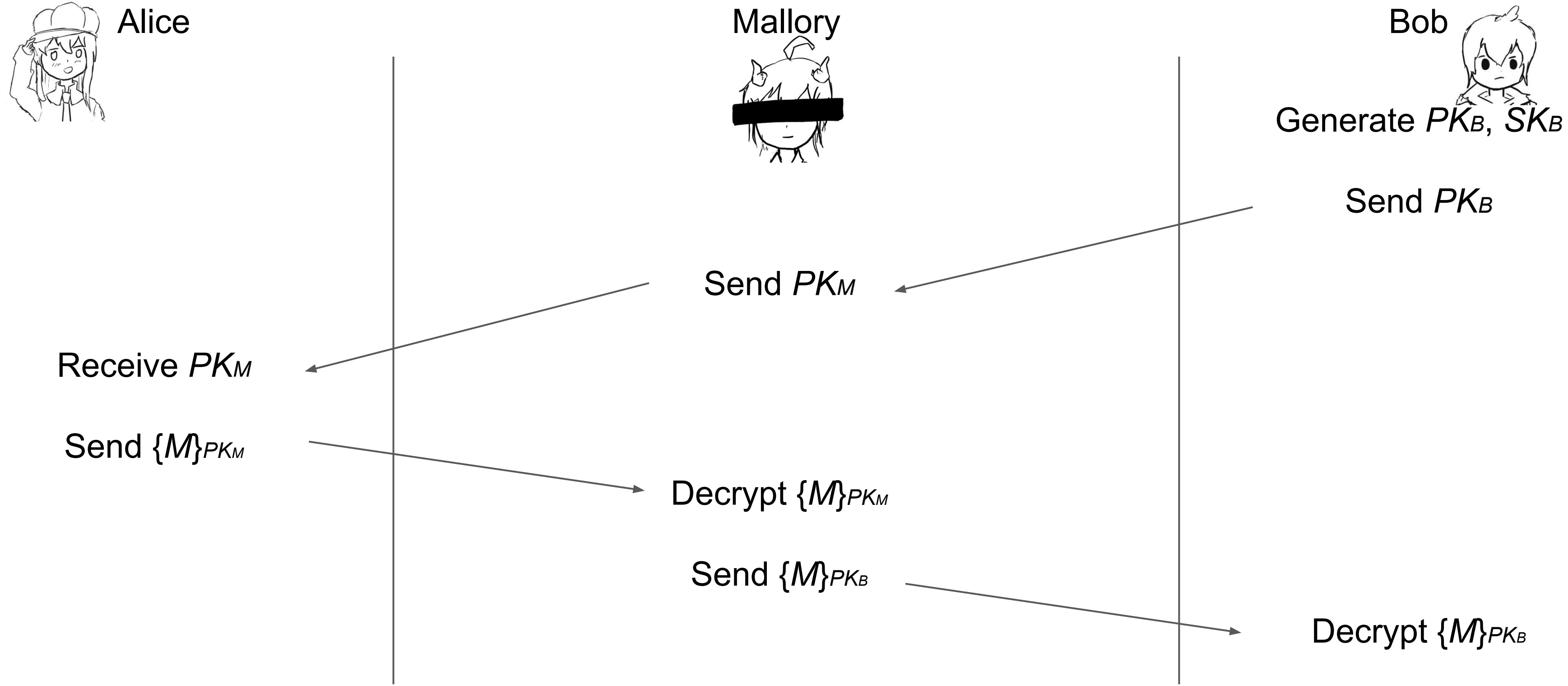
Review: Public-Key Cryptography

- Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key
- What’s the catch?

Problem: Distributing Public Keys

- Public-key cryptography alone is not secure against man-in-the-middle attacks
- Scenario
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- What can Mallory do?
  - Replace Bob's public key with Mallory’s public key
  - Now Alice has encrypted the message with Mallory’s public key, and Mallory can read it!

Problem: Distributing Public Keys



Problem: Distributing Public Keys

- Idea: Sign Bob's public key to prevent tampering
- Problem
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob's public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- You cannot gain trust if you trust nothing. You need a root of trust!
  - **Trust anchor**: Someone that we implicitly trust
  - From our trust anchor, we can begin to trust others

Trust-on-First-Use

- **Trust-on-first-use**: The first time you communicate, trust the public key that is used and warn the user if it changes in the future
  - Used in SSH, Whatsapp and a couple other protocols
  - Idea: Attacks aren’t frequent, so assume that you aren’t being attacked the first time you communicate
  - Also known as “**Leap of Faith**”

Certificates

- **Certificate**: A signed endorsement of someone's public key
  - A certificate contains at least two things: The **identity** of the person, and the **key**
- Abbreviated notation
  - Encryption under a public key *PK*: $\{\text{"Message"}\}_{PK}$
  - Signing with a private key *SK*: $\{\text{"Message"}\}_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't check the signature by itself!
- Scenario: Alice wants Bob's public key. Alice trusts EvanBot (PK_E, SK_E)
  - EvanBot is our trust anchor
  - If we trust PK_E, a certificate we would trust is $\{\text{"Bob's public key is } PK_B\text{"}\}_{SK_E^{-1}}$

Attempt #1: The Trusted Directory

- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
  - The TD has a public/private keypair PK_{TD}, SK_{TD}
  - The directory publishes PK_{TD} so that everyone knows it (baked into computers, phones, OS, etc.)
  - When you request Bob's public key, the directory sends a certificate for Bob's public key
    - {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - If you trust the directory, then now you trust every public key from the directory
- What do we have to trust?
  - We have received TD's key correctly
  - TD won't sign a key without verifying the identity of the owner

Attempt #1: The Trusted Directory

- Let’s say that Michael Drake (MD, President of UC) runs the TD
  - We want Peyrin Kao’s public key: Ask MD
  - We want David Wagner’s public key: Ask MD
  - We want Raluca Ada Popa’s public key: Ask MD
  - MD also needs to make sure that his private key isn’t stolen!
- Problems: Scalability
  - One directory won’t have enough compute power to serve the entire world
- Problem: Single point of failure
  - If the directory fails, *services depending on this become unavailable*
  - If the directory is compromised, you can’t trust anyone
  - If the directory is compromised, it is difficult to recover

Any ideas?

Certificate Authorities

- Addressing scalability: Hierarchical trust
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {"Carol Christ's public key is PK_{CC}, and I trust her to sign for UCB"}$_{SK_{MD}^{-1}}$
    - {"Dave Wagner's public key is PK_{DW}, and I trust him to sign for the CS department"}$_{SK_{CC}^{-1}}$
    - {"Raluca Ada Popa's public key is PK_{RAP} (but I don't trust her to sign for anyone else)"}$_{SK_{DW}^{-1}}$
  - MD is still the root of trust (**root certificate authority**, or **root CA**)
  - CC and DW receive delegated trust (**intermediate CAs**)
  - RAP's identity can be trusted
- Addressing scalability: Multiple trust anchors
  - There are ~150 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices
  - Each delegation step can restrict the scope of a certificate's validity
  - Creating the certificates is an *offline* task: The certificate is created once in advance, and then served to users when requested

Revocation

- What happens if a certificate authority messes up and issues a bad certificate?
  - Example: {"Bob's public key is PK_M"}$_{SK_{CA}^{-1}}$
  - Example: Verisign (a certificate authority) accidentally issued a certificate saying that an average Internet user's public key belonged to Microsoft
  - Other users will trust the wrong PK, e.g. may think that Microsoft signed some binary but instead some malicious user signed malware

How can we revoke certificates?

Revocation: Expiration Dates

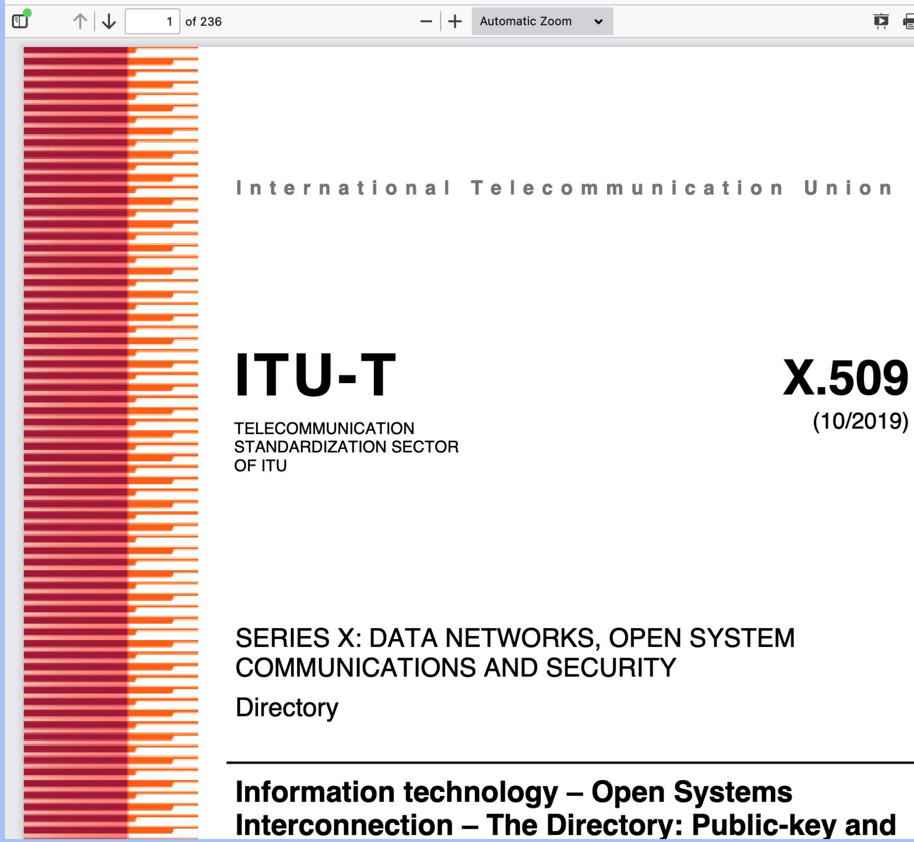
- Approach #1: Each certificate has an expiration date
  - When the certificate expires, request a new certificate from the certificate authority
  - The bad certificate will eventually become invalid once it expires
- Benefits
  - Mitigates damage: Eventually, the bad certificate will become harmless
- Drawbacks
  - Adds management burden: Everybody has to renew their certificates frequently
  - If someone forgets to renew a certificate, their website might stop working
- Tradeoff: How often should certificates be renewed?
  - Frequent renewal: More secure, less usable
  - Infrequent renewal: Less secure, more usable
- LetsEncrypt (a certificate authority) chose very frequent renewal
  - It turns out frequent renewal is more usable:
    It forces automated renewal instead of a once-every 3 year task that gets forgotten!

Revocation: Announcing Revoked Certificates

- Approach #2: Periodically release a list of invalidated certificates
  - Users must periodically download a Certification Revocation List (CRL)
- How do we authenticate the list?
  - The certificate authority signs the list!
    - {“The certificate with serial number 0xdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
- Drawbacks
  - Lists can get large
    - Mitigated by shorter expiration dates (don’t have to list them once they expire)
  - Until a user downloads a list, they won’t know which certificates are revoked
- What happens if the certificate authority is unavailable?
  - Fail-safe default: Assume all certificates are invalid? Now we can’t trust anybody!
    - Possible attack: Attacker forces the CA to be unavailable (denial of service attack)
  - Use old list: Potentially dangerous if the old list is missing newly revoked certificates

Certificates: Complexity

- Certificate protocols can get very complicated
  - Example: X.509 is incredibly complicated (a 236 page standard!) because it tried to do everything



Alternative: Web of Trust

- Modern public-key infrastructures are structured like trees
- Originally, public-key infrastructures looked like graphs instead
  - Everybody can issue certificates for anyone else
  - Example: Alice signs Bob's key. Bob signs Carol's key. If Dave trusts Alice, he trusts Bob and Carol.
  - Benefit: You know the trust anchor personally (e.g. because you met them in-person, or because you signed their key)
  - Problem: Graphs get far more complex than trees!
- OpenPGP (Pretty Good Privacy) originally used the web of trust model
  - Key-signing parties: meeting in-person to sign each other's public keys
  - It quickly proved to be a disaster
  - Instead, everyone just relies on MIT's central keyserver which is broken!
- **Takeaway**: Trust anchors make public-key infrastructures much simpler!

Summary: Certificates

- Certificates: A signed attestation of identity
- Trusted directory: One server holds all the keys, and everyone has the TD's public key
  - Not scalable: Doesn't work for billions of keys
  - Single point of failure: If the TD is hacked or is down, cryptography is broken
- Certificate authorities: Delegated trust from a pool of multiple root CAs
  - Root CAs can sign certificates for intermediate CAs
  - Revocation: Certificates contain an expiration date
  - Revocation: CAs sign a list of revoked certificates

Password Hashing

Textbook Chapter 14

Review: Cryptographic Hashes

- Hashes accept arbitrarily large inputs
- Hashes "look" random
  - Change a single bit on the input and each output bit has a 50% chance of flipping
  - And until you change the input, you can't predict which output bits are going to change
- The ones we talked about are *fast*
  - Can operate at many many MB/s: Faster at processing data than block ciphers
- Recall: Security properties
  - One way: Given an output $y=H(x)$ from a random x, it is infeasible to find any input x' such that $H(x') = y$.
  - Collision resistant: It is infeasible to find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.

Storing Passwords

- Password: A secret string a user types in to prove their identity
  - When you create an account with a service: Create a password
  - When you later want to log in to the service: Type in the same password again
- How does the service check that your password is correct?
- Bad idea #1: Store a file listing every user's password
  - Problem: What if an attacker hacks into the service? Now the attacker knows everyone's passwords!
- Bad idea #2: Encrypt every user's password before storing it with a service key
  - Problem: The attacker could steal the passwords file *and* the key and decrypt everyone's passwords!
- We need a way to verify passwords *without* storing information that would allow someone to easily recover the original password

Password Hashing

- For each user, store a *hash* of their password
- Verification process
  - Hash the password submitted by the user
  - Check if it matches the password hash in the file
- What properties do we need in the hash?
  - Deterministic: To verify a password, it has to hash to the same value every time
  - One-way: We don’t want the attacker to reverse hashes into original passwords

Password Hashing: Attacks

- What if two different users decide to use **`password123`** as their password?
  - Hashes are deterministic: They'll have the same password hash
  - An attacker can see which users are using the same password
- Brute-force attacks
  - Most people use insecure, common passwords
  - An attacker can pre-compute hashes for common passwords: *H*(**`"password123"`**), *H*(**`"password1234"`**), *H*(**`"1234567890"`**), etc.
  - **Dictionary attack**: Hash an entire dictionary of common passwords
- **Rainbow tables**: An algorithm for computing hashes that makes brute-force attacks easier

Salted Hashes

- Solution #1: Add a unique, random salt for each user
- **Salt**: A random, public value designed to make brute-force attacks harder
  - For each user, store: username, salt, H(password || salt)
  - To verify a user: look up their salt in the passwords file, compute H(password || salt), and check it matches the hash in the file
  - Salts should be long and random
  - Salts are not secret (think of them like nonces or IVs)
- Brute-force attacks are now harder
  - Assume there are M possible passwords and N users in the database
  - Unsalted database: Hash all possible passwords, then lookup all users' hashes ⇒ $O(M + N)$
  - Salted database: Hash all passwords for each user's salt ⇒ $O(MN)$

Slow Hashes

- Solution #2: Use slower hashes
- Cryptographic hashes are usually designed to be fast
  - SHA is designed to produce a checksum of your 1 GB document as fast as possible
- Password hashes are usually designed to be slow
  - Legitimate users only need to submit a few password tries. Users won't notice if it takes 0.0001 seconds or 0.1 seconds for the server to check a password.
  - Attackers need to compute millions of hashes. Using a slow hash can slow the attacker by a factor of 1,000 or more!
  - Note: We are not changing the asymptotic difficulty of attacks. We're adding a large constant factor, which can have a huge practical impact for the attacker

Slow Hashes: PBKDF2

- **Password-based key derivation function 2 (PBKDF2)**: A slow hash function
  - Setting: An underlying function that outputs random-looking bits (e.g. HMAC-SHA256)
  - Setting: The desired length of the output (*n*)
  - Setting: Iteration count (higher = hash is slower, lower = hash is faster)
  - Input: A password
  - Input: A salt
  - Output: A long, random-looking *n*-bit string derived from the password and salt
  - Implementation: Basically computing HMAC 10,000 times
- Benefits (assuming the user password is strong)
  - Derives an arbitrarily long string from the user's password
  - Output can be directly used as a symmetric key
  - Output can also be used to seed a PRNG or generate a public/private key pair
  - Algorithm is slow, but doesn't use a lot of memory (alternatives like Scrypt and Argon2 use more memory)

Offline and Online Attacks

- **Offline attack**: The attacker performs all the computation themselves
  - Example: Mallory steals the password file, and then computes hashes herself to check for matches.
  - The attacker can try a huge number of passwords (e.g. use many GPUs in parallel)
  - Defenses: Salt passwords, use slow hashes
  - If an attacker can do an offline attack, you need a really strong password (e.g. 7 or more random words)
- **Online attack**: The attacker interacts with the service
  - Example: Mallory tries to log in to a website by trying every different password. Mallory is forcing the server to compute the hashes.
  - The attacker can usually only try a few times per second, with no parallelism
  - Defenses: Add a timeout or rate limit the number of tries to prevent the attacker from trying too many times

Summary: Password Hashing

- Store hashes of passwords so that you can verify a user’s identity without storing their password
- Attackers can use brute-force attacks to learn passwords (especially when users use weak passwords)
    - Defense: Add a different **salt** for each user: A random, public value designed to make brute-force attacks harder
- **Offline attack**: The attacker performs all the computation themselves
    - Defense: Use salted, slow hashes instead of unsalted, fast hashes
- **Online attack**: The attacker interacts with the service
    - Defense: Use timeouts