Authentication I

Computer and Network Security

Emilio Coppa

Authentication

- Authentication: human vs computers
- Authentication through passwords
  - pitfalls and attacks
  - salted passwords, Lamport's Hash
- Protocols for authentication:
  - Two entities share a secret symmetric key
    - challenge-response based, timestamp based, attacks
  - Entities share a secret with a trusted entity
    - authentication server, Needham–Schroeder symmetric protocol
    - Kerberos
  - Entities have public/private keys
    - Needham–Schroeder public-key protocol
    - PKI, X.509

Authentication

The process of reliably verifying the identity of someone (or something).

Human vs computer authentication:

- computer can store a high-quality secret, such as a long random-looking number, and it can do cryptographic operations
- a workstation can do cryptographic operations on behalf of the user, but the system has to be designed so that all the person has to remember is a password
- password can be used to acquire a cryptographic key in various ways:
  - derive the key from the password, e.g., performing hash of the password
  - use the password to locally decrypt a higher quality key, e.g., decrypt RSA key

Authentication of people

Different approaches:

- what you known
- what you have
- who you are
- where are you

These approaches can be combined to have multi-factor authentication. Authenticate the user on a machine is often the first step to several authentication protocols (e.g., to use a service in a network).

Authentication of people: what you known

The most prominent example is using passwords, i.e., a secret known only to the user, answers to specific personal questions (recovery procedure).

Many problems as these secrets are:

- often guessable by an attacker as they are typically short and not truly random
- can be captured with specific malicious software (e.g., a Trojan showing a fake login and/or using a keylogger) or social engineering tehniques

More details over authentication through password later in other slides.

Authentication of people: what you have

- authentication token: a physical device that can perform specific cryptographic operations based on an internal secret key (which is never revealed). Can communicate directly with a system for carrying out a specific protocol or generate a One Time Password (OTP) for the user, which is valid for a limited amount of time (e.g., 60 seconds)

- smartphone: it can play the role of a “software” authentication token (general purpose, wide spread, easy to replace) or can be used to receive a OTP from another channel (e.g., OTP via SMS, which however is not 100% secure)

Open standards for OTP:

- HMAC-based One-Time Password (HOTP)
- Time-based One-Time Password (TOTP)

Authentication of people: who you are

Mainly based on biometric:

- fingerprint reader: available to most people nowadays due to smartphone but hardware and software implementation is not certified and always trustable.

- retina examination: it can be accurate but require expensive hardware

- voice recognition: a lot of research during the latest years (e.g., Alexa), accuracy can be affected by health issues (e.g., flu), not always suitable for security (e.g., due to attacks based on machine learning)

Authentication of people: where you are

In general it is not considered to be secure as an attacker can fake his location. However, it can be used as an additional factor during authentication and operations:

- Websites will often warn you about logins from unexpected locations
- Banks may refuse transactions when credit cards are used from unexpected locations or when used on unexpected web sites at unexpected time

Real-world example: 3-D Secure

Strong customer authentication (SCA) is a requirement of the EU Revised Directive on Payment Services (PSD2) on payment service providers within the European Economic Area.

One way for performing SCA is 3-D Secure. The name refers to the "three domains" which interact using the protocol: the merchant/acquirer domain, the issuer domain, and the interoperability domain.

The basic concept of the protocol is to tie the financial authorization process with online authentication: one common way is to redirect the user to a page (from the Bank) and asks a password that is tied to the current card. More recently, OTP via SMS.

Major concern: the user should understand that the page (or iframe) is the right one

HMAC-based One-Time Password algorithm (HOTP) [RFC 4266]

Idea: compute a value using HMAC providing two inputs (secret K, counter C), which are known to both parties. The value must d digits, e.g., d=8. After each attempt, the counter is incremented.

$$value = HOTP(K, C) \bmod 10^d$$
$$HOTP(K, C) = truncate(HMAC(K, C))$$

Problem. The counter must been synchronized

Possible (partial) solution. The server will compute W values (e.g., W=100), considering many increasing value of the counter C and will check all of them. Hence, there is “window” for re-resynchronization in each attempt. However, if the client increments the counter too much (e.g., a kid pressing the button on the HTOP device), then it will not work even using a “window”.

Time-based One-Time Password algorithm (TOTP) [RFC 6238]

Idea: use a time-based value instead a synchronized counter in HOTP.

$$TOTP(K, C) = HTOP(K, C_T)$$

$$C_T = \left\lfloor \frac{T - T_0}{T_X} \right\rfloor$$

where:

Check this Python implementation!

- T is the current unix time
- T_0 is an initial time (unix epoch)
- T_X is the length of window duration (e.g., 30 seconds)

To make the process easier, the server will check will often consider T, T+1, T-1.

Authentication through Passwords

Passwords

Common problems:

- robustness: passwords chosen by humans are weak as they are short and not random, thus they can be guessed by an attacker.

- never send password in clear: attacker can sniff the password when sent in clear, hence a authentication password should be different each time to prevent replay attacks

- store passwords securely: if the password is stored somewhere then it is crucial to store it safely, e.g., password manager (e.g., Bitwarden) should keep password encrypted

Storing password locally

Many (operating) systems store passwords for (local) users in a (local) file:

- the file cannot be read by all standard users: e.g., on Linux, password are stored inside /etc/shadow:

  ```
  $ ls -l /etc/shadow
  -rw-r----- 1 root shadow 1545 set  1 19:40 /etc/shadow
  ```

- the file should contain hash of the passwords: even if the file is leaked, the adversary does not (yet) know the passwords.

Dictionary Attack

Since passwords are not truly random in practice, an attacker can take a “dictionary”, i.e., a list of popular words used in passwords, and repeating the procedure for each word from the dictionary:

- online attack: the adversary performs the authentication procedure on the system. He does not need to know details about the authentication procedure (e.g., the specific hash algorithm), but the throughput (attempts per second) could be low. Most systems make the authentication process slow (e.g., 1-2 seconds) to limit the throughput and ban the user after several wrong attempts.

- offline attack: the adversary knows the authentication procedure and has a dump of the database of hashed passwords (e.g., /etc/passwd in UNIX). Since it can run the procedure on its machine, the throughput can be very high (e.g., using a GPU for massive parallelism).

Additionally problems of hashed passwords

Authentication procedures without a randomizer:

- if the attacker gets a dump of the database of hashed passwords, it can easily detect when two users have the same password (the hash will be the same)

- attackers can easily find online precomputed tables of hashed passwords for rich dictionaries.
    - offline attack is much easier: lookup in a table
    - the attack can “cross” information among different sites:
        - website A (e.g., win-2-euros.it) is compromised, dump of password database is leaked, weak authentication process. The attacker recovers password of user X.
        - website B (e.g., Google) is NOT compromised but user X use the same password on website A and website B. The attacker try the compromised password from website A and some variations.

Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

https://crackstation.net/

Non sono un robot

reCAPTCHA
Privacy - Termini

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Download CrackStation's Wordlist

How CrackStation Works

CrackStation uses massive pre-computed lookup tables to crack password hashes. These tables store a mapping between the hash of a password, and the correct password for that hash. The hash values are indexed so that it is possible to quickly search the database for a given hash. If the hash is present in the database, the password can be recovered in a fraction of a second. This only works for "unsalted" hashes. For information on password hashing systems that are not vulnerable to pre-computed lookup tables, see our hashing security page.

Crackstation's lookup tables were created by extracting every word from the Wikipedia databases and adding with every password list we could find. We also applied intelligent word mangling (brute force hybrid) to our wordlists to make them much more effective. For MD5 and SHA1 hashes, we have a 190GB, 15-billion-entry lookup table, and for other hashes, we have a 19GB 1.5-billion-entry lookup table.

Home Notify me Domain search Who's been pwned Passwords API About Donate

https://haveibeenpwned.com/

';--have i been pwned?

Check if you have an account that has been compromised in a data breach

email address | pwned?

479	10,196,051,455	113,758	194,794,935
pwned websites	pwned accounts	pastes	paste accounts

Largest breaches

772,904,991	Collection #1 accounts
763,117,241	Verifications.io accounts
711,477,622	Onliner Spambot accounts
622,161,052	Data Enrichment Exposure From PDL Customer accounts
593,427,119	Exploit.In accounts
457,962,538	Anti Public Combo List accounts
393,430,309	River City Media Spam List accounts
359,420,698	MySpace accounts
268,765,495	Wattpad accounts
234,842,089	NetEase accounts

Recently added breaches

1,284,637	Experian (South Africa) accounts
3,385,862	LiveAuctioneers accounts
166,031	Unico Campania accounts
235,233	Utah Gun Exchange accounts
1,173,012	Catho accounts
751,700	Sonicbids accounts
23,927,853	Zoosk (2020) accounts
444,453	ProctorU accounts
768,890	Kreditplus accounts
599,667	TrueFire accounts

Salted passwords

Idea: use a randomizer for each user to “salt” the password

$$h = hash(password \mathbin{||} salt)$$

The salt can be stored in clear near the hash of the password. Salt is not secret, but is chosen randomly. its role is to make the life of an attacker harder by requiring to perform the authentication procedure from scratch for each salt / for each user.

- if two users have the same password, the salt (which will be different for the two users) will make the hashes different
- precomputed table cannot be used (assuming that the salt is not fixed but user-specific)

Salted passwords (2)

- increase the work required from the attacker, preventing from “reusing” or “caching” work done for other users

- if the password of a user is weak, then the attacker can still make “quickly” an attack by hashing common words from a dictionary.

Linux /etc/shadow: username:idsalt$hash:[....]

- username: user name in the system
- id: hashing algorithm (1: MD5, 2a: Blowfish, 5: SHA-256, 6: SHA-512)
- salt: e.g., 8 random chars
- hash: hash of the password computed using the algorithm and the salt

Remote authentication

Suppose that Alice needs to authenticate on a remote server: she will use a client machine to perform the authentication. The client machine compute the hash of the password and sends it to the remote server....

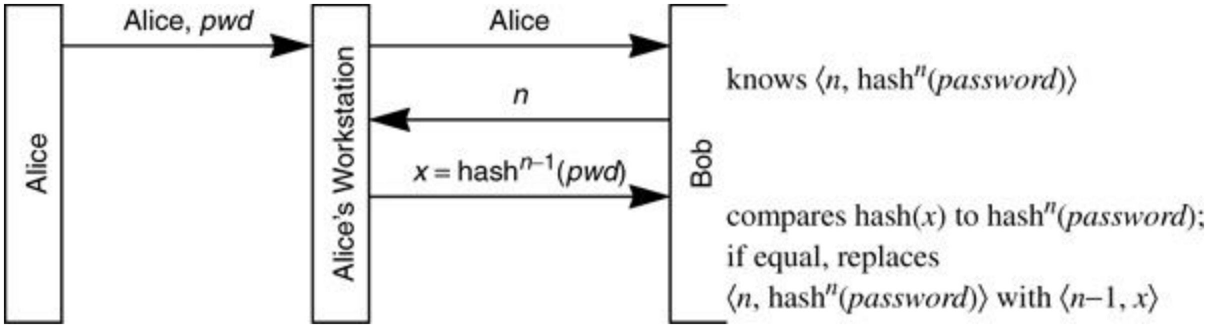
Problems:

- the hash of the password can be intercepted by an adversary and reused later...
- ...even when the hash of the password is useless on the remote server, the adversary may want to crack the hash to get a password from Alice (and then use it for login on another system)

Lamport's Hash

Idea: avoid offline guessing even when sending hash through network

- Bob stores (n, $hash^n$(password)) and sends to Alice n when auth request
- Alice computes and sends $hash^{n-1}$(password)
- Bob checks it comparing $hash^n$(password) with hash($hash^{n-1}$(password)), then updates entry to to (n-1, $hash^{n-1}$(password))



Remark. if hashing is incremented instead of decremented then attack is trial

Salted Lamport’s Hash

Adding salt to Lamport’s Hash: (n, salt, $hash^n$(password || salt))

- “secure” even when user use the same password in different systems
- easy reset when n=0: change the salt instead of changing the password
- more work for the attacker: for each word, for each salt, perform n-th hashing ops

Lamport's Hash: flaws

- Bob is not authenticated to Alice: Man-in-the-Middle (MITM) attack is possible
  Small n attack:
  - Bob stores (n, $hash^n$(password))
  - Attacker impersonates Bob with Alice sending n' with n' << n
  - Alice provides $hash^{n'}$(password)
  - Attacker can authenticate (n - n') times since it just needs to perform hashing starting from $hash^{n'}$(password)

  Mitigation: Alice should keep track of n from Bob

- Machine of Alice cannot be dumb as it has to perform hashing. A solution ("human and paper" approach): hashes are precomputed and given to Alice, which then uses each of the hash only once for authentication. This is known as S/Key, standardized in RFC 1938 "A one-time password system"

Strong Password Protocols

Motivation: Alice and Bob share a secret which is "weak", they want to build a "secure" secret key and attacker should not gain any benefit from observing message exchanged during this process.

Requirements:

- strong w.r.t. dictionary attacks
- mutual authentication
- generate a secure session key

⇒

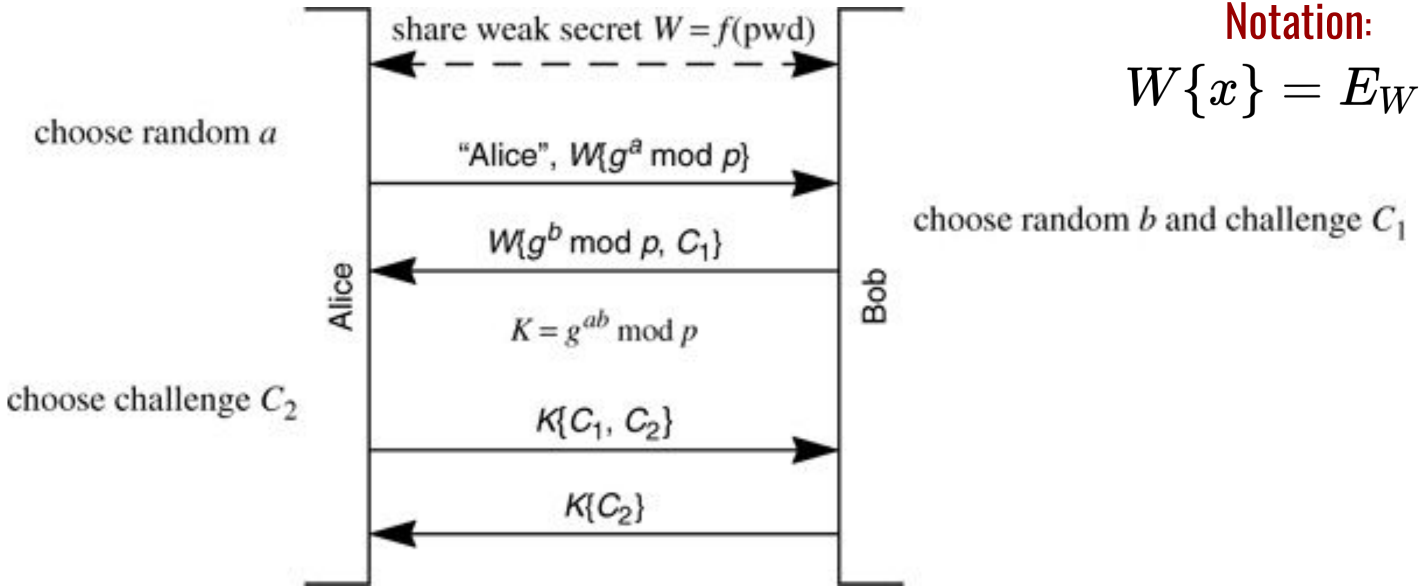
Protocols:

- EKE
- SPEKE

The weak secret could be a weak password (or hash of weak password).

EKE: Encrypted Key Exchange

Idea: Diffie-Hellman Key Exchange where numbers are encrypted using the weak secret W



Notation:

$$W\{x\} = E_W(x)$$

Remark: Challenge C_1 and C_2 are used to perform mutual authentication

EKE: Encrypted Key Exchange (2)

- exchanged messages appear as random numbers

- no way of performing offline attack since a and b change every time. Online attacks are still possible.

- no replay attacks since a and b change every time

- given W, an attacker can authenticate but it is computationally hard to get W by looking at the messages.

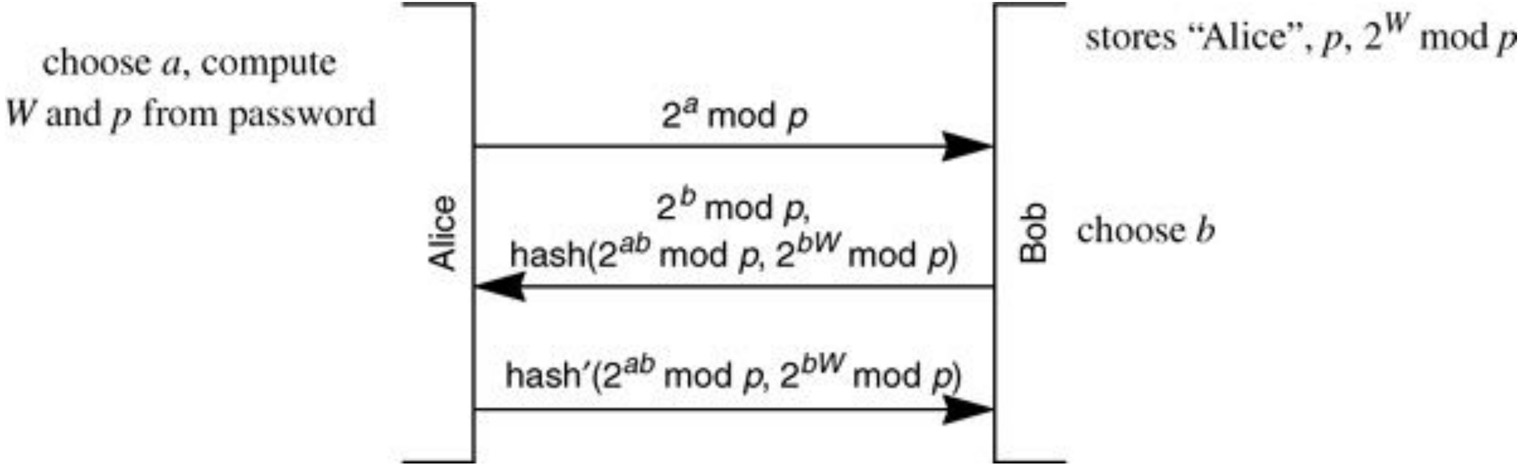
Variants of EKE:

- SPEKE: Simple Password Exponential Key Exchange
  - EKE based on DH where W is used in place of g
  - $g = hash(W)^2 \bmod p$

- PDM: Password Derived Moduli (paper & slides)
  - EKE based on DH where g = 2 and p is derived from the password W

EKE, SPEKE and PDM can be improved or “augmented” by avoiding that Bob stores directly the secret W but only a derived data from W. The idea is to prevent attacker from stealing W from the server (Bob).

Augmented PDM



There exist variants of augment EKE, SPEKE, and PDM that for performance reasons perform RSA in the second part of the protocol to avoid a second DH exponentiation on the server (Bob).

Password-Based Key Derivation Function 2 (PBKDF2)

Very common key derivation function with a “sliding” computational cost, used to reduce (i.e., slow down) vulnerabilities to brute-force attacks.

NIST SP 800-132: Recommendation for Password-Based Key Derivation (PDF)

PBKDF2

DK = PBKDF2(PRF, Password, Salt, c, dkLen)

where:

- DK: derived key
- PRF: a pseudorandom function of two parameters with output length hLen (e.g., a keyed HMAC)
- Password: master password from which a derived key is generated
- Salt: a cryptographic salt
- c: number of iterations desired
- dkLen: desired bit-length of the derived key

For instance, in WPA2: DK = PBKDF2(HMAC−SHA1, passphrase, ssid, 4096, 256)

PBKDF2 (2)

Each hLen-bit block T_i of derived key DK, is computed as follows (+ means string concatenation):

- $DK = T_1 + T_2 + \ldots + T_{dklen/hlen}$
- $T_i = F(Password, Salt, c, i)$

where function F is the XOR of c iterations of chained PRFs:

- $F(Password, Salt, c, i) = U_1 \wedge U_2 \wedge \ldots \wedge U_c$

PBKDF2 (3)

where:

- U_1 = PRF(Password, Salt + INT_32_BE(i))
- U_2 = PRF(Password, U_1)
- ....
- U_c = PRF(Password, U_{c-1})

1Password.com:

SECURITY AND PRIVACY

How PBKDF2 strengthens your Master Password

Learn how 1Password uses Password-Based Key Derivation Function 2 to make it harder for someone to repeatedly guess your Master Password.

Master Password guessing times with hashcat's 4 GPU system

	10000 PBKDF2 iterations (minimum for new keychains)	25000 PBKDF2 iterations (typical for new keychains)	45000 PBKDF2 iterations (high end)
	300,000 guesses/sec	120,000 guesses/sec	66,667 guesses/sec
Password Strength Entropy (in bits)			
39	9 days	23 days	41 days
52	193 years	482 years	867 years
65	1,498,426 years	3,746,064 years	6,742,915 years
78	12 billion years	29,129 billion years	52,433 billion years
90	91 trillion years	227 trillion years	408 trillion years

(link)

PBKDF2

Unfortunately PBKDF2 is today “obsolete”:

- it is not resistant to modern GPU and ASIC attacks

- it has one design flaw: it does not take into account memory usage for performing an attack. It could be valuable to have a KDF that requires a lot of memory to derive a password and thus perform an attack attempt.

One well-known alternative to PBKDF2 is Scrypt: more details here and here. In 2013, a Password Hashing Competition (PHC) was held to develop a more resistant approach. On 20 July 2015 Argon2 was selected as the final PHC winner: details on its algorithm here.

Credits

These slides are based on material from:

- Slides of Prof. D'Amore from CNS 2019-2020
- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. http://www.crypto-textbook.com/
- Charlie Kaufman, Radia Perlman, and Mike Speciner. Network Security - Private Communication in a Public World. Prentice Hall.
- Wikipedia (english version)