Authentication based on passwords

Password-based authentication

- It doesn’t need definition
    - everybody knows it
- Based on pairing a username and a password
- The factor "What you know"
- Recall: a password must be typeble

Leet speak

- Alternative alphabet used primarily on the internet, where letters in words are replaced with other characters that resemble them visually
    - Introduced in 80s on BBSs
- Often used in combination with some password
    - humans need to remind them, they cannot be a random sequence of letters/digits/symbols
        - unless a password manager is used

Leet speak examples

Standard text	Leet speak
hello	h3ll0
hacker	h4×0r
elite	3l1t3
password	p@55w0rd

Remind...

For the same length, a key has more possible combinations than a password, because some of these bit-combinations are either not typeable or do not correspond to any meaningful character

Common password attack models

Model	Capabilities	Example	Impact on design
Online attacker	Limited guesses; must go through system's login interface	Botnet trying to brute-force accounts	Enforce lockouts, MFA, CAPTCHA, rate limits
Offline attacker	Has stolen hashes/salts; unlimited guesses offline	Breach of hashed password DB	Use strong hashing (Argon2, bcrypt), long passwords
Insider attacker	Legitimate access to system internals	Malicious admin	Separation of duties, logging, privileged account monitoring
Targeted attacker	Focused on one user; may use personal info	Stalker or APT	Avoid guessable passwords, enable MFA

Attacker's goals

Password Attack Type	Main Goals	Definition
Brute force / Dictionary attack	Unauthorized access, service abuse	Systematically tries many password combinations (or words from a list) until the correct one is found
Credential stuffing	Unauthorized access, lateral movement	Uses stolen username/password pairs from one breach to try logging into other sites
Social engineering	Unauthorized access, data theft, privilege escalation, selling credentials	Manipulates people into revealing passwords or information
Keylogging / malware	Data theft, unauthorized access, privilege escalation	Malicious software records keystrokes or captures stored credentials
Database breach	Selling credentials, credential stuffing, lateral movement	Extracts password databases from compromised servers

A simple example

- A = client, B = server
- B stores H(password) *(H is OWF)*
- A authenticates to B

1. A→B: A
2. B→A: T *(timestamp + nonce)*
3. A→B: H(H(password)||T)

B verifies

- Robust against replay
- Not fully robust against other attacks

Two possible attacks

- If the server DB is compromised, the attacker can perform an offline brute-force or dictionary attack
    - Attackers can exploit precomputed lookup tables mapping candidate words to hash values
        - Optimized version: rainbow tables
        - These circumvent the one-way property of the hash function
- If an adversary eavesdrops, they can perform an offline guessing attack using the captured communication

Partial mitigation

- Lookup tables can be made useless by adding a **salt**
    - A salt is a sequence of random bits
    - Server stores H(password||salt)
    - Each user has a different salt
    - Salts are stored in plaintext
    - Salting does not prevent offline guessing
- It is infeasible to precompute a table for all possible salts
- This slows down offline brute-force and dictionary attacks

Revised protocol

- A = client, B = server
- B stores (salt, H(password||salt)) *(H is OWF)*
- A authenticates to B

1. A→B: A
2. B→A: (salt,T) *(T = timestamp + nonce)*
3. A→B: H(H(password||salt)||T)

Lookup tables are now useless

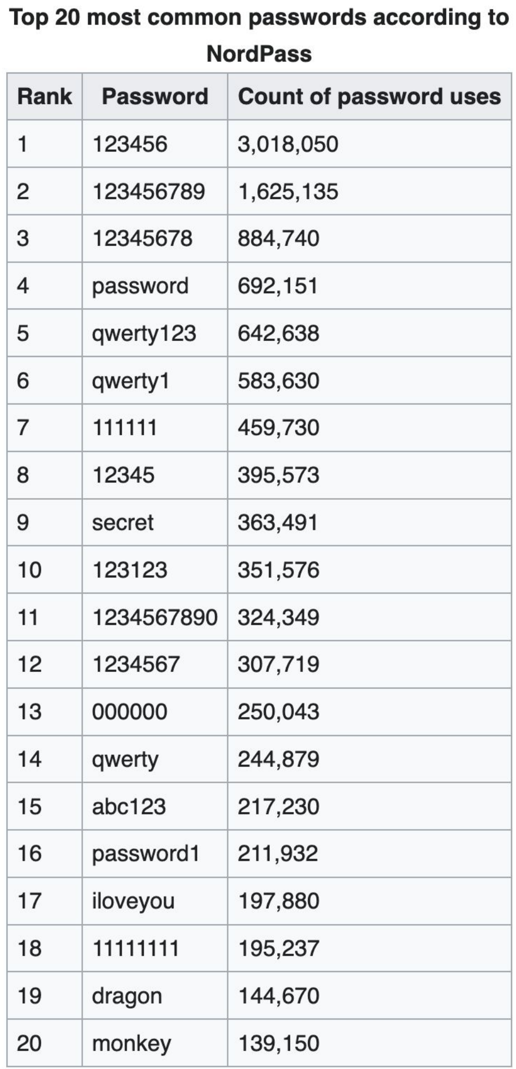
B verifies

Leaked passwords

- The website “Have I Been Pwned” (HIBP) maintains a very large and constantly growing database of compromised passwords called Pwned Passwords
- As of 2025, HIBP’s database includes approximately 1.29 billion unique compromised passwords that have been collected from various data breaches, leaks, and malware captures
- https://haveibeenpwned.com/
- To obtain the list of passwords, get the downloader at https://github.com/HaveIBeenPwned/PwnedPasswordsDownloader

Weak passwords

- From Wikipedia https://en.wikipedia.org/wiki/List_of_the_most_common_passwords#NordPass



Top 20 most common passwords according to NordPass

Rank	Password	Count of password uses
1	123456	3,018,050
2	123456789	1,625,135
3	12345678	884,740
4	password	692,151
5	qwerty123	642,638
6	qwerty1	583,630
7	111111	459,730
8	12345	395,573
9	secret	363,491
10	123123	351,576
11	1234567890	324,349
12	1234567	307,719
13	000000	250,043
14	qwerty	244,879
15	abc123	217,230
16	password1	211,932
17	iloveyou	197,880
18	11111111	195,237
19	dragon	144,670
20	monkey	139,150

Common mistakes & mitigations

- Weak password selection
  - Use of dictionary words
  - Predictable choices exploitable via social engineering
- Password reuse
  - Extends the impact of a single credential compromise across multiple systems
- Mitigations
  - Employ alternative authentication mechanism or use MFA
  - Use a password manager
    - Integrated in some operating systems as a native feature

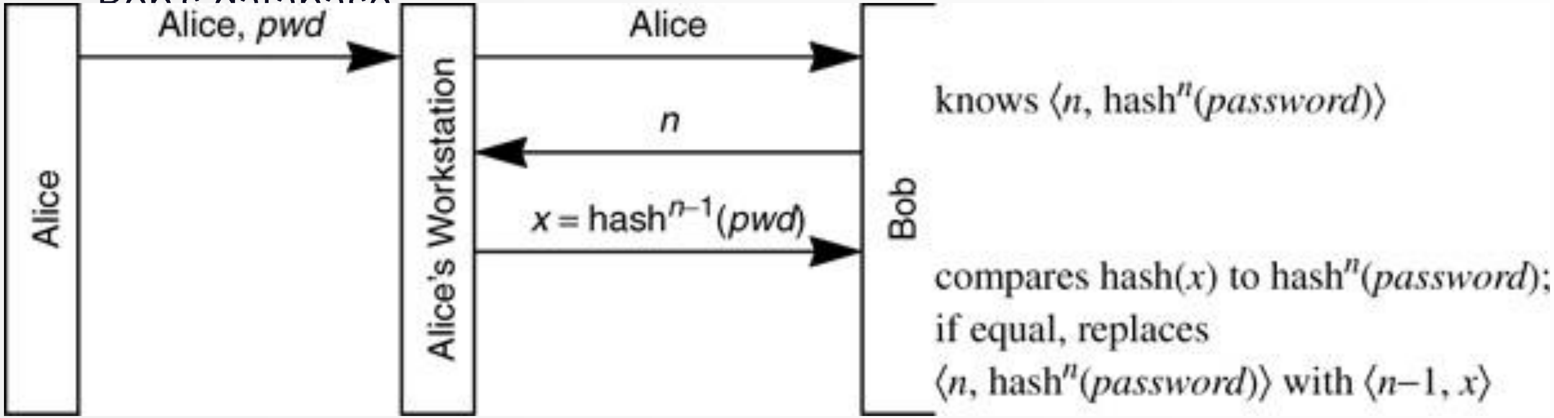
More advanced password protocols

Goals

- Obtaining the benefits of cryptographic authentication with the user being able to remember passwords only
- In particular
  - no security information is kept at the user’s machine (the machine is trusted but not configured)
  - someone impersonating either party will not be able to obtain information for off-line password guessing (online password guessing is not preventable, but mitigable)

Lamport’s Hash [1981]

- Bob stores <username, n, h^n(password)>, n is a relatively large number, like 1000
- Alice’s workstation sends x = h^{n-1}(password)
- Bob computes h(x): if match successful, n is decremented, x replaces h^n in Bob’s database


Alice
Alice, *pwd*
Alice’s Workstation
Alice
n
$x = \text{hash}^{n-1}(pwd)$
Bob
knows $\langle n, \text{hash}^n(\textit{password})\rangle$
compares hash(x) to $\text{hash}^n(\textit{password})$;
if equal, replaces
$\langle n, \text{hash}^n(\textit{password})\rangle$ with $\langle n-1, x\rangle$

- why is sequence of hash transmissions reversed?
- safe against eavesdropping, database reading
- no authentication of Bob

Salting Lamport’s hash

- h^{n-1}(pwd||salt) is used for authentication
- salt is stored at Bob’s at setup time, Bob sends salt each time along with n
- advantages
  - Alice can use the same password with multiple servers
    - if servers use different salts hashes are different!
    - to ensure that the salts are different, servers name are also hashed in
  - easy password reset (when n reaches 1): just change the salt
  - defense against dictionary attacks
    - dictionary attack without the salt: compile h^k of all the words in the dictionary, for all k's from 1 to 1000; *easier to check results in pwd db!*

Lamport’s hash: other properties

- **small n attack**
  - when Alice tries to login Trudy impersonates Bob and sends n′ < n and salt, when Trudy gets the reply, she can impersonate Alice until n is decremented to n′
  - **defense**: Alice’s workstation shows submitted n to Alice to verify the “approximate” range (Alice must remember it)
    - or Alice uses a personal device (unique) that can store n
- **“human and paper” environment**
  - in case Alice workstation is not trusted or too “dumb” to do hashing
  - Alice is given a list of all hashes starting from 1000, she uses each hash exactly **once**
    - automatically prevents small n attack
    - string size? 64 bits (~10 characters) is secure enough
- implemented as S/Key and standardized as one-time password system (RFC 1938)

Authentication EKE: Encrypted Key Exchange

Problem: dictionary attack if weak keys (i.e., easily guessable) are chosen

EKE

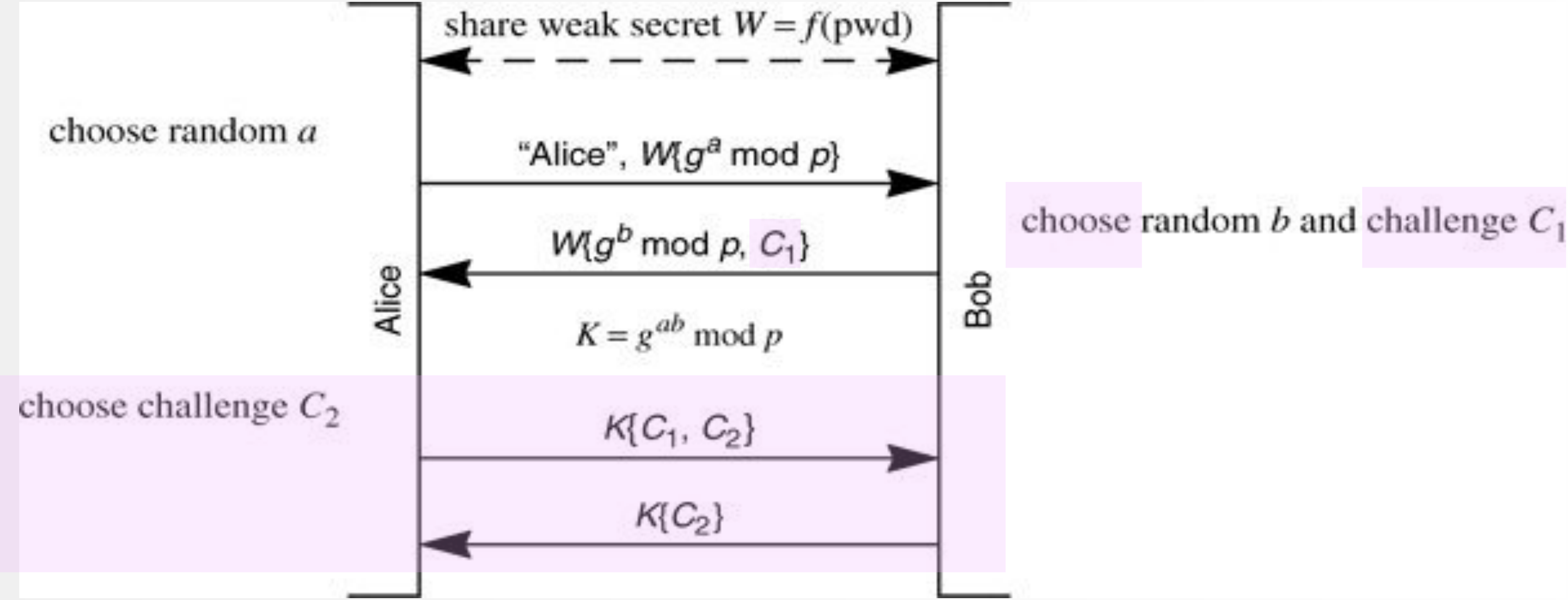
- Strong wrt dictionary attack
- Mutual authentication
- Define session key

Scenario:

- User and server share a weak secret
- User and server use secret to authenticate and define a session key (Diffie-Hellman)

EKE basic authentication

pwd = Alice's password; Bob just knows *W*



EKE: basic properties

EKE is strong wrt

- replay attacks
  - ○ *a* is changed every time
- dictionary attacks
  - ○ even if the chosen password is weak the choice of random *a* does not allow the attacker to compute g^a

authentication is strong because uses strong session key *k*

Note: if the attacker knows the password, then can act in place of A

Strong hashing

- **Purpose**: slow down offline password guessing by increasing computational cost
- **Key properties**:
  - Salted: prevents precomputed attacks (rainbow tables)
  - Memory-hard: limits advantage of GPUs/ASICs
  - Configurable cost: adjustable iteration count to match current hardware speeds
- Examples
  - bcrypt – CPU-bound, built-in salt, adjustable cost factor
  - scrypt – memory-hard, resists parallel cracking
  - Argon2 – Winner of the PHC; configurable memory, time, parallelism
  - PBKDF2 – widely supported; uses HMAC, adjustable iterations
- Best Practice
  - Always combine strong KDFs with unique per-user salts and MFA

Comparison

Algorithm	Salt built-in	Memory-hard	Configurable cost	Security profile
bcrypt	Yes	No	Iterations	Security (>25 years), widely supported, slower on GPUs
scrypt	Yes	Yes	Memory & time	Good GPU/ASIC resistance via high memory use
Argon2	Yes	Yes	Memory, time, parallelism	PHC winner; modern standard for password hashing
PBKDF2	No	No	Iterations	Very widely supported; must add own salt; fast on GPUs

EKE variants

- SPEKE (Simple Password Exponential Key Exchange)
  - uses W in place of g in DH exchange
  - transmits W^a mod p and W^b mod p, session key is W^{ab} mod p
- PDM (Password Derived Moduli)
  - chooses p depending upon password and uses g = 2

EKE weakness and defense

- if Trudy knew *W*, could impersonate Alice
- if passwd file stolen, it is possible to do a dictionary attack
  - if successful Trudy could impersonate the user
  - if unsuccessful, knowledge of *W* still allows to impersonate Alice
- basic EKE schemes (EKE, SPEKE, and PDM) can be modified to have an **augmented** property
  - the idea is for Bob to store a quantity derived from the password that can be used to verify the password, but Alice's machine is required to know the password (not the derived quantity stored at the server)

Enhancing EKE schemes

Augmentation property

- **Definition**: an EKE protocol has the augmentation property if compromise of the server’s stored verifier does not enable an attacker to authenticate without first knowing the original password
- **Benefit**: server compromise alone is insufficient for user impersonation, since authentication still requires the original password or an offline dictionary attack against the verifier

On augmentation

Augmentation

1. the server stores a derived value (not the password itself), so even if the database is compromised, attackers cannot directly impersonate users
2. requires the client to know the actual password to complete authentication, enhancing security without additional user burden

Why EKE is not augmented

- it relies on the server storing and directly using password-derived data for authentication. If the server is compromised, this data can be used in offline attacks or attacker can impersonate Alice
- augmented protocols store derived values that are useless without the user's actual password, providing better security against server breaches

Augmented EKE

- Globals: prime p, generator g
- Bob’s per-user stored data
    - Username (e.g., “Alice”), random public salt u
    - W = H(u||"Alice"||password)
    - v = g^W mod p — verifier derived from password-based W

Augmented EKE steps

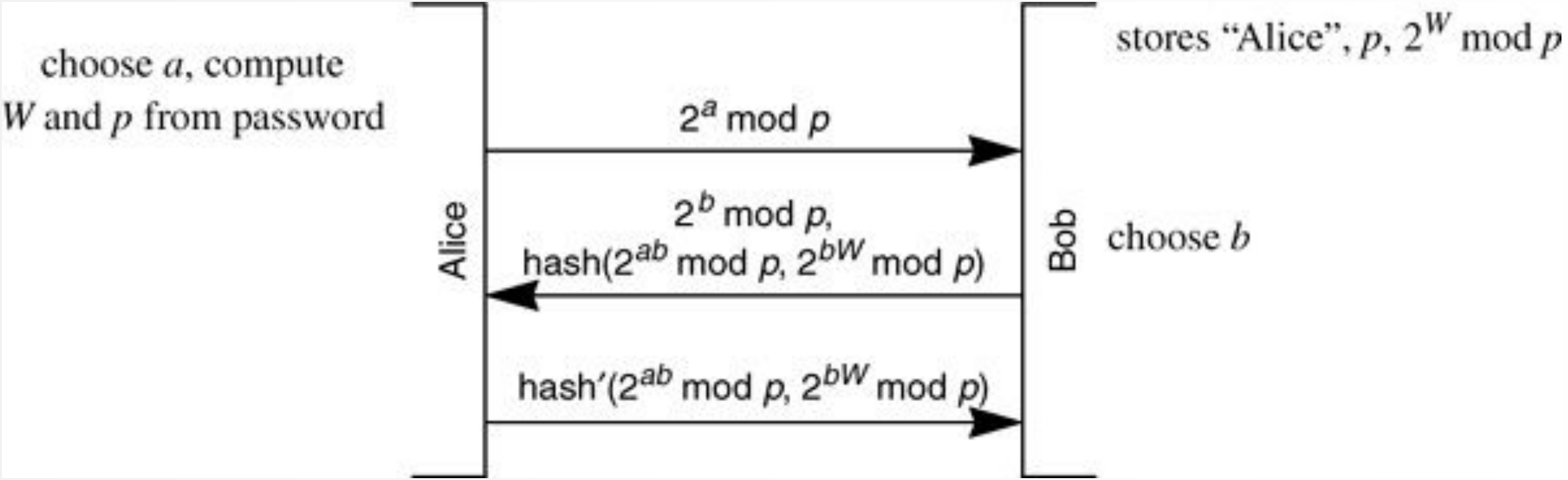
1. A→B: "Alice", $A=g^a$ mod p *(Alice picks random a)*
2. B→A: $B=(v^u \cdot g^b)$ mod p, u, C_B *(Bob chooses random b, u and challenge C_B)*
3. A→B: $E_K(C_B)$, C_A *(Alice computes K, using u to remove v from B)*
4. B → A: $E_K(C_A)$ *(Bob computes same K)*

Key computation

- Alice:
  - $S=(B/v^u)^a$ mod p = $(g^b)^a$ mod p
  - K = H(S)
- Bob:
  - S = A^b mod p = $(g^a)^b$ mod p
  - K = H(S)

Augmented PDM

server stores p and 2^W mod p, where W is (still) a hash of user's password



Alice

choose a, compute W and p from password

Bob

stores “Alice”, p, $2^W \bmod p$

$2^a \bmod p$

choose b

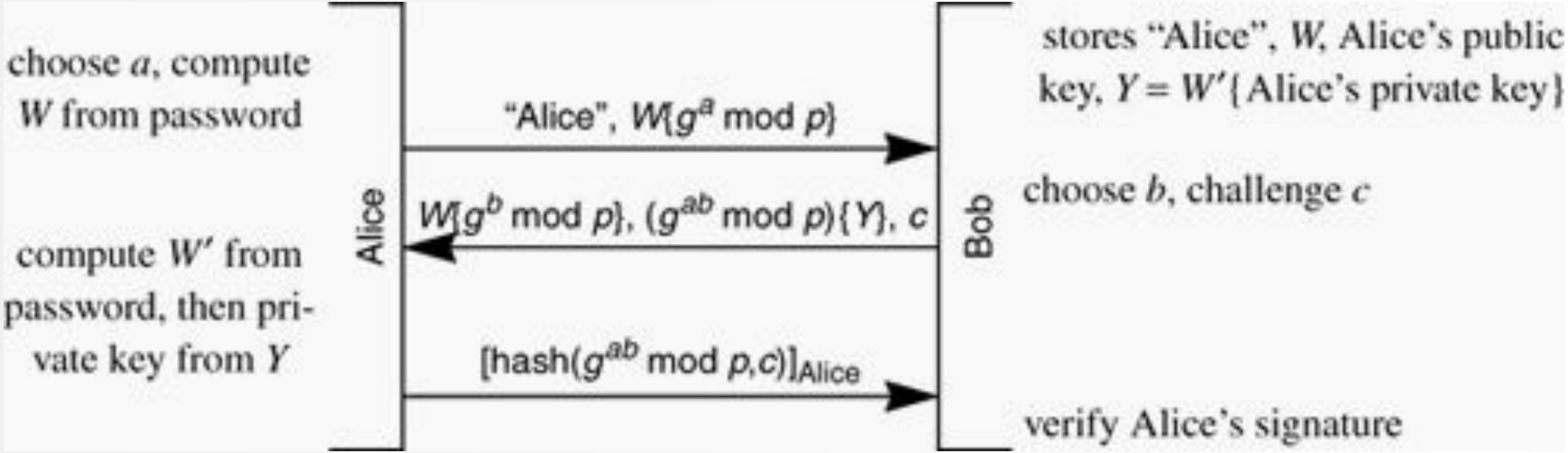
$2^b \bmod p$, hash($2^{ab} \bmod p$, $2^{bW} \bmod p$)

hash′($2^{ab} \bmod p$, $2^{bW} \bmod p$)

Augmentation at higher performance (server side)

- instead of requiring server to do an additional Diffie-Hellman exponentiation, it does an RSA verify operation, which is much less expensive
- Bob stores, for Alice, an RSA private key encrypted with Alice's password, and the corresponding public key
- this can be done with any of the basic schemes (EKE, SPEKE, or PDM)

Augmentation at higher performance (server side)



choose a, compute W from password

stores "Alice", W, Alice's public key, $Y = W'\{$Alice's private key$\}$

"Alice", $W\{g^a \bmod p\}$

choose b, challenge c

$W\{g^b \bmod p\}$, $(g^{ab} \bmod p)\{Y\}$, c

compute W' from password, then private key from Y

$[\text{hash}(g^{ab} \bmod p, c)]_{\text{Alice}}$

verify Alice's signature

Discussion

- Bob stores Y, which is Alice's private key encrypted with a function of their password
    - Different hash of the password than W, or someone that stole the server database would be able to obtain her private key
- Bob also stores Alice's RSA public key corresponding to the encrypted private key
- In message 1, Alice sends the usual first EKE message, consisting of her Diffie-Hellman value encrypted with W
- In message 2, Bob sends his Diffie-Hellman value, along with Y (Alice's encrypted private key), encrypted with the agreed-upon Diffie-Hellman key
    - Alice extracts Y by decrypting with g^{ab} mod p, and then decrypts Y with her password to obtain her private key
- In message 3, Alice signs a hash of the Diffie-Hellman key and the challenge c, and Bob verifies her signature using the stored public key
- **This achieves mutual authentication as well as the augmented property**

Authentication with/without session keys

Understand differences

What we mean here

- Two possible outcomes after authentication
  - Authentication + fresh session key establishment
  - Authentication only (no new key derived)
- Session key = short-lived, high-entropy key for secure channel
- Authentication only = proof of identity without key generation

Real-world case studies

- Smart card door access
  - Authenticates, no encrypted channel after
- Offline device login
  - Local password check without network key exchange
- TLS client certificate
  - Mutual auth + session keys for data protection