1 Authentifizierung

1.1 Grundlagen

1.1.1 Begriffsdefinitionen

Identität

Eine Menge von Attributen, die eine Entität beschreiben (z.B. Name, Geburtsdatum, Wohnort).

Authentisierung

Die **Bereitstellung** von Unterlagen oder Nachweisen, die es ermöglichen, die Identität zu prüfen (z.B. das Vorzeigen eines Personalausweises).

Authentifikation / Authentifizierung

Die **Prüfung** und Echtheitsbezeugung der vorgelegten Unterlagen zur Identitätsfeststellung (z.B. der Vergleich des Fotos auf dem Ausweis mit der Person).

Autorisierung

Die Gewährung oder Verwehrung von Rechten an eine (authentifizierte) Entität.

1.1.2 Drei Ansätze der Authentisierung

Ziel ist die Identifikation von Subjekten (Menschen, Systeme, Dienste) und der Nachweis ihrer Identität.

- **Durch Wissen:** z.B. PIN, Passwort, kryptographischer Schlüssel.
- **Durch Besitz:** z.B. Smartcard, Token, SIM-Karte.
- **Durch Merkmale:** z.B. Biometrie (physiologische Eigenschaften).

1.2 Authentisierung durch Besitz

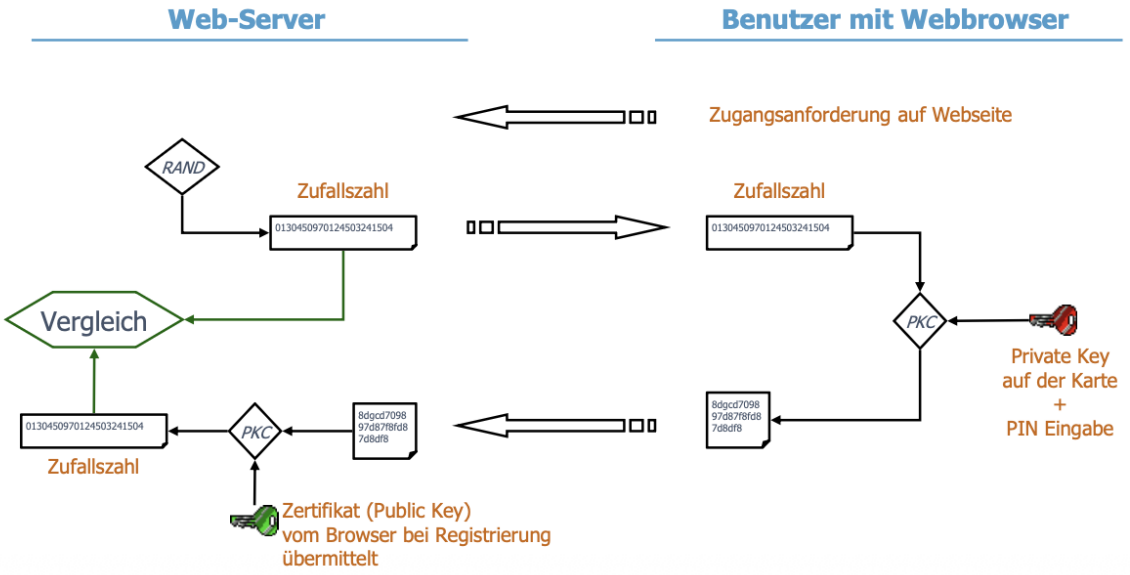
- **Statische Tokens:** Ein gespeichertes Geheimnis (z.B. privater Schlüssel) wird direkt genutzt.
- **Dynamische Tokens:** Das Geheimnis wird zur Berechnung von Authentifizierungs-Informationen genutzt (z.B. Challenge-Response).
- **Hardware-Tokens:** Schlüssel, SmartCard, Transponder.
- **Software-Tokens:** Cookie, Client-Zertifikat.

1.2.1 Beispiel: ATHENE KARTE (SmartCard)

- Besitzt einen **Kryptoprozessor** (z.B. CardOS 4.3b).
- Träger eines privaten Schlüssels (z.B. 2048 Bit RSA) und eines öffentlichen Zertifikats (digitale ID).
- Der **private Schlüssel ist nicht auslesbar** und zusätzlich durch eine PIN geschützt.

1.2.2 Challenge-Response-Verfahren (mit Kryptoprozessor)

Dies ist ein dynamisches Verfahren, das den privaten Schlüssel nutzt, ohne ihn preiszugeben.



1. **Benutzer (mit Webbrowser)** initiiert eine Zugangsanforderung auf einem Web-Server.
2. **Web-Server** generiert eine **Zufallszahl** (die "Challenge") und sendet sie an den Benutzer.
3. **Benutzer** gibt seine PIN ein, um den Kryptoprozessor der Karte freizuschalten. Die Karte "signiert" die Zufallszahl mit dem **privaten Schlüssel**.
4. **Benutzer** sendet die signierte Zufallszahl zurück an den Server.
5. **Web-Server** nutzt das **öffentliche Zertifikat** (Public Key) des Benutzers (das er z.B. bei einer Registrierung erhalten hat), um die Signatur zu prüfen.
6. Stimmt die verifizierte Zufallszahl mit der ursprünglich gesendeten überein, ist der Benutzer authentifiziert.

1.2.3 Probleme

- **Diebstahl:** Offensichtliches Problem.
- **Gegenmaßnahme:** Sicherung des Tokens durch ein zusätzliches Merkmal, z.B. Wissen (PIN für Hardware-Crypto, Passwort für Software-Crypto) oder 2. Faktor.
- **Extraktion der Schlüssel:** Angriffe auf die Token-Hardware.
- **Methoden:** Schwachstellen in der Firmware oder **Side-Channel-Angriffe** (z.B. Monitoring des Stromverbrauchs), um den privaten Schlüssel auszulesen.

1.3 Authentisierung durch Merkmale (Biometrie)

1.3.1 Anforderungen an biometrische Merkmale

- **Universalität:** Jede Person besitzt das Merkmal.
- **Eindeutigkeit:** Merkmal ist für jede Person verschieden.
- **Beständigkeit:** Merkmal ist (weitgehend) unveränderlich.
- **Quantitative Erfassbarkeit:** Messbar mittels Sensoren.
- **Performanz:** Genauigkeit und Geschwindigkeit der Erfassung/Prüfung.
- **Akzeptanz:** Benutzer müssen bereit sein, das Merkmal zu nutzen.

- **Fälschungssicherheit:** Schutz gegen Angriffe.

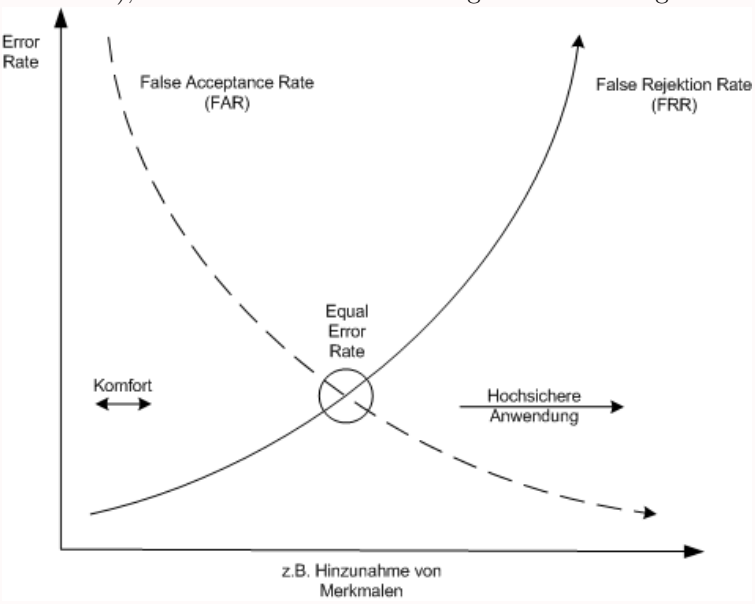
1.3.2 Prozesse und Fehlerraten

- **Enrollment:** Erstmalige Registrierung eines Benutzers und Erfassung seines Merkmals (Template).
- **Verifikation:** Erneute Erfassung und Vergleich mit dem gespeicherten Template.
- **Biometrie ist immer fehlerbehaftet** (ein statistischer Test).

Fehlerraten

- **False Acceptance Rate (FAR):** Ein Unberechtigter wird fälschlicherweise authentifiziert. **(Sicherheitsproblem!)**
- **False Rejection Rate (FRR):** Ein Berechtigter wird fälschlicherweise abgewiesen. **(Benutzbarkeits-/Akzeptanzproblem!)**
- **Equal Error Rate (EER):** Der Punkt, an dem FAR = FRR.

Man muss für den Anwendungsfall abwägen: Eine hochsichere Anwendung optimiert auf eine niedrige FAR (auf Kosten der Bequemlichkeit), eine komfortable Anwendung auf eine niedrige FRR.



1.3.3 Beispiel: Fingerabdruck

- Es wird nicht das Bild gespeichert, sondern ein Template aus **Minutien** (Verzweigungen, Endpunkte, etc.) mit relativen Koordinaten und Winkeln.
- Problem: Schlechte Abdrücke können zu fehlenden Minutien führen (-¿ höhere FRR).

1.3.4 Probleme der Biometrie

- **Datenschutz:** Biometrische Merkmale können ”intrusiv” sein und sensible Daten enthüllen (z.B. Venenmuster, DNA -¿ Gesundheitsdaten).
- **Speicherung:** Es sollten Referenzdaten (Templates) gespeichert werden, aus denen das Merkmal nicht rekonstruiert werden kann.
- **Keine oder begrenzte Widerrufbarkeit:** Ein kompromittierter Fingerabdruck (oder Iris, DNA) kann nicht einfach ”gesperrt” und ersetzt werden wie ein Passwort.
- **Kompromittierung:** Wenn eine Kopie erstellt werden kann (z.B. von einem Lesegerät gestohlen), wird die Authentifizierung durch *Merkmal* (”Wer bin ich?”) zu einer Authentifizierung durch *Wissen* (”Wie sieht Merkmal X aus?”).

- **Gegenmaßnahme: Lebendverifikation** (Liveness Detection) prüft, ob ein echter Finger (keine Plastik-Kopie) aufliegt.

1.4 Authentisierung durch Wissen

1.4.1 Passwörter

Gängigste Methode. Werden (idealerweise) nicht im Klartext, sondern als **Hash-Wert** gespeichert (z.B. in `/etc/shadow` (Linux) oder via LSASS (Windows)).

Evolution der Passwort-Authentifizierung

1. **Plaintext-Übertragung:** Passwort wird im Klartext gesendet (z.B. `telnet`, `ftp`).
2. **Problem:** Passiver Angreifer (Sniffer) im Netz sieht alle Passwörter.
3. **Übertragung mit TLS:** Der Kanal ist geschützt (z.B. HTTPS).
4. **Problem:** Wenn der Server das Passwort im Klartext in seiner Datenbank speichert, erlangt ein Angreifer bei einem Datenbank-Leak alle Passwörter.
5. **Server speichert Hash:** Server speichert $H_{ID} := h(P_{ID})$. Beim Login sendet der Nutzer P_{ID}, der Server berechnet $h(P_{ID})$ und vergleicht es mit dem gespeicherten H_{ID}.
6. **Problem: Rainbow-Tables.** Da h("Passwort123") für alle Nutzer gleich ist, kann ein Angreifer eine Tabelle mit Hashes für Millionen gängiger Passwörter vorab berechnen und die gehashte Datenbank sehr schnell knacken.
7. **Server speichert Hash mit Salt:** Server speichert $H_{ID} := h(P_{ID}, s_{ID})$, wobei s_{ID} ein einzigartiger, zufälliger **Salt** pro Nutzer ist (wird mit H_{ID} gespeichert).
8. **Vorteil:** $h(\text{"Passwort123"}, s_1) \neq h(\text{"Passwort123"}, s_2)$.
9. **Eine Rainbow-Table muss nun für jeden Nutzer (jeden Salt) separat erstellt werden**, was den Geschwindigkeitsvorteil zunichte macht und den Angriff stark verlangsamt.

Passwort-Manager Da man für unterschiedliche Dienste unterschiedliche, starke Passwörter nutzen soll, ist das Auswendiglernen unmöglich.

- **Lösung:** (Lokale) Passwort-Manager, die mit einem starken Masterpasswort geschützt sind.

1.4.2 Challenge-Response-Verfahren (CHAP)

Authentisierung durch Wissen, ohne das "Wissen" (Passwort) zu übertragen. Nutzt symmetrische Kryptographie (HMAC).

- **Voraussetzung:** Alice (Nutzer) und Bob (Server) teilen ein Geheimnis (P_{ID}, z.B. das Passwort).
- **Ablauf (CHAP):**
  1. Alice $\rightarrow$ Bob: ID
  2. Bob $\rightarrow$ Alice: $RAND$ (Zufallszahl, "Challenge")
  3. Alice $\rightarrow$ Bob: $c = \text{HMAC}(P_{ID}, RAND)$
  4. Bob prüft: Berechnet $c' = \text{HMAC}(P_{ID}, RAND)$ und testet, ob $c' == c$.
- **Probleme:**
  - Der Klartextraum für $RAND$ muss groß sein, sonst **Replay-Attacke** (Angreifer kann mehrfach genutzte Challenges korrekt beantworten).
  - Server muss P_{ID} im Klartext kennen. Speichert er stattdessen $h(P_{ID})$, braucht der Angreifer bei einem Leak auch nur noch den Hash (und nicht das Passwort), um sich zu authentifizieren.
  - Schützt nur die Authentisierung, nicht den restlichen Kommunikationskanal (Integrität).

1.5 Single Sign On (SSO)

- **Problem:** ”Passwort-Müdigkeit” – zu viele Dienste erfordern eigene Passwörter.
- **Definition:** Eine Authentisierungsmethode, die es einem Benutzer ermöglicht, sich mit **einem einzigen Satz** von Anmeldeinformationen bei **mehreren unabhängigen** Softwaresystemen anzumelden.
- **Idee:** Ein zentraler, vertrauenswürdiger **Provider** bestätigt die Identität des Nutzers gegenüber allen anderen **Services**.
- **Vorteile:** Weniger Passwörter (nur ein starkes nötig), erhöhte Sicherheit (wenn gut implementiert), Komfort, bessere Kontrolle.
- **Nachteil: Single Point of Failure.** Wird das SSO-Login kompromittiert, hat ein Angreifer Zugriff auf *alle* verbundenen Dienste.

1.5.1 Kerberos (Ein SSO-Protokoll)

- **Ziele:** Authentifizierung von *Principals* (Benutzer, Server), Austausch von Sitzungs-Schlüsseln, SSO innerhalb einer administrativen Domäne (*Realm*).
- **Design:**
  - Pro *Realm* ein **Key Distribution Center (KDC)**.
  - **KDC = Authentication Server (AS) + Ticket Granting Server (TGS)**.
  - Basiert auf **Pre-Shared Secrets**: Das KDC kennt einen geheimen Schlüssel (K) für jeden Principal in seinem Realm (z.B. K_{Bob}, K_{TGS}, K_{SMB}). Für Benutzer wird K_{Bob} aus deren Passwort-Hash generiert.

Kerberos-Ablauf (vereinfacht) Ziel: Benutzer Bob (C) möchte auf den SMB-Server (S) zugreifen.

1. **Login + TGT-Anfrage:**
   - Bob gibt sein Passwort ein. Client C generiert $K_{Bob} = \text{Hash(Passwort)}$.
   - $C \rightarrow AS$: $(K_{Bob}(\text{timestamp}), \text{Bob}, \text{TGS})$
     (Bob bittet den AS um ein Ticket für den TGS, authentifiziert sich mit einem verschlüsselten Timestamp).
2. **AS-Antwort (TGT):**
   - $AS \rightarrow C$: $\{K_{Bob,TGS}\}_{K_{Bob}} + \{TGT\}_{K_{TGS}}$
   - Der AS prüft den Timestamp. Wenn gültig:
   - Er sendet den **Sitzungsschlüssel** für C und TGS ($K_{Bob,TGS}$), verschlüsselt mit Bobs Schlüssel (K_{Bob}).
   - Er sendet das **Ticket Granting Ticket (TGT)**, welches ($K_{Bob,TGS}, \text{Bob}, ...$) enthält, alles verschlüsselt mit dem geheimen Schlüssel des TGS (K_{TGS}). **Der Client kann das TGT nicht lesen.**
3. **Service-Ticket-Anfrage:**
   - $C \rightarrow TGS$: $\{A_{Bob}\}_{K_{Bob,TGS}} + \{TGT\}_{K_{TGS}} +$ ”SMB”
   - C entschlüsselt $\{K_{Bob,TGS}\}_{K_{Bob}}$, um den Sitzungsschlüssel $K_{Bob,TGS}$ zu erhalten.
   - C erstellt einen *Authenticator* $A_{Bob} =$ (Bob, IP, timestamp) und verschlüsselt ihn mit $K_{Bob,TGS}$.
   - C sendet den Authenticator, das (unlesbare) TGT und den Namen des Zieldienstes (”SMB”) an den TGS.
4. **TGS-Antwort (Service Ticket):**
   - $TGS \rightarrow C$: $\{K_{Bob,SMB}\}_{K_{Bob,TGS}} + \{T_{Bob,SMB}\}_{K_{SMB}}$
   - TGS entschlüsselt das TGT (mit K_{TGS}) und den Authenticator (mit $K_{Bob,TGS}$) und prüft sie.
   - Er generiert einen neuen Sitzungsschlüssel für Bob und den SMB-Server ($K_{Bob,SMB}$).
   - Er sendet $K_{Bob,SMB}$, verschlüsselt mit $K_{Bob,TGS}$.

- Er sendet das **Service Ticket** ($T_{Bob,SMB}$), welches ($K_{Bob,SMB}$, Bob, ...) enthält, alles verschlüsselt mit dem geheimen Schlüssel des SMB-Servers (K_{SMB}).

5. **Zugriff auf Dienst:**
   - $C \rightarrow SMB$: $\{A'_{Bob}\}_{K_{Bob,SMB}} + \{T_{Bob,SMB}\}_{K_{SMB}}$
   - C entschlüsselt $K_{Bob,SMB}$.
   - C erstellt einen *neuen* Authenticator A'_{Bob} und verschlüsselt ihn mit $K_{Bob,SMB}$.
   - C sendet den neuen Authenticator und das (unlesbare) Service Ticket an den SMB-Server.
6. **Verifikation:**
   - Der SMB-Server entschlüsselt das Service Ticket (mit K_{SMB}) und den Authenticator (mit $K_{Bob,SMB}$) und prüft sie.
   - Wenn alles gültig ist, ist Bob authentifiziert und Bob und SMB teilen sich den Sitzungsschlüssel $K_{Bob,SMB}$ für die weitere Kommunikation.

Kerberos-Angriffe

- **Pass the Hash:** Der Angreifer stiehlt den Hash K_{Bob} (z.B. aus dem LSASS-Prozessspeicher). Da K_{Bob} das "Geheimnis" ist, das Kerberos verwendet, kann der Angreifer **Schritt 1** des Protokolls direkt ausführen und ein TGT erhalten, **ohne das Klartextpasswort zu kennen.**
- **Golden Ticket:** Der Angreifer kompromittiert das KDC und stiehlt den geheimen Schlüssel des TGS selbst (den Hash des `KRBTGT`-Kontos). Mit diesem Schlüssel kann der Angreifer **offline** ein TGT für **jeden beliebigen Benutzer** (z.B. Administrator) mit **beliebiger Gültigkeitsdauer** fälschen. Dies gewährt dem Angreifer uneingeschränkten, persistenten Zugriff auf die gesamte Domäne.

1.6 Autorisierung (Zugriffskontrollmodelle)

Nach der Authentifizierung (Wer bist du?) folgt die Autorisierung (Was darfst du?).

- **Referenzmonitor:** Ein abstraktes Konzept, das jede Anfrage eines **Subjekts** (Prozess, Benutzer) auf ein **Objekt** (Datei, Speicher) prüft und anhand einer Rechtedatenbank entscheidet (gewährt / abgelehnt).
- **Schutzziele:** Integrität und Vertraulichkeit.

1.6.1 Discretionary Access Control (DAC)

	Datei1	Datei2	Datei3	Prozess1	Prozess2
Prozess1	{ read, write }		{ read, write }		{ send, receive }
Prozess2				{ send, receive }	
Prozess3		{ owner, execute }		{ signal }	

- **Definition:** Der **Eigentümer** eines Objekts ist für die Vergabe von Zugriffsrechten verantwortlich ("at his discretion").
- **Modell: Zugriffsmatrix** ($M : S \times O \rightarrow \mathcal{P}(R)$), die Rechte von Subjekten S auf Objekte O abbildet.

- **Implementierung (Speicherung der Matrix):**
  - **Spaltenweise (Access Control Lists, ACLs):** Jedes *Objekt* hat eine Liste, die alle Subjekte und deren Rechte aufführt. (z.B. Dateiberechtigungen in Windows/Linux).
  - *Vorteil:* Effizient zu bestimmen: "Wer darf auf diese Datei zugreifen?"
  - **Zeilenweise (Capability Lists, CLs):** Jedes *Subjekt* hat eine Liste (Capability) mit allen Objekten, auf die es zugreifen darf, und den jeweiligen Rechten.
  - *Vorteil:* Effizient zu bestimmen: "Worauf darf dieser Benutzer zugreifen?"
- **Nachteile:** Keine formalen Garantien für Informationsfluss (Problem "Trojanisches Pferd": Ein Programm, das im Kontext des Nutzers läuft, kann dessen Rechte missbrauchen, z.B. eine Datei kopieren und unerlaubt weitergeben).

1.6.2 Role-based Access Control (RBAC)

- **Definition:** Berechtigungen werden nicht direkt an Benutzer, sondern an **Rollen** (z.B. "Arzt", "Buchhalter", "Admin") vergeben. Benutzer werden dann diesen Rollen zugewiesen.
- **Vorteile:** Bildet Organisationsstrukturen gut ab; erleichtert Prinzipien wie "Need-to-Know" und "Separation-of-Duty".

1.6.3 Mandatory Access Control (MAC)

- **Definition:** Systembestimmte (regelbasierte) Festlegung von Sicherheitseigenschaften. **Systemregeln dominieren (überschreiben) Benutzerwünsche** (DAC-Einstellungen).
- **Ziel:** Kontrolle des Informationsflusses.

Beispiel: Bell-La Padula (BLP) Modell Ein MAC-Modell, das sich auf **Vertraulichkeit** (Confidentiality) konzentriert.

- **Konzept:** Subjekte und Objekte erhalten **Sicherheitsklassen (Labels)**, z.B. (Level, {Kategorien}).
- **Level:** Haben eine totale Ordnung (z.B. unklassifiziert $<$ vertraulich $<$ geheim $<$ streng geheim).
- **Kategorien:** Eine Menge von Zuständigkeiten (z.B. {Buchhaltung}, {Forschung}).
- **Dominanz ($\geq$):** Ein Subjekt S dominiert ein Objekt O ($SC(S) \geq SC(O)$), wenn S ein höheres oder gleiches Level hat **und** die Kategoriemenge von O eine Teilmenge der Kategoriemenge von S ist ($L_S \geq L_O \wedge C_O \subseteq C_S$).

Bell-La Padula Regeln

- **1. Simple-Security-Property (No-Read-Up):**
- Ein Subjekt S darf ein Objekt O nur **lesen**, wenn $SC(S) \geq SC(O)$ (Subjekt dominiert Objekt).
- *(Ein "geheimer" Sekretär darf keine "streng geheimen" Bilanzdaten lesen).*
- **2. $*$-Property (No-Write-Down):**
- Ein Subjekt S darf ein Objekt O nur **schreiben**, wenn $SC(S) \leq SC(O)$ (Objekt dominiert Subjekt).
- *(Ein "strenger geheimer" CEO darf Bilanzdaten nicht in eine "unlassifizierte" Website schreiben und so leaken).*

- **Nachteile BLP:** Informationen fließen sukzessive nur "nach oben". Erlaubt "blindes Schreiben" (Schreiben in ein Objekt, das man nicht mehr lesen darf $\rightarrow$ Integritätsproblem). Modelliert keine "Covert Channels".