Password

authentication through passwords

- Human beings
  - Short keys; possibly used to generate longer keys
  - Dictionary attack: adversary tries more common keys (easy with a large set of users)
  - Trojan horse
  - Countermeasures: slow login, close after several unsuccessful attempts
- Computers
  - Quality keys (long and not predictable)
  - Hidden: not stored in the clear (encrypted, one time passwords)

Passwords

Password problems

Eavesdropping: adversary is sniffing

- password must not be sent in the clear
- Authentication should be different each time (to avoid replay attacks)

Store password securely:

- Adversary can access database of passwords: encrypt passwords

Unix password hash

Idea: passwords are not stored: data obtained from passwords are stored (use hash)

- user password is first converted to a secret key K (56 bits, obtained by considering the 7-bit ASCII associated with each of the first 8 characters of password - then DES parity added)
- store $DES_K(000…0)$
  - actually $DES_K(DES_K(DES_K(…..DES_K(000…0)…)))$ (25 times)
- DES' variant used for making fast DES hardware devices useless

Problem: dictionary attack (users keys are predictable)

- attacker reads password database and there is a high probability that there is at least one user with a weak password
- to increase security use salt (12 bit random number): modify DES through salt and encrypt 000… 0<salt>
- salt is per-user generated and can be stored in the clear
  - salt increases work for attacker because it makes impossible to hash a (guessed) password and check if it matches some user's password, but does not solve the problem of weak users' key

Unix password hash

Alice wants to authenticate herself to Bob

- send passwd in the clear - eavesdropper!
- do Diffie-Hellman exchange for establishing a secret key to be used for encrypting passwd - Trudy can impersonate Bob!
- use a challenge/response handshake - eavesdropper can carry out dictionary attack
- use strong password protocols

Alice's authentication

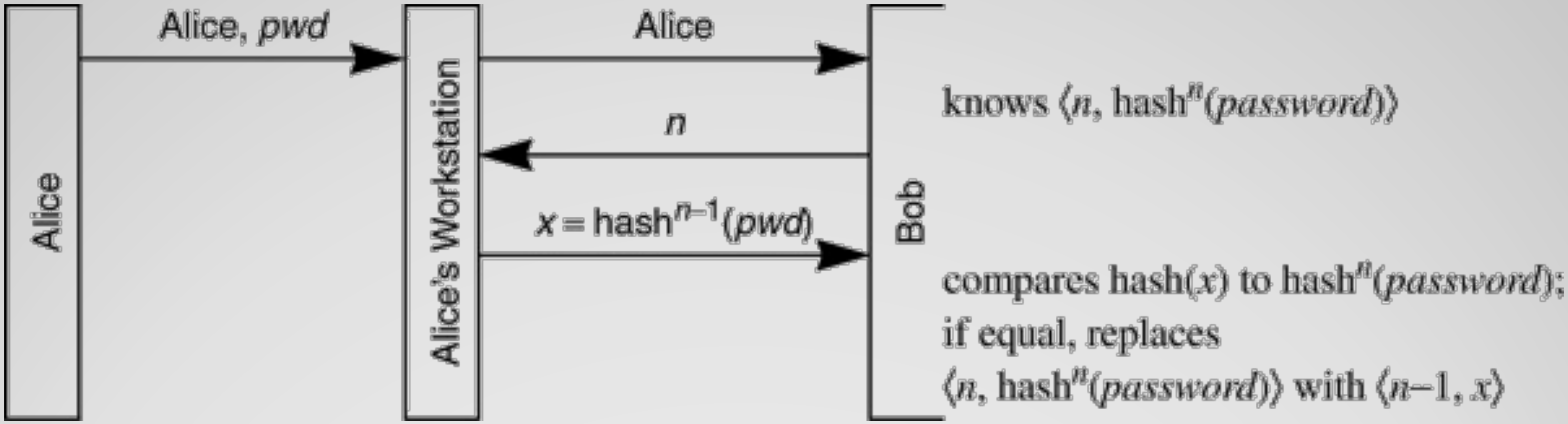
Strong password protocols

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)

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 successful, n is decremented, x replaces h^n in Bob’s database



- why is sequence of hash transmissions reversed?
  - if you increment instead of decrementing it does NOT work
- safe against eavesdropping, database reading
- no authentication of Bob

- 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, why?
    - 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!*

Salting Lamport's Hash

- **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 has to remember it)
- **“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)

Lamport’s Hash: other properties

Authentication EKE: Encrypted Key Exchange

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

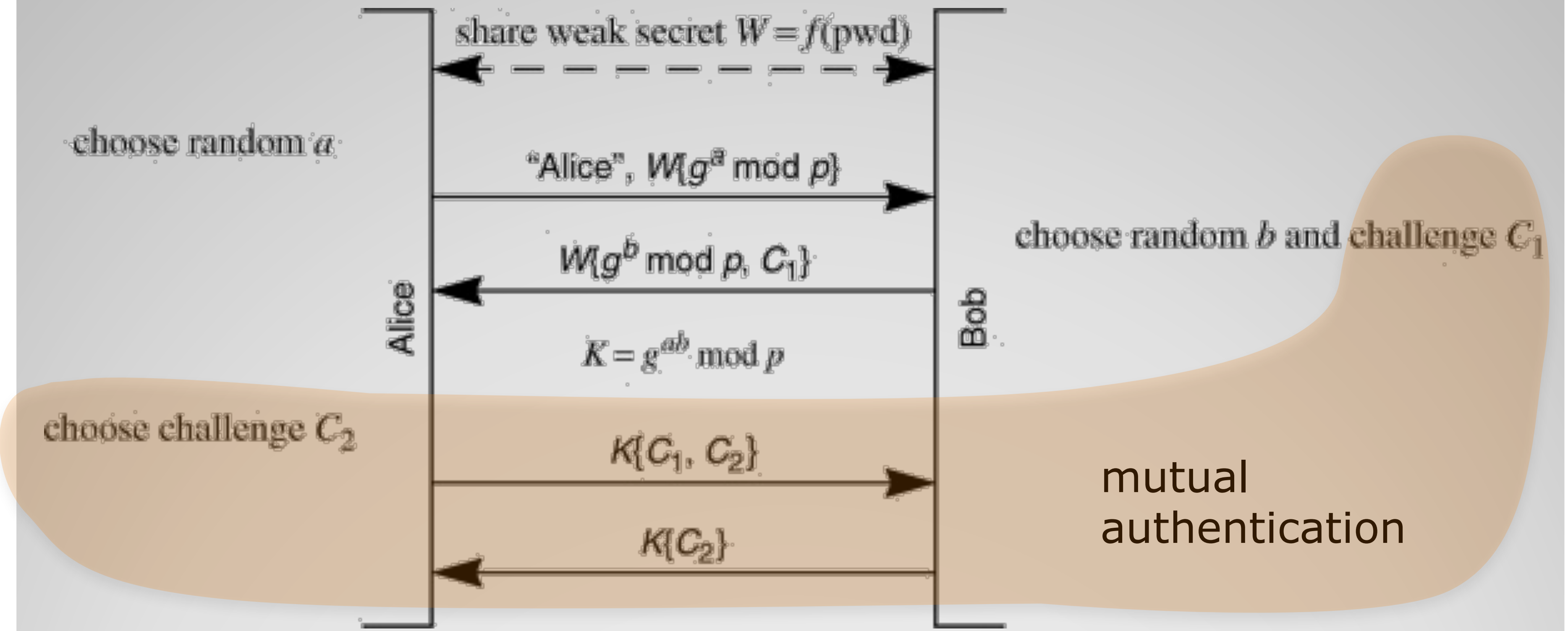
EKE

- Strong w.r.t. 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)

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



EKE basic authentication

EKE: basic properties

EKE is strong w.r.t.

- 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

SPEKE (Simple Password Exponential Key Exchange)

- uses W in place of g in D.-H. 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 $g = 2$

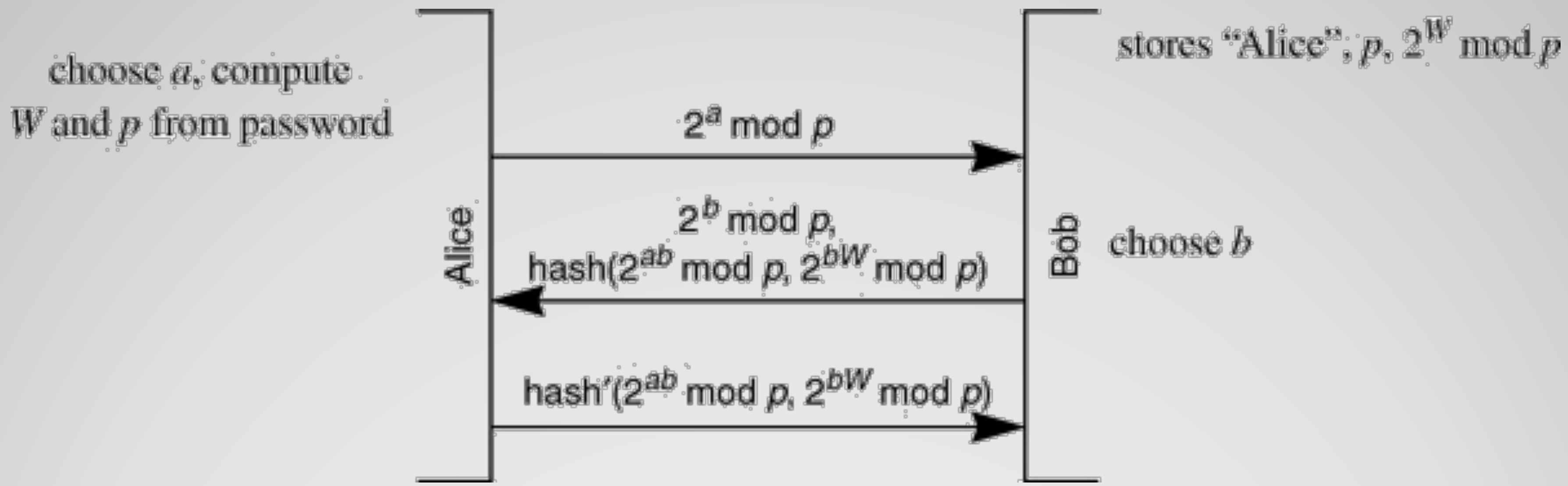
EKE variants

- 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 a 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)

EKE weakness and defence

example for PDM

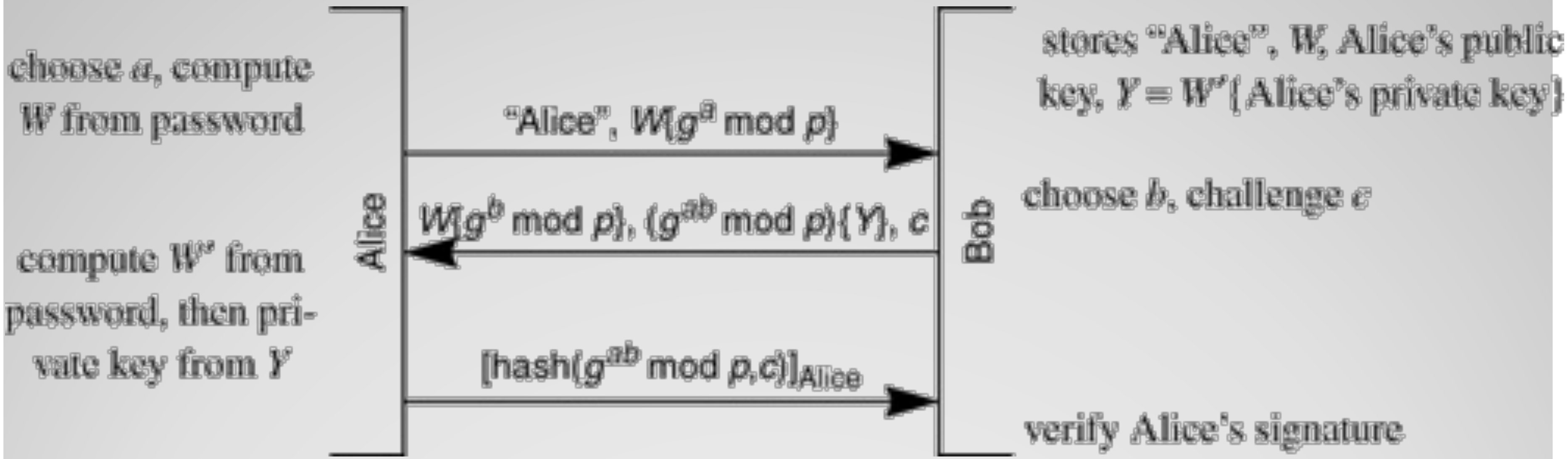
- server stores p and 2^W mod p, where W is (still) a hash of user's password



Augmented strong password protocols

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
- this is accomplished by having Bob store, 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)



augmented EKE example

- Bob stores Y, which is Alice's private key encrypted with a function of her 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

discussion