# IT-Sicherheit

**Safety:** Schutz vor Störungen von Innen (Hardware-/ Programmierfeher): -> Betriebssicherheit
**Security:** Schutz vor Störungen von Außen: Angriffe durch Hacker, Daten-Manipulation

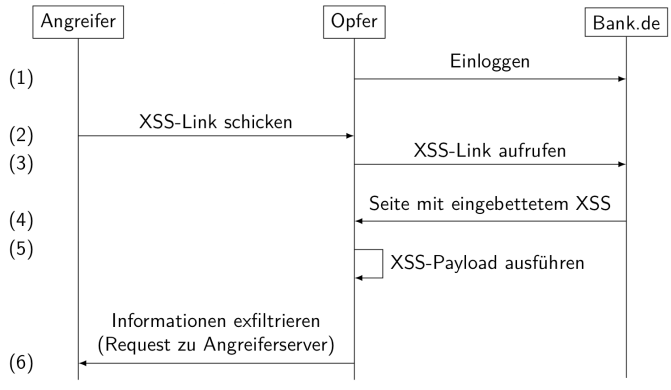
**Angriffsvektor:** konkrete Angriffsweg (eine oder mehrere Schwachstellen ausnutzt), um System-Sicherheit zu gefährden
**Schwachstelle**: ermöglicht es, die Sicherheitskontrollen des Systems umgegangen oder getäuscht werden können
**Bedrohung:** Umstand mit dem Potenzial, System durch unbefugten Zugriff, Zerstörung, Offenlegung zu beeinträchtigen

**Schutzziele** (Basis-Ziele **CIA**: **C**onfidentiality, **I**ntegrity, **A**vailability)
- **Vertraulichkeit:** Schutz vor unautorisierter Informationsgewinnung
- **Integrität:** Schutz vor unautorisierter und unbemerkter Modifikation der Daten
- **Verfügbarkeit:** Schutz vor unbefugter Beeinträchtigung der Nutzbarkeit und der Funktionalität
- **Authentizität:** Nachweis der Echtheit und Glaubwürdigkeit der Identität eines Objekts
- **Verbindlichkeit:** Schutz vor unzulässigem Abstreiten durchgeführter Handlungen
- **Privatheit:** Schutz der Privatsphäre/ Informationelle Selbstbestimmung

- **HttpOnly-Attribut:** wenn HttpOnly=true -> verhindert JavaScript, Cookie mit `document.cookie` auszulesen -> 避免 XSS
- **Secure-Attribut**: erlaubt Senden und Modifizieren des Cookies nur via HTTPS ✅, nicht via HTTP ❌
- **Same-Site Attribut** - Im Standardfall ohne Same-Site Attribut wird Cookie im Browser automatisch bei CSRF gesendet
  - 1) bei **None** - Cookie wird bei **allen Anfragen** mitgesendet; 2) bei **Lax** (standardmäßig) - Cookie wird bei **seitenübergreifenden Anfragen** (✅ GET; ❌ POST) nur mitgesendet, wenn direkt navigiert - durch Link-Klicken; 3) bei **strict** - Cookie wird mitgesendet, nur wenn die Anfrage von der **selben Webseite** ist

**Same-Origin-Policy:** JavaScript-Code hat nur lesenden Zugriff auf Inhalte auf demselben *Domain* & *Protokoll* & *Port*
**CSRF-Tokens:** füge randomisiertes Token zu jedem Request hinzu, welches vom Server validiert werden muss, 避免 CSRF
**Input Sanitization:** Ersetzen von gefährlichen Charakteren in der Eingabe

- **SQL-Injection:** Einschleusen von schädlichen SQL-Code `' OR 1=1 –` in Eingabefelder, um unautorisierte Datenbankoperationen durchzuführen (read, write, remove) - Prepared Statements (Inputs-Platzhalter verwenden)
- **CSRF-Angriff (Cross-Site Request Forgery):** manipuliert ein Opfer dazu, unbewusst schädliche Aktionen auf einer Webanwendung auszuführen, auf der der Benutzer authentifiziert ist - Anti-CSRF-Token, Same-Origin-Policy
- **XSS (Cross-Site Scripting):** ermöglicht die Ausführung von bösartigem Javascript im Browser des Opfers -> Session-Cookie/ Inhalt der angezeigten Seite werden für Angreifer zugänglich sodass extrahiert werden kann - Input sanitization (Benutzereingaben filter), Content-Security Policy, HttpOnly=true, Same-Origin-Policy
  - **Reflected:** Payload wird mittels URL an Opfer übertragen; **Persistent/ Stored:** Payload wird auf Server gespeichert




Der Angreifer findet Sicherheitslücke auf einer Webseite (bsp. Formulareingaben/ URL-Parametern)
1) Opfer hat sich bei einer vertrauenswürdigen Seite ("Bank.de") eingeloggt
2) Angreifer sendet dem Opfer einen Link mit schädlichen Code (den XSS-Payload)
3) Opfer ruft den Link (oft über Kommentarbereich/ Nutzerprofil) auf
4) Die Webseite wird mit dem eingebetteten XSS geladen
5) XSS-Payload im Browser des Opfers wird automatisch ausgeführt
6) Informationen des Benutzers (bsp. Cookies/ Session-Token) werden unbemerkt an Angreifer-Server gesendet

Bei **Stored** wird 2) durch XSS auf Bank-Server platzieren ersetzt

**Kryptografie:** die Wissenschaft, sichere Ver- und Entschlüsselungsverfahren zu entwickeln
**Kryptoanalyse:** die Wissenschaft, Ver- und Entschlüsselungsverfahren zu analysieren und zu brechen
- **Symmetrische Kryptografie:** benutzt **einen geheimen gleichen Schlüssel** für Verschlüsselung und Entschlüsselung
  - Übertragung der **eigentlichen Nachricht** mittels symmetrischer Verschlüsselung e.g. AES
- **Asymmetrische Kryptografie:** jeder Kommunikationspartner besitzt **je einen öffentlichen Schlüssel zum Verschlüsseln** und **je einen privaten Schlüssel zum Entschlüsseln** e.g. RSA
  - Übertragung des **Schlüsselmaterials** mittels asymmetrischer Verschlüsselung

-- **Verschlüsselung:** $RSA_e(x) = x^e \bmod n = y$

- **RSA-Verfahren** (asymmetrisch) (e public, d private) - kein PFS -- **Entschlüsselung:** $x = RSA_d(y) = y^d \bmod n$

  **wählt zwei Primzahlen** $p \neq q$ **und berechnet** $n := pq$ **und** $\varphi(n) = (p-1)(q-1)$

  **wählt ein** $e \in \mathbb{N}$ **mit** $\mathrm{ggT}(e, \varphi(n)) = 1$ **und berechnet** $d \in \mathbb{N}$ **mit** $ed \equiv 1(\bmod \varphi(n))$

  Warum müssen Schlüssel fürs RSA-Verfahren **deutlich länger** sein als Schlüssel für symmetrische Verfahren?
  - **Analytische Verfahren auf RSA sind deutlich stärker als für symmetrische Verfahren** (Sicherheit des RSA-Verfahrens basiert auf der **Schwere der Primfaktorzerlegung**)

**Kerckhoffs' Prinzip:** Sicherheit eines kryptografischen Verfahrens sollte nur von der Geheimhaltung des Schlüssels abhängen, nicht von der Geheimhaltung des Verfahrens abhängen (**Kein Security by Obscurity**)

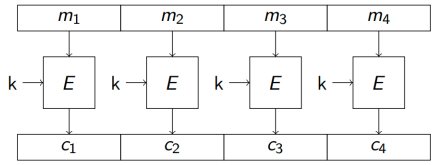
**Blockchiffre:** symmetrisches Verfahren, Unterteilung eines Klartext in Blöcke fester Länge e.g. AES
- **Padding notwendig in Klartext (如果正好: padding一整个新block)**
  - PKCS#7 (Padding-Wert = Anzahl der hinzufügenden Padding-Bytes)
- **bijektiv:** jedem Klartextblock muss genau ein Ciphertextblock zugeordnet werden
- **Diffusion:** jedes Klartext-Bit beeinflusst jedes Ciphertext-Bit
- **Konfusion:** Zusammenhang zwischen Key und Ciphertext ist nicht effizient herstellbar

**Blockchiffre** arbeiten immer auf **Blöcken fixer Größe**. Um **Nachrichten der Länge mehrere Blöcke zu verschlüsseln** werden **Betriebsmodi** benötigt. Wann immer die **Nachrichtenlänge nicht durch die Blockgröße teilbar** ist muss Padding hinzufügt werden um dies wieder sicherzustellen

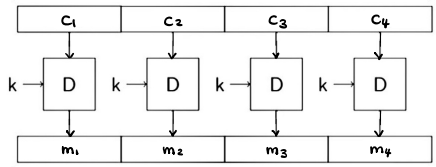
**Stromchiffre**: symmetrisches Verfahren, Klartext wird mit individueller, pseudorandomisierter Schlüsselfolge der gleichen Länge ver- und entschlüsselt (z.B. ChaCha20, AES-CTR)

## Betriebsmodi von Blockchiffren

**Verschlüsselung**

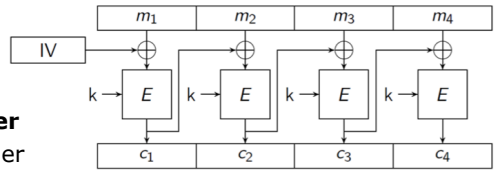



**Entschlüsselung**

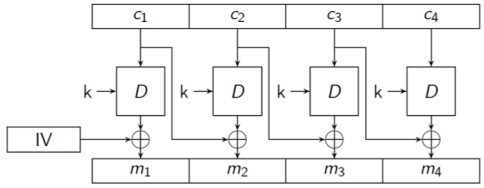



- **ECB Modus - Electronic Code Book**
  - ✅ **Parallele Ver- und Entschlüsselung** (Jeder Block wird unabhängig von einander verarbeitet) -> keine Fehlerausbreitung
  - ❌ **Gleiche** Klartext-Blöcke ergeben **gleiche** Chiffre-Blöcke -> **Muster** des Klartexts auch im Chiffretext **sichtbar/ angreifbar**
  - ❌ Nachrichten können vom Angreifer gezielt modifiziert werden bzw. neue Nachricht zu erzeugen, indem Blöcke von bekannten Klartext-/ Ciphertext-Paar beliebig zusammengefügt werden.

**Verschlüsselung**

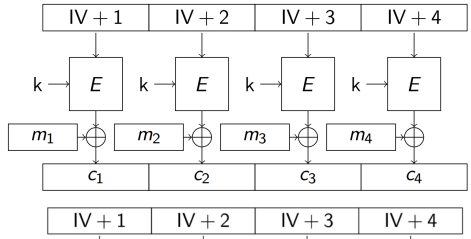



**Entschlüsselung**



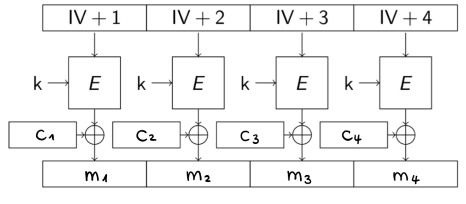

- **CBC Modus – Cipher Block Chaining**
  - ✅ Muster unsichtbar - Abbilden zwischen Klartext- u. Ciphertextblock unvorhersehbar
  - ❌ **Parallelisierung unmöglich:** aufgrund Abhängigkeit muss sequenziell verarbeitet
  - ❌ **Blockübergreifende Fehlerausbreitung** (auf **2 Blöcke** beschränkt): Ein **Bitfehler** im Block $c_{i-1}$ bei Entschlüsselung sorgt für einen korrupten Block $m_{i-1}$, und einen Bitfehler an der gleichen Stelle im Block $m_i$
  - ❌ **Padding Oracle Angriff** (wenn MAC-then-Encrypt bzw. keine Integritätsprüfung vor Entschlüsselung)**:** Vollständige Entschlüsselung des Ciphertexts einer Nachricht
    - nutzt gezielt die Ausbreitung von Bitfehlern vom verschlüsselten Block $c_n$ in den entschlüsselten Block $m_{n+1}$. Dadurch ist ein Byteweises Brute-Force des Zwischenzustands $D_k(c_{n+1})$ möglich, woraus mit dem bekannten $c_n$ der Klartext $m_{n+1}$ bestimmt werden kann: $m_{n+1} = c_n \oplus D_k(c_{n+1})$

**Verschlüsselung**



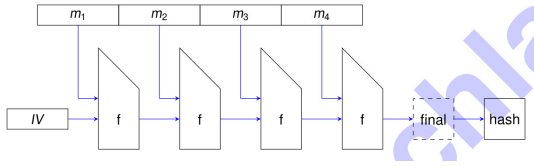

- **CTR Modus - Counter Mode** ctr(i) = ctr(i-1) + 1
  - **Verschlüsselung** des i-ten Blockes: $c_i = m_i \oplus E_k(\text{Nonce} \,\|\, \text{ctr}(i))$
  - **Entschlüsselung** des i-ten Blockes: $m_i = c_i \oplus E_k(\text{Nonce} \,\|\, \text{ctr}(i))$
  - ✅ **Parallele Ver- und Entschlüsselung** (Jeder Block wird unabhängig von einander verarbeitet) -> keine Fehlerausbreitung

**Entschlüsselung**



**Hashfunktionen** - Wegen der **Merkle-Damgärd Konstruktion** kann jede gefundene Kollision durch **Anhängen weiterer gleicher Blöcke** eine neue Kollision erzeugen (Kryptografisch sichere Hashfunktion: **SHA-3, SHA-256**)

- Zerlegen der Eingabe m in Blöcke: **$m = m_1 || m_2 || .... || m_n$**
- **Kompressionsfunktion f** verarbeitet $m_i$ mit $f(m_{i-1})$




- **Anforderungen an eine kryptografische Hashfunktion**

1. Für alle m gilt: H(m) ist **einfach zu berechnen**
2. **Einwegeigenschaft:** Für ein gegebenes h = H(m) ist das Bestimmen des Wertes $m = H^{-1}(h)$ nicht effizient möglich
3. **Schwache Kollisionsresistenz:** nicht effizient möglich ein m' für m mit m≠m' zu finden -> gilt H(m) = H(m')
4. **Starke Kollisionsresistenz:** nicht effizient möglich ein Paar unterschiedlicher (m, m') zu finden -> gilt H(m) = H(m')

**Message-Authentication-Code (MAC) -> Schutzziele: Vertraulichkeit, Integrität, Authentizität**

- **MAC-Berechnung:** MAC(m, $k_{AB}$) = H(m | $k_{AB}$) mit geheimen Schlüssel $k_{AB}$

- **"Encrypt-Then-MAC" Prinzip sollte eingehalten:** Nachrichten zuerst verschlüsselt und dann mit einem MAC gebildet -> sicherstellt, dass Nachrichten erst nach erfolgreicher Verifizierung entschlüsselt werden
- **Problem bei "MAC-Then-Encrypt":** Das Entschlüsseln von Nachrichten ohne vorherige Überprüfung ihrer Integrität und Authentizität ermöglicht Padding Oracle Angriff, bei dem Angreifer durch Analyse der Entschlüsselungsreaktionen die Klartext-Informationen in Form von vaidem/invalidem Padding extrahieren kann

**Klassischer Kollisionsangriff:** Finde Kollisionsblöcke C ≠ C', sodass bei gleichem Prefix P gilt H(P|C) = H(P|C')
**Chosen-Prefix Kollisionsangriff:** Finde Kollisionsblöcke C≠C', sodass bei verschiedene Prefixe P, P' gilt H(P|C) = H(P'|C')

❌ **Length Extension Angriff** auf Merkle-Damgard-Hashfunktionen möglich: ohne Kenntnis des Originalschlüssels einen gültigen Message Authentication Code (MAC) für erweiterte Nachrichten zu generieren
**(SHA256 SHA512 length extension möglich)** $\text{data'} = \text{data} \,||\, \text{fake}$ mit $\text{MAC} = H(k_S \,||\, \text{data'}) = H(k_S \,||\, \text{data} \,||\, \text{fake}) = H(h \,||\, \text{fake})$

**HMAC-Verfahren -> gegen Length Extension Angriffe**

- **HMAC-Konstruktion:** HMAC $(m, k)$ = H($k \oplus opad$ | (H($k \oplus ipad$ | $m$))

**AEAD (Authenticated Encryption with Associated Data):** zusätzliches Senden von (Header-)Daten zusammen mit Ciphertext zum Nachweis der Authentizität des Ciphertexts und asoziierter Daten - e.g. AES-GCM

✅ reduzieren die Fehleranfälligkeit, indem sie Verschlüsselung und Integritätsschutz in einem Schritt kombinieren und CIA Schutzziele (Vertraulichkeit, Integrität sowie Authentizität) mit nur einem Schlüssel gewährleistet (im Vergleich zu traditioneller, separater Verschlüsselung und Integritätsschutz mittels MACs) -> Padding Oracle Angriff unmöglich

**Digitale Signatur:** verwendet ein asymmetrisch Verfahren (z.B. RSA) zur **eindeutigen Zuordnung des Urhebers der Nachricht.** Signieren mit private Key $RSA_d(h) = sig$, Prüfen mit public Key $RSA_e(sig) = h$
**MACs** verwenden einen gemeinsamen symmetrischen Schlüssel für die Authentifizierung von Nachrichten zwischen zwei oder mehr Parteien, ermöglichen aber **keine eindeutige Identifizierung des Nachrichtenurhebers**
Die **Länge des Digests von 32 Bit** in einem Hash-Verfahren wie MH5 ist für sicherheitstechnische Anwendungen **nicht ausreichend groß**. Wegen des Geburtstagsparadoxons sind nur ca. $2^{32}$ Versuche notwendig sind um mit einer Wahrscheinlichkeit von mehr als 50 Prozent eine Kollision zu finden. auf heutigen PCs mittels Brute-Force leicht zu finden

**RainbowTables:** zugehörige Hashwerte für viele mögliche Passwörter vorberechnet und gespeichert, ums Knacken von Passwörtern durch Abgleich beschleunigen zu können - effektiv bei einfachen Hash-Funktionen ohne Salt

**Passwort-Speichern:** Für jedes Passwort sollte ein einzigartig random **Salt** generiert werden. Das Passwort sowie der Salt sollten dann mit einer speziellen **langsamen Passworthashfunktion** (z.B. bcrypt, scrypt, PBKDF2, Argon2) gehashed, und lediglich der Hash sowie Salt gespeichert werden -> sodass kein Brute-Force-Angriff möglich

**Passwort-Policy: Nutzer werden gezwungen ihr Passwort regelmäßig zu wechseln**

✅ das potenziell kompromittierte Passwörter verfallen regelmäßig
❌ Nutzer wählen (Durch den Druck regelmäßig das Passwort zu ändern) einfachere und leichter zu merkende Passwörter
❌ minimale Änderungen an bestehenden Passwörtern vornehmen

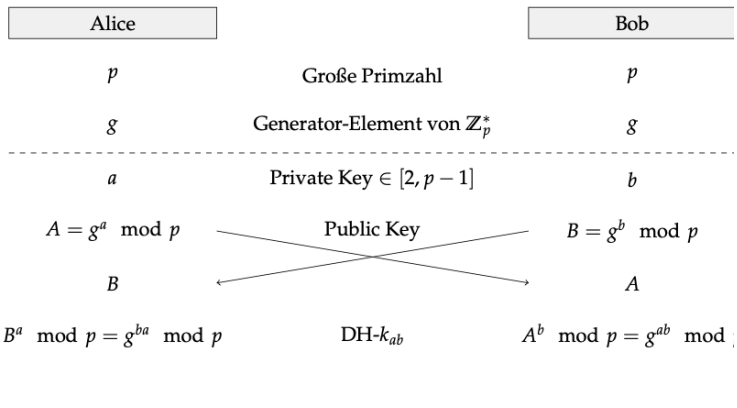
**Pseudo Random Number Generators (PRNG):** deterministisch, basiert auf Startwert Seed, vorhersagbar
**Kryptografisch sichere Zufallszahlengenerator (CSPRNG): nicht vorhersagbar** (selbst wenn Angreifer bereits Teil der generierten Zahlenfolge kennt); dürfen **keinen Hinweis auf vorhergehende Zufallszahlen** geben; erzeugen Zufallszahlen die statisch gleich viele Nullen und Einsen haben; nur **vorhersagbar** wenn **Angreifer Seed kennt**

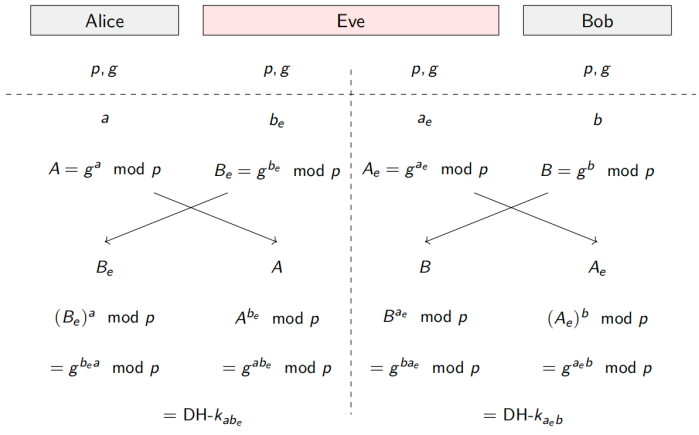
**Key Encapsulation Mechanism (KEM):** symmetrisch Schlüssel $k_{AB}$ wird asymmetrisch ver u. entschlüsselt e.g.RSA-KEM

**Perfect Forward Secrecy:** jede Sitzung verwendet einen **einzigartigen Sitzungsschlüssel** - unabhängig vom langlebigen Schlüssel X (Mit der **Aufdeckung** eines **langlebigen** Schlüssels **X** (z.B. Masterschlüssel) darf man **nicht** in der Lage sein, **vergangene mit X abgesicherte Kommunikationen** zu entschlüsseln)
- **Beispiel mit PFS: TLS 1.3. und ephemeral DH-Verfahren** (**dynamische Schlüssel** - Auf beiden Seiten müssen jeweils frische private sowie öffentliche Schlüssel pro DH-Schlüsselaustausch generiert werden)

**Diffie-Hellman-Verfahren (DH):** asymmetrischen Kryptografieverfahren - ermöglicht zwei oder mehr Parteien, über einen unsicheren Kanal einen gemeinsamen geheimen Schlüssel (typischerweise für symmetrische Verschlüsselung verwendet) zu generieren

❌ **Man-in-the-Middle(MiTM):** Angreifer modifiziert geheim den Datenaustausch zwischen zwei oder mehreren Kommunikationspartnern (weil die ausgetauschten öffentlichen Parameter jeweils nicht authentifiziert) - Angreifer ersetzt die öffentlichen Schlüssel der Kommunikationspartner durch seinen eigenen, wodurch er deren Nachrichten entschlüsseln, einsehen und manipulieren kann
**Maßnahmen gegen MiTM:** öffentliche Schlüssel sollten mittels RSA signiert und zusammen mit ihren Signaturen übertragen werden. Für die Verifikation dieser Signaturen ist der sichere Austausch der jeweiligen öffentlichen RSA-Schlüssel erforderlich, entweder durch vorherigen Out-of-Band-Austausch oder mittels X.509-Zertifikaten von vertrauenswürdigen Zertifizierungsstellen
**-> Der Wechsel von DH zu ECDH beeinflusst die MitM-Problematik nicht**, da das Hauptproblem in der fehlenden Nachrichtenauthentifizierung liegt, wodurch die Parteien nicht verifizieren können, mit wem sie kommunizieren. Die Änderung der mathematischen Basis (von ganzen Zahlen zu elliptischen Kurven) hat keinen Einfluss auf die Sicherheit gegenüber MitM-Angriffen




**Pseudonymisierung** ersetzt Identitätsdaten durch Pseudonyme, wobei die Identität (Klarnamen) mit zusätzlichen Informationen rekonstruierbar bleibt (z.B. Nickname, LRZ-Kennung)
**Anonymisierung** entfernt/ verändert Daten so, dass eine Identifizierung der Personen mit weiteren Informationen auch nicht möglich ist (z.B. Ersetzen von Namen durch Zufallszahlen)

**Replay-Angriff:** ein Angreifer sendet abgefangene Daten oder Informationen erneut, um Aktionen des ursprünglichen Absenders zu wiederholen oder Sicherheitsmechanismen zu umgehen
**Replay Cache:** speichert temporär Details von bearbeiteten Anfragen und vermeidet unberechtigte Doppelaktionen

**Challenge-Response (CR-Verfahren):** Nachweis der Identität durch ein- bzw. wechselseitiges Senden einer Challenge von einer Seite (z.B. RAND) und Senden einer *eindeutigen* Antwort von der anderen Seite, die nur diese Person lösen kann
- **schützt vor Replay Angriff**, solange $\text{RAND}_C$ und $\text{RAND}_S$ (soll möglichst groß) nicht von Client oder Server wiederverwendet werden, muss jeder Teilnehmer bei jedem Durchlauf eine frische Challenge beantworten

✅ Beim passiven Abhören kann ein Angreifer das Passwort nicht aus der Aufzeichnung rekonstruieren (eine einmalige Antwort auf eine spezifische Herausforderung nicht für zukünftige Authentifizierungen nutzbar)

**Basis-Faktoren zur Authentisierung**
- **Wissen:** Passwort, PIN, Schlüssel
- **Besitz:** Smartcard, Token, SIM Karte
- **Biometrie:** Fingerabdruck, Gesicht, Iris

**Multi-Faktor Authentifizierung:** mehrere Authentifizierungsfaktoren werden kombiniert
✅ Sicherheit deutlich erhöht: Für einen Angreifer ist es wesentlich schwieriger, alle geforderten Authentifizierungsfaktoren gleichzeitig zu kompromittieren

**Anforderungen an biometrische Merkmale - z.B. Fingerabdruck**
- **Universalität:** Jede Person besitzt das Merkmal
- **Eindeutigkeit:** Merkmal identifiziert jede Person (relativ) eindeutig
- **Beständigkeit:** Merkmal ist unveränderlich und degradiert nicht über die Zeit
- **Quantitative Erfassbarkeit:** Das Merkmal lässt sich mittels eines Sensors erfassen und verarbeiten
- **Performance:** Das Merkmal lässt sich möglichst genau und schnell erfassen
- **Akzeptanz:** Für Benutzer muss es akzeptabel sein, dass das Merkmal erfasst und für die Authentifikation verwendet wird
- **Fälschungssicherheit:** Es darf nicht einfach möglich sein das Merkmal einer Person zu fälschen

**Enrollment** (bei biometrischer Authentisierung): ein neuer Nutzer wird registriert, indem seine Referenzdaten (z.B. Fingerabdrücke) für die biometrische Authentisierung aufgenommen und gespeichert werden

**False Negative:** berechtigter Nutzer wird abgelehnt **False Rejection Rate (FRR):** Wahrscheinlichkeit für False Negative
**False Positive:** unberechtigt Nutzer wird akzeptiert **False Acceptance Rate (FAR):** Wahrscheinlichkeit für False Positive
**Equal Error Rate (ERR)**: FAR = FRR; Maß für die Güte eines biometrischen Systems

**One-Time-Passwort (OTP):** einmalig gültige Passwörter für die Authentifizierung eines Nutzers, werden periodisch automatisiert generiert - z.B. Zweifaktorauthentifizierung oder TAN-Verfahren für das Abrufen von Kontoumsätzen
- in **Software**-basiert oder **Hardware**-basiert unterschieden
- in **Challenge-Response** basiert und **Zeit**synchronisiert (Google Authenticator, RSA SecureID-Token) unterschieden

**OTPs erschweren Phishing:** Nutzbarkeit gestohlener Daten zeitlich begrenzt, doch können Phishing-Seiten OTPs noch während ihrer Gültigkeit abfangen und nutzen. Kernproblem ist fehlende Webseiten-Authentifizierung beim Nutzer

**Single-Sign-on Konzept:** ermöglicht Nutzern, sich einmalig bei einem zentralen Authentication Center zu authentifizieren und danach mit einer Authentizitätsbescheinigung nahtlos auf mehrere Dienste zuzugreifen
- **Identitätsprovider (IdP):** verantwortlich für Nutzer-Authentifizierung u. Authentizitätsbescheinigung-Ausstellung
- **Service Provider (SP):** Akzeptiert die vom IdP ausgestellten Authentizitätsbescheinigung und gewährt dem Nutzer Zugriff auf seine Dienste, basierend auf der etablierten Vertrauensbeziehung zum IdP

- **Shibboleth**
  - **Discovery Service (DS):** Mit Hilfe des DS ist es einem Service Provider möglich den zum Nutzer zugehörigen IdP zu finden
  - **SP kann die Antworten von verschiedenen IdPs** durch Besitzen eines Anchor of Trust zur PKI (alle IdPs sind Teil einer PKI) **verifizieren**, indem er die **Signaturen der von den IdPs signierten Authentication Assertions überprüft**. Dies stellt sicher, dass die Daten authentisch und unverändert sind
  - **geringer Schutz vor Phishing:** ✅ bei einem Login muss ein Nutzer lediglich die Authentizität des IdP Login-Formulars überprüfen; ❌ ein Angreifer kann nach wie vor eine Fake-Webseite sowie ein Fake IdP Login-Formular erstellen

**Zertifikatshierarchie:** Ein Browser muss nur dem Stammzertifikat USERTrust RSA Certification Authority direkt vertrauen, um automatisch allen von diesem oder transitiv vertrauten Zertifikaten signierten Websites wie www.tum.de zu vertrauen

**Schritte zur Zertifikat-Prüfung -** Dementsprechend Verifizierung für alle Zertifikate der Vertrauenskette durchgeführt
1. Stimmt das Subject (oder eins der Subject Alternative Names) mit der **Webseitendomain** überein?
2. Ist das Zertifikat im **Gültigkeitszeitraum**?
3. Sind die verwendeten Krypto-Primitive (wie z.B. das **Hashingverfahren** zum signieren) **stark genug**?
4. Wird das Zertifikat für seinen **Bestimmungszweck** genutzt?
   - **CA:TRUE** dienen als Zertifizierungsstellen - kann andere Zertifikate signieren
   - **CA:FALSE** berechtigt nicht zur Zertifikatsausstellung - kann andere Zertifikate nicht signieren
5. Ist Zertifikat **zurückgerufen** worden? (Um ein Zertifikat zurückzurufen, muss man beim Aussteller CA den Rückruf beantragen, Aussteller aktualisiert dann deren Status in öffentlichen Listen wie CRL oder OCSP)
6. **Verify($Cert_{Sig}$, $C^{pub}_{parent}$ ) ?= H(Zertifikat)**
7. Wird dem Zertifikat **vertraut**? (d.h. wird dem Zertifikat **direkt vertraut**, oder gibt es eine **lückenlose Zertifikatskette**, die in einem vertrauten Stammzertifikat endet?)

**sicherer Kanal:** gewährleistet grundlegende CIA-Schutzziele (Vertraulichkeit, Integrität, Authentizität) wodurch nicht mehr jede Anwendung eigene Sicherheitsmaßnahmen ergreifen muss, sondern kann sich auf den sicherer Kanal verlassen
- abstrahiert vom eigentlichen Anwendungsprotokoll, sichert die Schutzziele aber den **höheren Protokollschichten** zu

**Zugriffsrechte bei reguläre Datei/ Verzeichnisse**
**read (r, 4):** Inhalt lesen/ kopieren; Namen gespeicherter Dateien auflisten
**write (w,2):** Inhalt hinzufügen/ verändern/ löschen; (nur wenn x auch gesetzt ist) gespeicherte Dateien erstellen / verschieben /umbenennen/ löschen  d.h. -w- selber erlaubt keinen Zugriff auf das Verzeichnis
**execute (x,1):** Als Programm ausführen; Zugriff auf Metadaten von bereits existierende Dateien - Nutzen der Dateien im Verzeichnis entsprechend dieser Datei-Rechte lesen/ bearbeiten/ ausführen

**Subjekt:** zugreifende Instanz, also **WER** einen Zugriff durchführen möchte - z.B. Nutzer, Prozess, Service
**Objekt:** genutzte Ressource, also auf **WAS** zugreifen - z.B. Datei, Datenbankeintrag, Page
**Zugriffsrecht: WIE** bzw. Auf welche Art und Weise wird zugegriffen - z.B. Lesend, Schreibend, Kamera einschalten
**Attribut:** Unter welchen **Bedingungen** ist Zugriff valide  - z.B. Zweck, Rolle, Tageszeit

**Prinzip der minimalen Rechte ("need-to-know"):** Jedes Subjekt sollte immer nur die erforderliche Rechte besitzen, darüber hinaus keine zusätzlichen Rechte
**Prinzip der complete mediation Zugriffskontrolle ("Zero Trust"):** Jeder Zugriff auf eine geschützte Ressource muss kontrolliert werden

**Discretionary Access Control (DAC)** erlaubt einem Nutzer (mit den notwendigen Rechten) diese Rechte an andere Nutzer weiterzugegeben und selbst zu bestimmen wer auf seine Objekte zugreifen darf
**Mandatory Access Control (MAC)** verteilt systemweite Regeln, die den Zugriff auf Objekte einschränken und dessen Befolgung erzwungen wird

**Access Control Lists (ACL):** vewalten die Zugriffsrechte pro **Objekt**
- **spaltenweise** - z.B. ACL (Datei 1) = ((Bill, {r,w}),(Joe, {r,w}),(Alice, {}))
**Capabilities Lists:** verwalten Zugriffsrechte pro **Subjekt**
- **Zeilenweise** - z.B. Clist (Joe) = ((Datei 1, {r,x}), (Datei 2, {}))

**Authentifizierung 身份验证:** die Identität eines Subjekts sicherzustellen
**Autorisierung 授权认证 (是否有访问权限):** überprüft, ob ein authentifiziertes Subjekt die erforderlichen Zugriffsrechte auf eine Ressource/ Aktion besitzt

**Policy Decision Point (PDP)** führt die Berechtigungskontrolle bei dem ersten Zugriff auf ein Objekt durch und stellt bei erfolgreicher Prüfung eine Bescheinigung aus -> **Authentifizierung**
**Policy Enforcement Point (PEP)** führt die Zulässigkeitskontrolle einer Aktion basierend auf der mitgelieferten Bescheinigung aus. Die Gültigkeit der Bescheinigung wird ebenfalls geprüft -> **Autorisierung**

**Bell-LaPadula-Modell** (jede User darf nur ein Sicherheitsklasse zugeordnet werden)
• **No-Read-Up (ro, x):** Kein Lesen von Objekten mit höherer Sicherheitsklasse als Subjekt  $sc(o) \leq sc(s)$
• **No-Write-Down (a, rw):** Kein Schreiben von Objekten mit niedriger Sicherheitsklasse als Subjekt
  - **append a** mit $sc(s) \leq sc(o)$; **read-write rw** mit $sc(s) = sc(o)$
• **Strong-Tranquility-Regel**: Keine Änderung der Sicherheitsklassen von Subjekt oder Objekt zur Laufzeit

**Problem:** Alle Daten werden sukzessiv höher klassifiziert, wenn ein Benutzer mit höherer Clearance Daten mit niedrigerer Klassifikation verändert. Folglich können die Benutzer mit niedrigerer Clearance die Daten nicht mehr lesen
**-> Lösung:** vertrauenswürdige Subjekte (Systemprozesse) können eingesetzt werden, die unter Umgehung der Zugriffsregeln die Daten wieder kontrolliert niedriger klassifizieren können
**Beispiel:** Ein Pressebericht ist während seiner Entstehung noch als geheim klassifiziert, kann nach der Freigabe aber veröffentlicht werden

**Statische Aufgabentrennung:** Die gleichzeitige **Mitgliedschaft** in den Rollen A und B ist ausgeschlossen
**Dynamische Aufgabentrennung:** Die gleichzeitige **Aktivität** in der Rolle A und der Rolle B ist unzulässig

Wenn ein Subjekt auf ein Objekt rw- oder ro,a-Rechte hat, müssen beide auf der **selben Sicherheitsstufe** sein
Wenn ein Subjekt **kein** Objekt schreiben/ lesen darf, kommt das Subjekt in die höchste/ niedrigste Stufe
Wenn ein Objekt von **keinem** Subjekt geschrieben/ gelesen werden darf, kommt das Objekt in die niedrigste/ höchste Stufe

**Informationssicherheitsmanagement**

- **PDCA**-Zyklus: Plan, Do, Check, Act
- **Vorgehen-Methodik STRIDE: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Priviledge**
- **Risiko-Kategorien DREAD: Damage, Reproducibility, Exploitability, Affected, Discoverability**
- **Bedrohungsbaum** (Pro Angriffsziel 1 Baum)
  - **Wurzel = Bedrohungsanalyse-Angriffsziel; Knoten = einzelne Angriffsschritt**

**Signal Protokoll:** End zu End Verschlüsselung, jedem redet mit jedem

**Double-Ratchet-Protokoll**

- **DH-Ratchet:** Schlüsselerneuerung mit DH Verfahren (Jedes neue DH-Paar löst einen DH-Ratchet-Schritt aus)
  - Ein **DH-Ratchet-Schritt** besteht also darin, die Root-KDF-Kette zweimal zu aktualisieren und die dabei generierten Schlüssel als neue Empfangs- und Sendeketten-Schlüssel zu verwenden
- **Symmetrisches Ratchet:** Verschlüsseln mit AES-Message-Key
  - Ein **symmetrischer Double-Ratchet Schritt** läuft nach dem absenden jeder Chat-Nachricht ab. Dabei wird ein neuer Chain-Key und ein neuer Nachrichtenschlüssel mit Hilfe der KDF abgeleitet. Dadurch werden pro Nachricht unabhängige Schlüssel erzeugt und der alte Chain-Key kann möglichst schnell verworfen werden

✅ **Perfect Forward Secrecy:** Ein Angreifer der alte Sitzungschlüssel erbeutet hat, kann keine neuen Schlüssel aus diesen ableiten

✅ **Break-In-Recovery:** Nach der Kompromitierung durch einen aktiven MitM, muss nur eine einzige DH-Ratchet Nachricht den MitM unmodifiziert passieren, um dem MitM vom weiteren Verlauf der Kommunikation auszuschließen

**Alice sends message A1**

1. Alice generiert ein neues DH-Pair $(e_A, d_A)$ und kennt a init public Key $e_{initB}$ von Bob
2. Alice put $ECDH(d_A, e_{initB})$ in root $KDF$, produziert ein neues Root Key $RK$ und ein Chain Key $CK$
3. Alice macht einen symmetrischen Ratchet Schritt on $CK$, mithilfe $KDF$ wird ein neues Chain Key $CK$ u. ein Message Key $A1$ für message $mA1$ abgeleitet
4. Alice verschlüsselt $mA1$ mit $A1$ als $cA1$ und sendet $cA1$ und $e_A$ zu Bob

**Bob receives $cA1$ und $e_A$**

1. Bob put $ECDH(d_{initB}, e_A)$ in root $KDF$, produziert ein neues Root Key $RK$ und ein Chain Key $CK$
2. Bob macht einen symmetrischen Ratchet Schritt on $CK$, mithilfe $KDF$ wird ein neues Chain Key $CK$ u. ein Message Key $A1$ für message $cA1$ abgeleitet
3. Bob entschlüsselt $cA1$ mit $A1$ als $mA1$

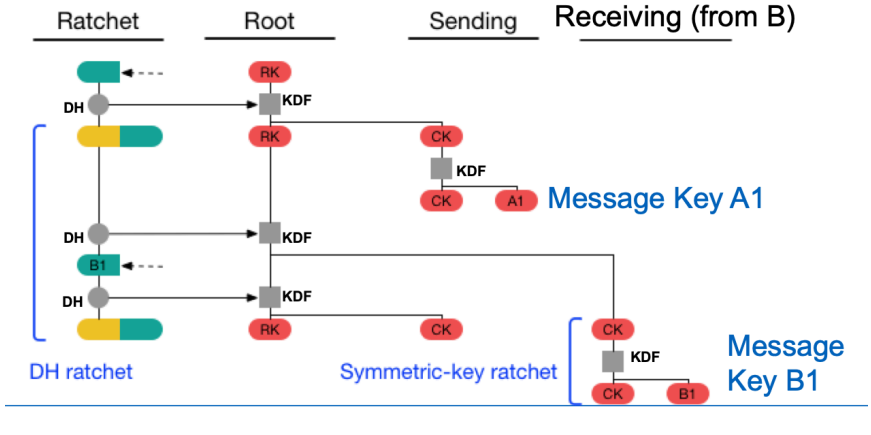
**Bob sends message $B1$**

1. Bob generiert ein neues DH-Pair $(e_B, d_B)$
2. Bob put $ECDH(d_B, e_A)$ in root $KDF$, produziert ein neues Root Key $RK$ und ein Chain Key $CK$
3. Bob macht einen symmetrischen Ratchet Schritt on $CK$, mithilfe $KDF$ wird ein neues Chain Key $CK$ u. ein Message Key $B1$ für message $mB1$ abgeleitet
4. Bob verschlüsselt $mB1$ mit $B1$ als $cB1$ und sendet $cB1$ und $e_B$ zu Alice

**Alice receives $cB1$ und $e_B$**

1. Alice put $ECDH(d_A, e_B)$ in root $KDF$, produziert ein neues Root Key $RK$ und ein Chain Key $CK$
2. Alice macht einen symmetrischen Ratchet Schritt on $CK$, mithilfe $KDF$ wird ein neues Chain Key $CK$ u. ein Message Key $B1$ für message $cB1$ abgeleitet
3. Alice entschlüsselt $cB1$ mit $B1$ als $mB1$

in nächster Runde wird Alice ein neues DH-Pair $(e'_A, d'_A)$ generieren usw.



**Transport Layer Security (TLS) - Schutzziele: Integrität, Vertraulichkeit, Authentizität**

- **Server authentifiziert sich bei Client**, indem er einen Teil des Handshakes mit seinem privaten Schlüssel des gesendeten Zertifikats signiert und dem Client diese Signatur $Sig_{ks}(H(\triangledown))$ zusammen mit seinem Zertifikat $C_s$ sendet
- **Client bestätigt die Identität des Servers** nach Prüfung der Signatur und der Finished Nachricht aus Schritt 2
  - prüft zuerst die Gültigkeit der Zertifikatskette entlang einer Hierarchie bis zu einem vertrauten Stammzertifikat
  - Dann überprüft die Signatur mit Hilfe des öffentlichen Schlüssels des Zertifikats
  - Finished-Nachrichten von Server u. Client sorgen dafür, dass kein MitM die vorherige Nachrichten unerlaubt verändert
- **Client authentifiziert** sich **durch Zertifikat $C_c$**, indem er alle bisherige Nachrichten mit seinem privaten Schlüssel des zugehörige Zertifikats in Schritt 3 signiert $Sig_{kprivc}(\triangledown)$; **ohne Zertifikat,** indem er ein Passwort als Teil der Anwendungsdaten schickt (OAuth, Shibboleth auch ok)

**0-RTT Handshake von TLS 1.3:** sendet bereits in der ersten Handshake-Nachricht verschlüsselte (mit alten zwischengespeicherten Schlüssel) Early Data

✅ schnellere Datenübertragung und geringere Latenzzeiten durch die Übertragung verschlüsselter Applikationsdaten in der ersten Nachricht, ohne dass dabei Nachrichten im Handshake weggelassen werden

❌ erhöhtes Risiko für Replay-Angriffe aufgrund der Verwendung von altem Schlüsselmaterial (z.B. POST Anfragen, die auf dem Server Zustandsänderung herbeiführen kann, wieder eingespielt werden kann)

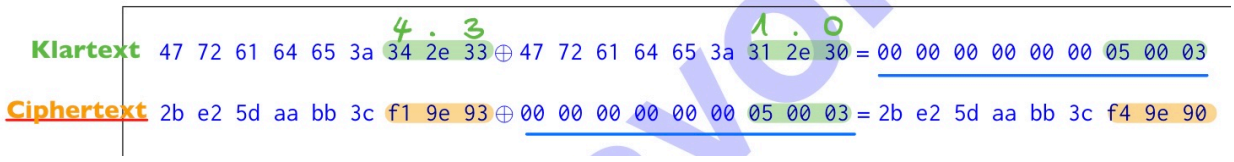
❌ kein PFS - aber akzeptabel in Webanwendungen, da die kurze Dauer von HTTP-Verbindungen das Risiko einer rückwirkenden Entschlüsselung durch kompromittierte Schlüssel minimiert

**1-RTT Handshake bei TLS 1.3:** sendet verschlüsselte Applikationsdaten erst nach Abschluss des Handshakes, nachdem beide Parteien das Schlüsselmaterial ausgetauscht und die Identität bestätigt haben

✅ höhere Sicherheit und Zuverlässigkeit durch einen vollständigen Handshake

❌ höherer Latenzzeit und potenzielle Ressourcenverbrauch aufgrund des zusätzlichen Nachrichtenaustauschs

Manipulieren mittels XOR-strategie den Klartext & Ciphertext von "Grade: 4.3" zu "Grade: 1.0", ohne Schlüssel k zu haben




Ein **aktiver** (= kann in die Kommunikation eingreifen, Nachrichten verändern, droppen, hinzufügen) Angreifer hat keine Möglichkeit, die Kommunikation abzuhören und die übertragenen Nachrichten zu entschlüsseln
Ein **passiver** (zuhören) Angreifer hat keine Möglichkeit, die Kommunikation abzuhören und die übertragenen Nachrichten zu entschlüsseln

**Klartext-Speicherung** der Daten mit dazugehörigen **MAC** -> geeignet, um **unautorisierte Datenmanipulation zu verhindern**

**fgets (buf, 0x10, stdin)** -> fgets ließt bis bis zu 15 Zeichen aus stdin und dann ein **Nullbyte \x00 am Ende hinzufügen** und speichert sie im Puffer buf

**Token-basierte Protokoll:** Dieses Token erhält von der ausstellenden Anwendung nur ein Subset der Rechte und kann jederzeit vom User wiederrufen werden.
**OAuth** ermöglicht es einem Client, sich gegenüber einem Service mit einem Token zu authentifizieren, das er von einem Authentifizierungsserver erhalten hat. Dieser Token bestätigt die Identität des Clients, ohne dass ein Passwort über die Leitung gesendet werden muss.

**Schwachstelle:** `echo "<p>$review</p>";` **Einbetten von Userinput direkt als Teil des HTML**

Bei **Copy & Paste-Angriffe**, **Diebstahl** schützt AES-XTS basierte Festplattenverschlüsselung gegen unautorisierte Informationsgewinnung

**ACME** ermöglicht das kosteneffiziente, automatisierte Ausstellen von Zertifikaten durch eine Certificate Authority. Die Identität des Servers wird dabei über die Kontrolle der Domain geprüft.

**51%-Attacke** ermöglicht es einem Angreifer, der mehr als die Hälfte der Rechenleistung in einem Blockchain-Netzwerk wie Bitcoin kontrolliert, Transaktionen zu manipulieren oder zu verhindern und Doppel-Ausgaben durchzuführen

**Blind SQL Injection** ist eine Technik, bei der Angreifer durch systematisches Testen von wahr/falsch Rückmeldungen auf Anfragen, Informationen aus einer Datenbank zeichenweise extrahieren, ohne direkte Datenlecks oder Fehlermeldungen zu erhalten

**Discoverability steht im Widerspruch zum Prinzip des Offenen Entwurfs**

**Angenommen Sie möchten auf die Webseite www.tum. de mittels HTTPS zugreifen und der Webserver schickt für seine Authentifikation ein X.509 Zertifikat. Dieses wurde von einer CA ausgestellt**
Folgende **Bedingungen** müssen erfüllt sein, damit der Client die Authentizität des Webservers erfolgreich validieren kann

- Der Client muss dem Root-Zertifikat der CA vertrauen
- Das Subjekt im Zertifikat muss mit www.tum.de übereinstimmen
- Das Zertifikat muss im Gültigkeitszeitraum sein
- Die Signatur der CA im Serverzertifikat muss valide sein
- Das Zertifikat wurde noch nicht zurückgerufen
- Die verwendeten Krypto-Primitive müssen stark genug sein

**MD5** = Hashfunktion - schwach kollisionskonsistent
**AES** = symmetrische Blockchiffre
**ChaCha20** = Verfahren Stromchiffre
**RSA** = Verfahren verschlüsseln, asymmetrisch Signieren
**ECDSA** = asymmetrisch Signieren
**3DES** = symmetrische Blockchiffre
**SHA3** = Hashfunktion

Schlüsselaustausch Gruppe: x25519

Signatur Schema: RSA-PSS-SHA256

Cipher Suite: TLS_AES_256_GCM_SHA384

**Key Exchange:** Festgelegt ist bei TLS 1.3 (anders als bei TLS 1.2), dass der Schlüsselaustausch immer via (Elliptic Curve) Diffie Hellman mit ephemeral Keys (nicht-statische Keys, werden bei jedem Key Exchange neu generiert und dann verworfen) durchgeführt wird um die PFS Eigenschaft sicherzustellen. Der ausgehandelte Parameter legt lediglich fest, auf welcher Gruppe der Schlüsselaustausch stattfindet. In diesem Fall: ECDH über die Curve25519.

**Authentifizierung des Key Exchange:** RSASSA-PSA als probabilistisches Signatur Schema (nicht Stoff der Vorlesung!), SHA256 als Hash-Funktion benötigt von dem PSS Signatur Verfahren.

**Verschlüsselung der Nutzdaten:** Die Nutzdaten werden mittels AES256 im GCM Betriebsmodus (AEAD-Chiffre) verschlüsselt.

**Pseudorandomfunktion:** SHA384 für Schlüsselableitung

(Notiz: In der Praxis wird für den Schlüsselaustausch meist nur ECDH implementiert, beispielsweise bietet OpenSSH *keine* Unterstützung für DH Schlüsselaustausch bei TLS 1.3).