Cryptography and Network Security

User authentication and Kerberos

The notes for the lesson “User authentication and Kerberos”: chapters 5.1 to 5.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 5.1 to 5.2

About these notes

These are the notes for the University free n easy video lesson **User authentication and Kerberos** on Cryptography and Network Security: chapters 5.1 to 5.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

5.1 User authentication: passwords, challenge-response, multi-factor

Authentication proves that a person or a device is who it claims to be. The system checks something you know (a password), something you have (a phone or token), or something you are (a fingerprint). Passwords alone are weak: people reuse them and servers leak them, so servers store salted slow hashes and demand a fresh random challenge at every login. Multi-factor authentication combines two or more kinds of proof. Bank apps, UPI payments and office VPNs all use these checks.

> You need hash functions and MACs (an earlier lesson, Public Key Cryptography and Hashing), plus replay and phishing attacks (an earlier lesson, Foundations).

Entity authentication and the three factors

Entity authentication proves that a live party is who it claims to be, right now, in this session. Message authentication proves that one specific message came from the claimed sender and was not altered; MACs and digital signatures do that job (an earlier lesson). A login screen performs entity authentication; every request after login should carry message authentication.

Proof comes from one of three categories, called factors. Knowledge: something you know, such as a password or PIN. Possession: something you hold, such as a phone or smart card.

Inherence: something you are, such as a fingerprint. Two secrets from the same category still count as one factor.

Some syllabi call this topic user authentication; the names cover the same material.

Storing passwords: salt and slow hashes

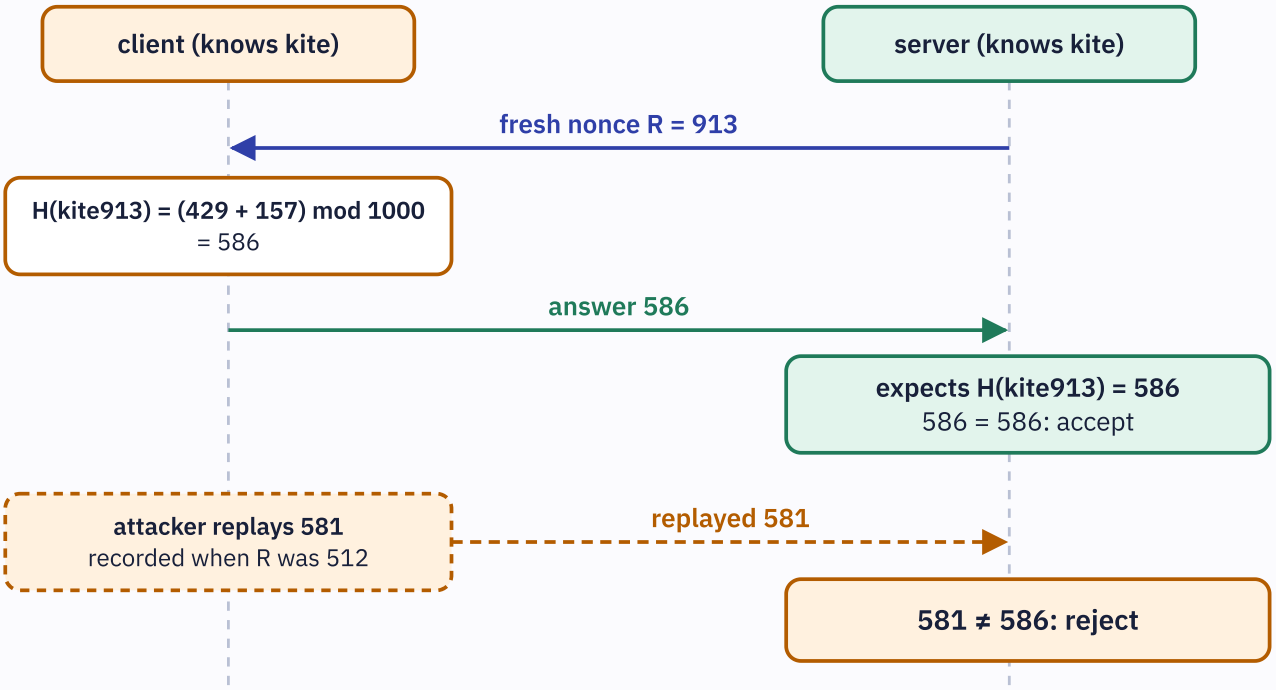
A server never stores plaintext passwords: one leaked database exposes every account, and reuse spreads the damage. A fast hash such as SHA-256 is not enough either, because the attacker hashes a wordlist once or uses precomputed tables.

A salt is a random value generated per user and stored in plain view beside the hash. The server keeps (salt, H(salt ‖ password)), where H is a hash function and ‖ means concatenation. Two users with the same password get different hashes, so precomputation fails and the attacker must guess separately for each account.

1. Generate a random 16-byte salt for this user at registration.
2. Compute h = H(salt ‖ password) with a slow password hash.
3. Store the user id, salt, h and algorithm parameters.
4. At login, read the salt, recompute h' from the typed password, compare in constant time.
5. On failure, return the same error whether the user exists or not.

Three slow hashes dominate: bcrypt, PBKDF2 and Argon2, each with a tunable work factor. In bcrypt the cost is a power of two, so cost 12 means $2^{12} = 4096$ rounds of key setup. PBKDF2 takes an iteration count, often several hundred thousand. Argon2 takes memory, iterations and parallelism; being memory-hard it hurts GPU farms. Tune so one login costs about 0.1 to 0.5 s, and raise the parameters as hardware improves.

Online guessing hits the server and is stopped by rate limits, exponential backoff and CAPTCHA. Lockout after five failures works, but a permanent lock lets anyone deny service to a user, so lock for a fixed time. Offline guessing happens after a leak and is limited only by hash cost and password length.



H(s) = sum of ASCII codes mod 1000: kite = 107 + 105 + 116 + 101 = 429, "913" = 157, "512" = 152.

Figure 5.1.1 The server sends the fresh nonce 913 and the client answers H("kite913") = 586, which the server accepts. The answer 581 recorded under the old nonce 512 does not match and is rejected.

Challenge-response: the fresh nonce

If the client sends the stored password hash to log in, that hash is a password equivalent: an attacker who records one login can replay it. A challenge-response exchange removes this. The server sends a fresh random number and the client's answer depends on it.

Answer = H(password ‖ R)

H is a hash function such as SHA-256, password is the shared secret, R is the server's random nonce, at least 128 bits, and ‖ means concatenation. The answer differs at every login because R differs.

1. The client sends the login name to ask for a challenge.
2. The server picks a fresh R from a cryptographic random generator and ties it to that session, so two logins never share a nonce.
3. The client computes H(password ‖ R) and sends the answer.
4. The server recomputes the expected answer, compares, then discards R.

Replay fails on freshness. Suppose R = 913. An attacker holds H(password ‖ 512) from an earlier session and sends it; the server computes H(password ‖ 913), sees a mismatch and rejects.

With a MAC the answer is MAC_K(R), where K is a key derived from the password.

The server needs the password in usable form to verify such an answer, so this exchange normally runs inside TLS (see 5.4). PAKE protocols such as SPAKE2 and OPAQUE avoid that need; FIDO2 keys sign the challenge with a private key instead.

Multi-factor: OTP, TOTP and hardware tokens

An OTP (one-time password) is valid once and expires in minutes. Sent by SMS it is weak, because SIM swap and network interception exist, but it is still better than a password alone. An app-based TOTP avoids the SMS channel entirely.

$$T = \lfloor t / 30 \rfloor, \quad code = \mathrm{Truncate}(\mathrm{HMAC}(K, T)) \bmod 10^6$$

t is Unix time in seconds, 30 is the time step in seconds, T is the step counter, K is the shared seed, HMAC uses SHA-1 or SHA-256, and truncation keeps 31 bits so the six-digit code is the remainder modulo 10^6.

The server accepts the current step and usually one step either side for clock drift, and refuses a code already accepted in that window. The code space is only 10^6, about 2^{20}, so rate-limiting the verification endpoint matters as much as the number of digits.

Hardware tokens and passkeys (FIDO2, WebAuthn) keep a private key inside the device and sign the server's challenge. The key never leaves the device and the signature is bound to the site's origin, so a phishing copy of the page gains nothing. Only a few syllabi go into those internals.

2FA means exactly two factors from different categories; MFA means two or more. Password plus OTP is 2FA. Password plus a security question is one factor asked twice. Step-up authentication adds a factor when risk rises: browse with a password, but demand an OTP to add a payee or move a large amount.

WORKED EXAMPLE

Question. A server uses the toy hash H(s) = (sum of ASCII codes of s) mod 1000, chosen only so you can compute it by hand, and the user's password is "kite". Today's nonce is R = 913. (a) Compute the client's answer H(password ‖ R). (b) An attacker recorded the answer for an earlier nonce R = 512. Does the server accept it? Give both hash values.

1. ASCII codes of "kite": 107, 105, 116, 101; sum = 429.
2. ASCII codes of "913": 57, 49, 51; sum = 157.
3. H("kite913") = (429 + 157) mod 1000 = 586. The client sends 586.
4. ASCII codes of "512": 53, 49, 50; sum = 152.
5. H("kite512") = (429 + 152) mod 1000 = 581. This is the recorded answer.
6. The server expects 586 but receives 581, so the comparison fails and the login is rejected.

Dry run

PART	ASCII SUM	RUNNING TOTAL MOD 1000
kite	429	429
913	157	586
512	152	581

Answer. The client answers 586, and the replayed 581 is rejected because the fresh nonce 913 changed the expected answer.

WORKED EXAMPLE 2

Question. A TOTP app uses a 30 s time step. At Unix time t = 1799 s, which step counter T does the client use?

1. Divide the time by the step: 1799 / 30 = 59.97.
2. Take the floor: T = ⌊59.97⌋ = 59.
3. The next step starts at t = 60 × 30 = 1800 s, so at 1799 s the counter is still 59.

Answer. The client uses step counter T = 59.

EXAM TRAPS

- Sending H(password) with no fresh challenge makes the hash a password equivalent; a recorded answer logs in again.
- Password plus OTP is 2FA; password plus security question is one factor counted twice. Two passwords are never two factors.
- A salt is public and does not slow a targeted guess. It stops precomputation; the slow hash parameters do the slowing.
- A six-digit OTP has only 10^6 values, about 20 bits. Without rate limiting the endpoint falls to brute force in minutes.
- A permanent lockout is a denial-of-service gift to an attacker. Use short temporary locks with backoff, and compare secrets in constant time.

CHECK YOURSELF

– REMEMBER **Define entity authentication and message authentication, with one example of each.** 2 marks

Entity authentication proves the identity of a live party in the current session, for example a login with password plus OTP. Message authentication proves a given message came from the claimed sender unaltered, for example a MAC on a document.

– REMEMBER **What is a salt, and where is it stored?** 2 marks

A random value generated per user, stored in plain view beside the hash as the pair (salt, H(salt ‖ password)). It makes the same password give different hashes for different users.

– UNDERSTAND **Why is plain SHA-256 a poor way to store passwords, and what fixes it?** 3 marks

SHA-256 is fast, so an attacker with a leaked file tests billions of candidates per second and precomputed tables cover common passwords. The fix is a per-user salt plus a deliberately slow hash such as bcrypt, PBKDF2 or Argon2, tuned to 0.1 to 0.5 s per login.

– APPLY **Same toy hash H(s) = (sum of ASCII codes of s) mod 1000, password "kite", nonce R = 913. Compute the answer the client sends.** 4 marks

ASCII of "kite" sums to 429, ASCII of "913" sums to 157. H = (429 + 157) mod 1000 = 586. The client sends 586.

– APPLY **A TOTP scheme uses a 30 s step. The server accepts the current step and one step either side. At t = 1799 s, which step counters are accepted?** 3 marks

Current step T = ⌊1799/30⌋ = 59. The server accepts 58, 59 and 60, and rejects a code already used inside that window.

– ANALYZE **A bank sends OTPs by SMS. Give two reasons it is weaker than an app-based TOTP, and one control that reduces the risk.** 5 marks

SMS crosses the signalling network, where interception is possible, and the number can be hijacked by SIM swap, so the possession factor passes to the attacker. TOTP codes are computed locally from a seed and never cross the network. Controls: single-use short validity, rate limits on verification, and binding the OTP to the exact amount and payee.

– ANALYZE **A site locks an account permanently after five wrong passwords. Explain the security problem and give a better design.** 4 marks

Anyone who knows a username can lock the owner out, a denial of service against a legitimate user. Better: temporary lockout with exponential backoff, per-IP and per-account rate limits, CAPTCHA, user alerts, and a recovery path that needs a second factor.

BEYOND THE SYLLABUS

Not for the exam. Password-authenticated key exchange (SPAKE2, OPAQUE) lets a client prove knowledge of a password without the server holding a usable password equivalent, and standards bodies now describe it for TLS. FIDO2/WebAuthn passkeys replace the shared secret with a per-site public key pair in a secure element, removing phishing and server-side password files together. NIST SP 800-63B guides current practice: no forced periodic rotation, screening against breach lists, and long passphrases allowed.

5.2 Kerberos authentication

Kerberos is a network authentication protocol built around one trusted server, the Key Distribution Center (KDC). A user proves identity once, and the KDC hands out time-limited encrypted tickets. Each ticket is a small blob that a service can decrypt and trust without calling the KDC or asking for a password, so passwords never travel on the wire. It is used where many users share many servers inside one organisation: campus logins, Active

Directory domains, NFS file shares, Hadoop clusters. This chapter follows Kerberos V5, the version still in use.

> You need: 5.1 for challenge–response and freshness, and an earlier lesson symmetric encryption for block ciphers and key distribution.

Principals, realms and the KDC

Kerberos starts with a counting problem. If M users must talk to N servers with shared secret keys, every pair needs its own key, so roughly M × N keys must be created and stored. Kerberos adds one trusted third party, the Key Distribution Center (KDC). Each user shares one long-term key with the KDC, and each server shares one long-term key with the KDC. That is M + N keys instead.

keys = M + N

M is the number of user principals, each sharing one long-term key with the KDC; N is the number of service principals, each sharing one long-term key with the KDC.

A principal is any named entity Kerberos authenticates. A user principal looks like alice@CSE.AC.IN. A service principal looks like nfs/host1.cse.ac.in@CSE.AC.IN, with an optional instance after the slash. A realm is the set of principals run by one KDC, written in uppercase, for example CSE.AC.IN. A ticket issued in another realm needs cross-realm trust, which is advanced material.

The long-term key comes from the user's password through a string-to-key function, or from a keytab file for a service. The KDC keeps a copy of every principal's long-term key. A session key is a fresh random symmetric key for one client–server conversation, alive only as long as its ticket.

The KDC runs two logical services, both on port 88. The Authentication Server (AS) issues the Ticket Granting Ticket (TGT). The Ticket Granting Server (TGS) issues service tickets.

Three exchanges: AS, TGS, AP

The client talks to the KDC twice and to the server once. Each exchange uses a different key, and each reply is mostly a blob the client cannot read.

1. AS-REQ (client → AS): client name, the TGS name, a nonce (a random number used once). When the KDC requires pre-authentication (common policy, not a core V5 requirement), the first request is refused with KDC_ERR_PREAUTH_REQUIRED and the client resends with a pre-authenticator: the current timestamp encrypted under its long-term key.

2. AS-REP (AS → client): two parts. Part one is encrypted under the client's long-term key and carries the session key K(c,tgs), the TGT lifetime and the nonce. Part two is the TGT itself, encrypted under the TGS's long-term key, holding the client name, K(c,tgs) and the ticket lifetime.
3. The client decrypts part one with its password-derived key, recovers K(c,tgs), and stores the TGT unread.
4. TGS-REQ (client → TGS): the TGT, the wanted service name, a nonce, and an authenticator — client name plus current timestamp, encrypted under K(c,tgs).
5. TGS-REP (TGS → client): the TGS decrypts the TGT with its own key, recovers K(c,tgs), decrypts the authenticator, and checks the timestamp and its replay cache. It replies with a service ticket encrypted under the server's long-term key and holding K(c,s), plus K(c,s) encrypted under K(c,tgs).
6. AP-REQ (client → server): the service ticket plus a fresh authenticator encrypted under K(c,s).
7. AP-REP (server → client): the server decrypts the ticket with its own key, gets K(c,s), decrypts the authenticator, and checks timestamp and replay cache. An optional reply carries the authenticator timestamp plus one, encrypted under K(c,s).

The AP-REP gives mutual authentication: only the real server can read K(c,s) and return the incremented timestamp. The ticket is reusable until it expires; the authenticator is made fresh for every request.

Timestamps, clock skew and the opaque TGT

Kerberos uses time instead of a challenge nonce between client and server. A server accepts an authenticator when $|t_{client} - t_{server}| \le skew$, where t_{client} is the timestamp inside the authenticator, t_{server} is the server's clock at arrival, and skew is the allowed difference, 5 minutes by default. Outside the window the server returns a clock skew too great error.

Replay is the risk. An attacker who copies an AP-REQ can resend it inside the window. Servers keep a replay cache of authenticators seen within the window and drop duplicates. The window also bounds how long that cache must be kept.

The client cannot read its TGT because it is encrypted under the TGS's long-term key, which only the KDC knows. The client carries it as an opaque token. This is deliberate: the ticket is self-protecting, so the TGS trusts its contents without a database lookup, and the client cannot change the session key or stretch the lifetime. The client holds K(c,tgs) separately in the readable AS-REP part.

Pre-authentication guards the AS exchange. Without it, anyone could send an AS-REQ naming alice, receive a blob encrypted under alice's key, and guess passwords offline. In V5 pre-authentication is an extensible framework, not a core requirement: many KDCs first answer an AS-REQ without a pre-authenticator with KDC_ERR_PREAUTH_REQUIRED, and only the resent request carries the encrypted timestamp. A wrong password fails at once and the attempt can be logged.

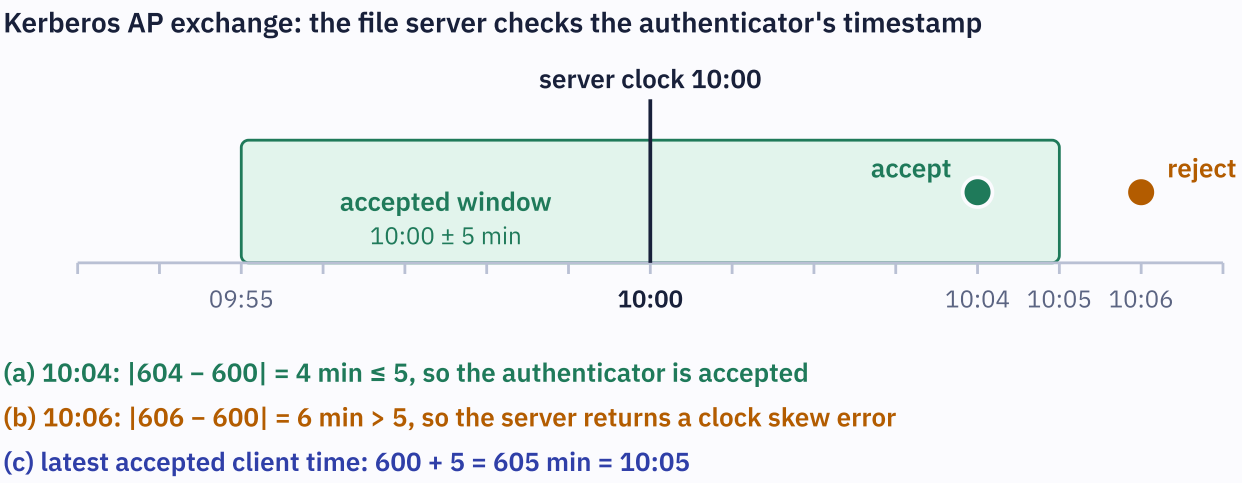


Figure 5.2.1 With the server clock at 10:00 and a 5-minute limit, timestamps from 09:55 to 10:05 pass. The authenticator stamped 10:04 is accepted and the one stamped 10:06 is rejected.

WORKED EXAMPLE

Question. The file server's clock reads 10:00. The realm's clock skew limit is 5 minutes. (a) The client sends an authenticator stamped 10:04 by its own clock. Is it accepted? (b) A later request carries 10:06. Is it accepted? (c) What is the latest client timestamp the server would still accept?

1. Convert all times to minutes after midnight. Server: $10 \times 60 + 0 = 600$ min. First client timestamp: $10 \times 60 + 4 = 604$ min. Second client timestamp: $10 \times 60 + 6 = 606$ min.
2. Case (a): $|t_{client} - t_{server}| = |604 - 600| = 4$ min.
3. Compare with the limit: 4 min ≤ 5 min, so the authenticator is accepted.
4. Case (b): $|606 - 600| = 6$ min.
5. 6 min > 5 min, so the server rejects it with a clock skew error.
6. Case (c): the largest accepted timestamp satisfies $t_{client} - t_{server} \le 5$ min, so $t_{client} \le 600 + 5 = 605$ min = 10:05.

Answer. (a) Accepted, the difference is 4 minutes, within the 5-minute limit. (b) Rejected, the difference is 6 minutes. (c) The latest accepted client timestamp is 10:05.

EXAM TRAPS

- The client never reads its own TGT: it is encrypted under the TGS's long-term key. Only the AS-REP part under the client's key is readable.
- A ticket is not an authenticator. The ticket is reusable and encrypted under the target's long-term key; the authenticator is fresh per request and encrypted under the session key.
- The timestamp check is loose in both directions and uses ≤, not <. A timestamp exactly 5 minutes away from the server clock is still accepted.
- Ticket lifetime and clock skew are different numbers. Skew is about unsynchronised clocks; lifetime is about how long the ticket stays valid.
- Pre-authentication is not the same as mutual authentication. Pre-auth protects the AS exchange from offline password guessing; the AP-REP proves the server knows the session key.

CHECK YOURSELF

– REMEMBER **Define principal and realm in Kerberos V5.** 2 marks

A principal is any named entity Kerberos can authenticate, such as alice@CSE.AC.IN or nfs/host1@CSE.AC.IN. A realm is the set of principals administered by one KDC, written in uppercase, for example CSE.AC.IN.

– REMEMBER **Name the two services inside the KDC and say what each one issues.** 2 marks

The Authentication Server (AS) issues the Ticket Granting Ticket (TGT). The Ticket Granting Server (TGS) issues service tickets for individual servers.

– UNDERSTAND **Why does the AS-REP contain two parts encrypted under different keys?** 3 marks

One part is encrypted under the client's long-term key so only the client can read K(c,tgs) and the nonce. The other part is the TGT, encrypted under the TGS's long-term key so only the TGS can read the client name and session key. Two recipients, two keys.

– UNDERSTAND **Why can the client not read its own TGT, and why is that not a problem for the client?** 3 marks

The TGT is encrypted under the TGS's long-term key, which the client does not hold. The client does not need to read it; it only forwards the blob to the TGS, and it already received its session key separately in the part encrypted under its own key.

– APPLY **The server clock reads 09:58 and the skew limit is 5 minutes. An authenticator arrives stamped 10:04. Is it accepted?** 4 marks

Server = 598 min, client = 604 min. $|604 - 598| = 6$ min. $6 > 5$, so the server rejects it with a clock skew error.

– APPLY **Which key encrypts the service ticket in the TGS-REP, and which key encrypts the authenticator in the AP-REQ?** 4 marks

The service ticket is encrypted under the server's long-term key so only that server can open it. The authenticator in the AP-REQ is encrypted under the client–server session key K(c,s) taken from the ticket.

– ANALYZE **What attack does pre-authentication prevent, and why does the defence work?** 5 marks

Without pre-authentication anyone can send an AS-REQ for a named user and receive a blob encrypted under that user's password-derived key, then guess passwords offline at high speed. Requiring the client to encrypt a timestamp under its key first means a wrong password produces a blob the KDC cannot decrypt, so the AS fails the request at once and can log or rate-limit the attempt.

– ANALYZE **An attacker gains full control of the KDC. What can the attacker do, and what limits the damage?** 6 marks

The KDC holds every principal's long-term key, so the attacker can forge TGTs and service tickets for any user or server, and impersonate the TGS. The real bounds: already-issued tickets expire; short lifetimes limit abuse. The attacker must stay hidden or re-forge after expiry. Recovery works by rotating keys — the krbtgt key and every principal's key — which invalidates everything the attacker minted. Detection and hardening are the practical controls.

BEYOND THE SYLLABUS

Not for the exam. Real deployments rarely stop at plain V5: Active Directory, Hadoop and NFS all run Kerberos underneath, with extensions layered on top. Public-key Kerberos lets a client authenticate with a certificate instead of a password-derived key, and cross-realm referrals let one realm's KDC point a client to another realm's KDC. Armoring wraps the AS exchange in a protected tunnel to defeat offline dictionary attacks on weak passwords, and constrained delegation lets a service act for a user only towards named targets. Search for the Kerberos V5 specification (RFC 4120), PKINIT, and Kerberos delegation if you meet this in industry.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Password (verifier)	A shared secret string stored by the server only as a salted hash such as H(salt‖password), never as plaintext.	5.1
Challenge-response	Server sends a fresh random number (nonce); client proves the secret by hashing or encrypting it with the nonce, so an old answer is useless.	5.1
Multi-factor authentication (MFA)	Proving identity with two or more different factors: something you know, something you have, something you are.	5.1
Key Distribution Center (KDC)	The trusted server in Kerberos that holds a master key per user and per service and issues tickets.	5.2
Ticket-granting ticket (TGT)	A time-limited ticket encrypted under the KDC's secret key that the client shows to get service tickets.	5.2

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– REMEMBER **Define entity authentication and message authentication. Explain, with a message exchange, how a challenge-response protocol defeats a replay of an old password response.** 10 marks

Entity authentication: proof of a live party's identity now; Message authentication: proof that a message came from the claimed source and was not altered; Challenge: server sends nonce; client returns hash of nonce, password and shared salt; Replay fails because the nonce differs each run; mention time-stamping and sequence numbers as alternatives; Contrast with sending a plaintext or even a hashed static password, which is replayable.

– UNDERSTAND **Describe the Kerberos authentication exchange. Explain the roles of the KDC, the ticket-granting ticket and the service ticket, and why passwords never cross the network.** 12 marks

KDC split into AS and TGS conceptually; Step 1: client authenticates to AS, gets TGT encrypted under KDC key; Step 2: client shows TGT plus an authenticator to get a service ticket; Step 3: client presents service ticket to the application server; Tickets carry a session key and a lifetime; the timestamp in the authenticator bounds replay; Password is used only locally to derive a key, so it is never transmitted.

– UNDERSTAND **Explain multi-factor authentication and describe how a FIDO-style security key or TOTP app changes the threat model compared with SMS one-time codes.** 8 marks

Factors: knowledge, possession, inherence; MFA needs different kinds, not two of the same; Passwords alone fall to phishing, reuse and credential-stuffing breaches; SMS OTP is vulnerable to SIM swap, interception and real-time relay phishing; TOTP app removes the telecom link but is still phishable in real time; Hardware authenticator binds the response to the site origin, so a proxy phishing page gets a useless signature.

Glossary (terms used in this lesson)

Authentication server The RADIUS server that checks credentials and returns accept or reject with keying material.

Nonce A value used once, so that a fresh run of a protocol cannot be answered with a recorded old reply.

Phishing A message or site that impersonates a trusted party to make the user hand over credentials or data.

Salt A random value stored with a hashed password so identical passwords hash differently and precomputed tables fail.

Session key A short-lived symmetric key used for one connection and then discarded.

Ticket In Kerberos, a block encrypted under a server's secret key that carries the client name and a session key.

VPN A virtual private network: an authenticated, encrypted tunnel carrying private traffic over a public network.

Sources and further reading

Textbooks

- William Stallings, *Cryptography and Network Security: Principles and Practice*: Chapters on authentication applications, IP security, transport-level security, electronic mail security, wireless security and web security.
- James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach*: The security chapter covering TLS, IPsec, wireless security and firewalls.
- Andrew S. Tanenbaum and David J. Wetherall, *Computer Networks*: The security chapter, including authentication, email and web security.
- Charlie Kaufman, Radia Perlman and Mike Speciner, *Network Security: Private Communication in a Public World*: Kerberos, PKI, IPsec and TLS chapters.

Open web resources

Each link below was opened by the book script on 10 Nov 2026 and returned a page; the pages were not read line by line.

- OWASP Top Ten: owasp.org/www-project-top-ten
- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: www.rfc-editor.org/rfc/rfc8446
- RFC 4301: Security Architecture for the Internet Protocol: www.rfc-editor.org/rfc/rfc4301
- RFC 4120: The Kerberos Network Authentication Service (V5): www.rfc-editor.org/rfc/rfc4120
- RFC 6376: DomainKeys Identified Mail (DKIM) Signatures: www.rfc-editor.org/rfc/rfc6376
- MDN Web Docs: Web security: developer.mozilla.org/en-US/docs/Web/Security

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Cryptography and Network Security (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.