

Vorlesung Sicherheit

Dennis Hofheinz

ITI, KIT

24.07.2017

1 Kurzüberblick häufige Sicherheitslücken

- Erinnerung
- Cross-Site Scripting
- SQL Injection
- Bonus: kryptographische Implementierungsprobleme
- Zusammenfassung

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

1 Kurzüberblick häufige Sicherheitslücken

■ Erinnerung

- Cross-Site Scripting

- SQL Injection

- Bonus: kryptographische Implementierungsprobleme

- Zusammenfassung

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

- Bislang idealisierte Bausteine/Algorithmen betrachtet
- Insbesondere: ideale Implementierung unterstellt
- Frage: was kann bei *Implementierung* schiefgehen?
- Ziel nachfolgend: häufige Fehlerquellen erklären

- **Ziel:** häufige Software-Fehlerquellen erklären
- Die fünf häufigsten Typen von Sicherheitsproblemen:
 - (Buffer) Overflows (schon erklärt)
 - Denial of Service (schon erklärt)
 - Code Execution (schon geklärt)
 - Cross-Site Scripting
 - SQL Injection
- **Bonus:** schlechte Zufallsgeneratoren, schlechte APIs

1 Kurzüberblick häufige Sicherheitslücken

- Erinnerung
- **Cross-Site Scripting**
- SQL Injection
- Bonus: kryptographische Implementierungsprobleme
- Zusammenfassung

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

- **Ziel:** (JavaScript-)Code auf Rechner des Opfers ausführen
- **Szenario:** Opfer surft auf Websites, denen es vertraut
- **Etwa:** Opfer surft in Diskussionsforum einer seriösen Website
- **Angriff:**
 - Angreifer postet Nachricht, die ausführbaren JS-Code enthält, als Forumsbeitrag
 - Opfer ruft Beitrag ab, liest JS-Code, Browser des Opfers führt JS-Code aus

- **Misstand:** Forenseite lässt Angreifer ausführbaren/ausgeführten JS-Code als Kommentar posten
- **Warum ist das schlimm?**
 - JS-Code läuft in vertrauenswürdigen Kontext
 - ... hat Zugriff auf Cookies/Funktionen von grundsätzlich vertrauenswürdiger Foren-Website

- **Szenario:** Facebook-Nutzer besucht Pinnwand des Angreifers
 - Angreifer hat auf seiner Pinnwand JS-Code platziert, der Facebook-Session-Cookie von Opfer an Angreifer sendet
 - Effekt: Angreifer sieht mit Facebook-Session-Cookie für Facebook wie Opfer aus und kann Konto von Opfer kontrollieren
 - Problem wurde 2010 behoben
- Andere Möglichkeit: Facebook-Wurm, der
 - 1 sich das Opfer mit dem Angreifer befreunden lässt
 - 2 und sich anschließend selbst an Freunde des Opfers sendet (Kommentarfunktion)

1 Kurzüberblick häufige Sicherheitslücken

- Erinnerung
- Cross-Site Scripting
- **SQL Injection**
- Bonus: kryptographische Implementierungsprobleme
- Zusammenfassung

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

- **Was ist SQL?** (Structured Query Language)

- Sprache, um Abfragen von Datenbanken zu formulieren
- Beispiel:

```
SELECT * FROM cd WHERE interpret = "Doktor Meta";
```

wählt alle CDs von Doktor Meta aus

- Nicht nur Queries möglich, sondern auch Anweisungen:

```
DROP TABLE cd;
```

löscht Datenbank aller CDs (aufpassen hiermit!)

- **Beispiel:** Nutzer wird nach Interpret gefragt
 - Frage an Nutzer: Nach welchem Album suchen?
 - Nutzereingabe über Webinterface, Adresszeile o.ä.
 - PHP auf Webserver redet mit SQL-Server
 - Was intern damit passieren könnte:

```
$alb = $_GET['album'];
```

```
sql_query($db,'SELECT * FROM cd WHERE album = "$alb";');
```

- Erste Zeile setzt `$alb` auf HTTP-Parameter „album“
- Zweite Zeile kommuniziert mit SQL-Server, soll nach allen Alben mit Namen `$alb` suchen

■ Beispiel:

- PHP-Skript, um nach von Benutzer gewähltem Album zu suchen:

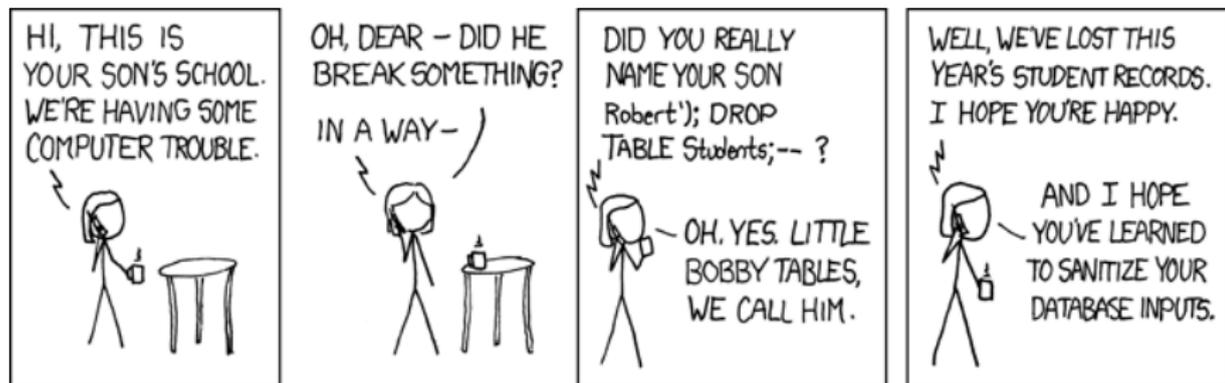
```
$alb = $_GET['album'];
```

```
sql_query($db,"SELECT * FROM cd WHERE album = '$alb';");
```

- Erwartete Eingabe (Beispiel): Evil Mastermind
- Tatsächliche Eingabe: `'; DROP TABLE cd; #`
- Konsequenz: PHP-Code setzt zwei SQL-Anfragen ab
 - 1 `SELECT * FROM cd WHERE album = ''`
 - 2 `DROP TABLE cd`
- Zweite Anfrage löscht Datenbank aller CDs

- **Problem:** Nutzereingaben werden intern weiterverarbeitet, ohne geeignet zu „escapen“
- Kann dazu führen, dass auf Server beliebiger SQL-Code ausgeführt wird
- Mögliche Effekte:
 - Löschen von Datenbanken
 - Datenzugriff auf andere Datenbank (z.B. DB aller Passwörter)
- **Gegenmaßnahme:** jede Nutzereingabe escapen/kontrollieren

SQL Injection: Demo



(xkcd.com)

1 Kurzüberblick häufige Sicherheitslücken

- Erinnerung
- Cross-Site Scripting
- SQL Injection
- **Bonus: kryptographische Implementierungsprobleme**
- Zusammenfassung

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

- Kryptographie braucht guten (d.h. unabhängig gleichverteilten) Zufall
- Besonders kritisch bei Schlüsselgenerierung (z.B. für PKE)
 - Schlüssel einmal generiert, sehr oft benutzt
 - Schlechter Zufall bei Verschlüsselung \Rightarrow Chiffre unsicher
 - Schlechter Zufall bei Schlüsselgenerierung \Rightarrow alle Chiffre unsicher
- Viele mögliche Gründe für schlechten Zufall: fehlende Entropie, physikalischer Angriff, **Implementierungsfehler**

Was kann passieren?

- Erinnerung ElGamal-/DSA-Signaturen:

$$pk = (\mathbb{G}, g, g^x) \quad sk = (\mathbb{G}, g, x)$$
$$\text{Sig}(sk, M) = (a, b) \quad \text{mit} \quad a := g^e \quad \text{für zufälliges } e$$

und b als Lösung von $a \cdot x + e \cdot b = M \bmod |\mathbb{G}|$

- Zweimal derselbe Zufall (d.h. dasselbe e) für Signaturen (a, b) und (a, b') verschiedener Nachrichten M, M' verwendet:

$$a \cdot x + e \cdot b = M \bmod |\mathbb{G}|$$
$$a \cdot x + e \cdot b' = M' \bmod |\mathbb{G}|$$

- Lineare Algebra liefert zunächst e und dann x (also $sk!$)
- **Real:** Sony hat für PS3 so Code signiert (\rightarrow Linux auf PS3)

Was kann passieren?

- Schlechter Zufall bei Schlüsselgenerierung verheerend
- Im Extremfall nur, sagen wir, 2^{16} mögliche Schlüsselpaare (Brute-Force-Suche über alle Schlüssel möglich)
- Leider realistisch: Debian-OpenSSL-Problem
 - September 2006: Debian-Maintainer patcht OpenSSL-Paket, damit Valgrind-Analyse-Tool keine Zugriffe auf undefinierten Speicher meldet
 - Problem: OpenSSL greift gezielt auf undefinierten Speicher zu, um Zufall für Schlüsselgenerierung zu erzeugen
 - Effekt: Debian-OpenSSL-Schlüsselgenerierung nutzt (für anfällige Versionen) nur Prozess-ID (16 Bit) als Zufall
 - Im Mai 2008 bemerkt und korrigiert

- API (Application Programming Interface):
Programmierschnittstelle
- Viele APIs kompliziert, unübersichtlich, ändern sich
- Ähnlich (strenggenommen keine API):
Kommandozeilenoptionen von komplexen Programmen
- Typisches Vorgehen bei Versuch, unbekannte API zu nutzen
 - 1 Rumprobieren
 - 2 Versuchen, Dokumentation zu lesen
 - 3 Vorwärtsblättern zu den Beispielen
 - 4 Versuch, Beispiel auf eigene Bedürfnisse anzupassen
 - 5 Sofort aufhören, sobald erwünschter Effekt eintritt
- **Sehr problematisch** bei kryptographischen APIs

- Erinnerung RSA:

$$pk = (N, e)$$

$$sk = (N, d)$$

$$C = M^e \bmod N$$

$$\text{(oder, besser:)} \quad C = \text{pad}(M)^e \bmod N$$

- **Beispiel:** <https://github.com/saltstack/salt/commit/5dd304276ba5745ec21fc1e6686a0b28da29e6fc>
 - Verwendet eine RSA-Implementation
 - API überlässt Wahl von e bei Schlüsselerzeugung dem Benutzer und erzeugt e nicht selbst
 - Ursprüngliche wurde im Projekt $e = 1$ gewählt...

1 Kurzüberblick häufige Sicherheitslücken

- Erinnerung
- Cross-Site Scripting
- SQL Injection
- Bonus: kryptographische Implementierungsprobleme
- **Zusammenfassung**

2 Allgemeine Bemerkungen

3 Überblick über die Vorlesung

- Häufig sicherheitsrelevante Probleme bei *Implementierung*
- Wiederkehrendes Problem: Eingabe (z.B. über Webinterface) wird nicht geeignet geparkt/überprüft/escaped
- Ermöglicht Angreifer oft sogar, Code einzuschleusen
- Kryptographiespezifisch: schlechter Zufall, schlechte APIs
- **Vermeidung der Probleme:** Eingaben immer konservativ parsen, Standardmethoden/-APIs/-pakete benutzen

- 1 Kurzüberblick häufige Sicherheitslücken
 - Erinnerung
 - Cross-Site Scripting
 - SQL Injection
 - Bonus: kryptographische Implementierungsprobleme
 - Zusammenfassung
- 2 Allgemeine Bemerkungen
- 3 Überblick über die Vorlesung

- Verschlüsselung benutzen
- Standardverfahren benutzen (RSA-/ECC-Varianten, AES, Keccak, TLS)
 - Bei TLS in aktueller Version, mit aktuellen Patches
- **Nicht:** schnell selbst in obskurem Artikel vorgeschlagenes chaosbasiertes Verschlüsselungsverfahren implementieren

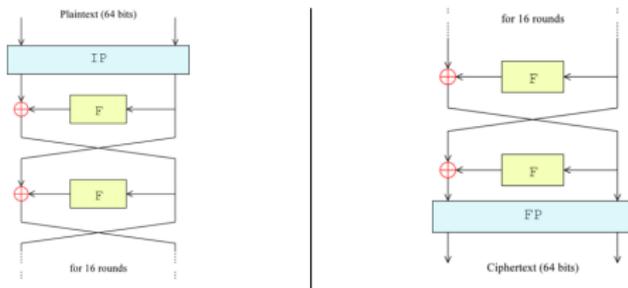
- Vorlesungen im kommenden Semester (WS17/18):
 - Komplexitätstheorie
 - Beweisbare Sicherheit in der Kryptographie
 - Digitale Signaturen
 - Asymmetrische Verschlüsselungsverfahren
 - Signale und Codes

- 1 Kurzüberblick häufige Sicherheitslücken
 - Erinnerung
 - Cross-Site Scripting
 - SQL Injection
 - Bonus: kryptographische Implementierungsprobleme
 - Zusammenfassung
- 2 Allgemeine Bemerkungen
- 3 Überblick über die Vorlesung

Symmetrische Verschlüsselung

- Verschlüsselung: $C = \text{Enc}(K, M)$
- Entschlüsselung: $\text{Dec}(K, C) = M$
- One-Time-Pad $C = M \oplus K$ (veränderbar, unhandlich)
- OTP unhandlich: **Stromchiffre** „simuliert“ OTP
 - Setze $C = M \oplus G(K)$ für Pseudozufallsgenerator G
 - $G(K) := (b^{(1)}, \dots, b^{(n)})$ mit $(b^{(i+1)}, K^{(i+1)}) := \text{SC}(K^{(i)})$
- Beispiel: LFSRs (unsicher, deshalb LFSRs kombinieren)
- Stromchiffre **schnell**, braucht **Synchronisation**, **verwundbar**

- **Blockchiffre** besteht aus $E : \{0, 1\}^k \times \{0, 1\}^k \rightarrow \{0, 1\}^k$ (invertierbare Funktion), die in Betriebsmodus genutzt wird
- Beispiele Betriebsmodus:
 - ECB: $C = (C_1, \dots)$ mit $C_i = E(K, M_i)$, simpel, **schwach**
 - CBC besser: $C_0 := IV$, $C_i := E(K, M_i \oplus C_{i-1})$
- Beispiel E: DES (**Feistelstruktur**, mittlerweile zu schwach)



- Angriffe: Meet-in-the-Middle, differentielle/lineare Analyse

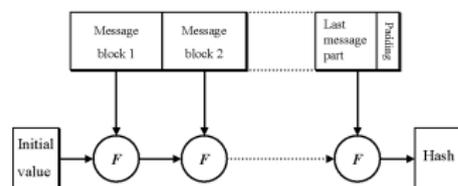
Sicherheit von Verschlüsselungsverfahren

- **Semantische Sicherheit:** Chiffre hilft nicht bei Berechnungen über Klartext, äquivalent zu IND-CPA
- **IND-CPA:** Chiffre von verschiedenen Klartexten nicht unterscheidbar
- Blockchiffre im CBC-Modus (mit zufälligem IV) IND-CPA
- Feistel-Rundenfunktion pseudozufällig \Rightarrow Feistel-Ausgabe pseudozufällig (und invertierbar!)

- Symmetrisches Verfahren gilt als gebrochen, wenn besserer Angriff als Brute Force bekannt
 - Manchmal strittig, ob Angriffsvoraussetzungen zu absurd (Differentielle Analyse von DES, Related-Key-Angriff auf AES-256)
- Populäre ungebrochene Schemata: AES, 3DES

Hashfunktionen

- **Hashfunktion** $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$: Daten-Fingerabdruck
- **Kollisionsresistenz**: kein effizienter Angreifer kann mit nicht-vernachlässigbarer Wahrscheinlichkeit Kollision finden
- **Einwegeigenschaft** (bzgl. $\{0, 1\}^{2k}$) impliziert von Koll.-res.
- Merkle-Damgård-Konstruktion (Kompressionsfunktion $\Rightarrow H$)



- F kollisionsresistent $\Rightarrow H$ kollisionsresistent
- Wichtige MD-Instanzen: MD5, SHA-1 (**beide gebrochen**)

- Die zwei populärsten Hashfunktionen gebrochen (MD5, SHA-1)
- Hashfunktion gebrochen \Leftrightarrow es gibt nichttrivialen Angriff
 - Trivialer und generischer Angriff: Birthday Attack (viele Paare hashen, nach Kollision suchen)
- Aktuell ungebrochen: Keccak (SHA-3)
- Bei Keccak: Ausgabelänge variabel (sollte 200 Bits oder mehr sein, wenn Kollisionsresistenz gewünscht)

Asymmetrische Verschlüsselung

- Schlüsselgenerierung: $(pk, sk) \leftarrow \text{Gen}(1^k)$
- Verschlüsseln: $C \leftarrow \text{Enc}(pk, M)$
- Entschlüsseln: $\text{Dec}(sk, C) = M$
- **RSA:** $C = M^e \bmod N$ (so unsicher, homomorph)
besser $C = \text{pad}(M)^e \bmod N$
 - RSA-OAEP unter RSA-Annahme aktiv sicher (nur im Random Oracle Model)
- **ElGamal:** $pk = (\mathbb{G}, g, g^x)$, $C = (g^y, g^{xy} \cdot M)$ (homomorph, besonders effizient auf elliptischen Kurven)
 - unter DDH-Annahme IND-CPA-sicher (im Standardmodell)

- Bester Angriff auf RSA(-Varianten): Faktorisieren
- Bester Faktorisierungsalgorithmus: Zahlkörpersieb
 - Komplexer, zweistufiger Algorithmus
 - Nur erster Teil parallelisierbar
 - Implementierung auf Standardrechnern: 700-800-Bit-Zahlen
 - Vorgeschlagene Spezialhardware für 1024 Bit: 1 Jahr, viele M\$
 - Deshalb vorgeschlagene Parameter: 2048-Bit- N

- Bester Angriff auf ElGamal: diskrete Logarithmen berechnen
 - Nur generische Algorithmen für (geeignete) elliptische Kurven
 - In $\mathbb{G} \subseteq \mathbb{Z}_p^*$ Index-Calculus-Algorithmus
 - IC zweistufig: einmalige Vorberechnung, dann DLog
 - Parameter: elliptische Kurven 200-300 Bit, \mathbb{Z}_p^* 2048 Bit
- Diskussion gilt auch für Varianten und Signaturschemata

Symmetrische Nachrichtenauthentifikation (MACs)

- Signieren: $\sigma \leftarrow \text{Sig}(K, M)$
- Verifizieren: $\text{Ver}(K, \sigma)$ (gibt 0 oder 1 aus)
- **EUFCMA-Sicherheit:** effizienter Angreifer, der Signaturen erfragen kann, kann kein σ für neue Nachricht fälschen
- **PRF als MAC,** Kandidat: $\text{PRF}(K, M) = H(K \| M)$
 - **Achtung:** Kandidat keine PRF, wenn M -Länge variabel, und wenn H nach Merkle-Damgård-Prinzip aufgebaut

Symmetrische Nachrichtenthauthentifikation (MACs)

- **Hash-Then-Sign:** signiere $H(M)$ (erhält EUF-CMA, Vorteil: Sig muss nur kurze Nachricht signieren)
- **Praxis:** HMAC ($\text{Sig}(K, M) = H(K \oplus \text{opad} \| H(K \oplus \text{ipad} \| M))$)

Asymmetrische Nachrichtenauthentifikation (Signaturen)

- Schlüsselgenerierung: $(pk, sk) \leftarrow \text{Gen}(1^k)$
- Signieren: $\sigma \leftarrow \text{Sig}(sk, M)$
- Verifizieren: $\text{Ver}(pk, \sigma)$ (gibt 0 oder 1 aus)
- EUF-CMA und Hash-Then-Sign (*fast* wie bei MACs)
 - EUF-CMA-Unterschied: Angreifer erhält pk (zu Beginn)

Asymmetrische Nachrichtenthenthifikation (Signaturen)

- RSA als Signaturschema ($\sigma = M^d \bmod N$):
 - Ungepaddet nicht EUF-CMA-sicher (homomorph!)
 - Besser: RSA-PSS
- ElGamal-Signaturen: nicht EUF-CMA-sicher
- DSA: quasi ElGamal-Signaturen gehasht, Sicherheit unklar

- Mit Schlüsselzentrum: Kerberos (mit symm. Verschlüsselung)
 - Kerberos praktikabel, stellt Authentifikation sicher
- Mit PK-Infrastruktur: **PK-Transport, Diffie-Hellman**
- Formale Sicherheitsmodelle komplex, viele Angriffsszenarien
- In Praxis benutzt: **TLS** (Transport Layer Security)
 - Schlüsselaustausch mit anschließender Verschlüsselung
 - Meistens nur ältere Versionen benutzt, viele Patches
 - Viele Angriffe: ChangeCipherSpec Drop, Angriff auf RSA-Padding, CRIME, ...
 - Historisch gewachsen, theoretisch unbefriedigend, Standard

- Ziel: PK-Infrastruktur gegeben, Partei identifiziert sich mit sk
- **PK-ID-Sicherheit:** Angreifer kann niemanden impersonieren, selbst wenn er vorher mit vielen Provern geredet hat
- Für PK-ID-Sicherheit **Signaturprotokoll** genug

$$\begin{array}{l} \mathbf{1} \quad P \quad \xleftarrow{R} \quad V \\ \mathbf{2} \quad P \quad \xrightarrow{\sigma := \text{Sig}(sk_A, R)} \quad V \end{array}$$

- Funktioniert auch mit (aktiv sicherer!) Verschlüsselung

Zero-Knowledge-Protokolle

- **Idee:** Prover beweist, dass er sk kennt, aber Verifier lernt nichts (außer, dass Prover sk kennt)
- **Zero-Knowledge:** Transkripte (mit beliebigem Verifier) können simuliert werden
- **Proof of Knowledge:** sk kann aus jedem erfolgreichen Prover extrahiert werden
- Beispielprotokoll mit $pk = \text{Graph}$, $sk = \text{Dreifärbung}$
- $\text{ZK} + \text{PoK} \Rightarrow \text{PK-ID-Sicherheit}$ nur, wenn $pk \rightarrow sk$ schwer
- Konzept nützlich, um beliebige NP-Aussagen zu zeigen

- Standard: Nutzerauthentifikation mit Passwort pw
- Möglich: Server kennt $H(pw)$
 - Wörterbuchangriff möglich, wenn Server korrumpiert
 - Effizienter: Wörterbuch komprimieren (Rainbow Tables),
Grundidee: Wörterbuch gezielt dynamisch bei Angriff ergänzen
 - Modern: Brute-Force-Suche mit vielen GPUs

- **Besser:** gesalzene Hashes $H(\text{pw}, \text{salt})$ (verhindert zumindest Komprimierung von Wörterbuch, nicht aber GPU-Angriff)
- Weitere Maßnahmen: Key Strengthening, bessere Passwortwahl
- Weitere Authentifikationstypen möglich (z.B. positionsbasierte Authentifikation)

- Ziel: Subjekten Rechte auf Objekte gewähren/verweigern
- Rechteverwaltung von Dateisystem statisch
- Bell-LaPadula: dynamische Rechteverwaltung (ds-, NoReadUp-, NoWriteDown-Eigenschaft), **nur Secrecy**
- Chinese Wall (ss-, Star-Property): vollkommen dynamisch, erkennt/verwaltet potentielle Interessenskonflikte

Achtung: Klausuraufgaben vor Sommersemester 2013 zu Bell-LaPadula nicht prüfungsrelevant, da Modell damals leicht anders definiert.

- **CIA-Paradigma:** Eigenschaften überprüfen (Confidentiality, Integrity, Availability), sehr anwendungsspezifisch
- **Simulierbarkeit:** Sicherheit durch Vergleich mit Idealisierung der Protokollaufgabe definieren
 - Relation „ \geq “ auf Protokollen
 - Modulares Design durch Kompositionstheorem

Häufige Sicherheitslücken

- Software-Sicherheitslücken in CVE-Datenbank gesammelt
 - <https://cve.mitre.org/>
- Häufige Probleme: Buffer Overflows, Denial of Service, Code Execution, Cross-Site-Scripting, SQL-Injection
- **Wichtig:** mit Nutzereingaben vorsichtig umgehen
- Weitere Probleme: schlechter kryptographischer Zufall, schlechte APIs