

# Vorlesung Sicherheit

Dennis Hofheinz

ITI, KIT

17.07.2017

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

## 1 Analyse größerer Systeme

### ■ Motivation

- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- Bislang Bausteine betrachtet
  - Verschlüsselung, Hashfunktionen, Authentifikation, ...
- Gesamtsystem erfordert üblicherweise mehrere Bausteine
  - Online-Banking benötigt Authentifikation und Verschlüsselung
- Welche Bausteine sollten wie eingesetzt werden?
- **Oder:** wie baut man größeres System modular auf?

- Zwei Zugänge: (Information) Security und kryptographisch
- **Security-Zugang:**
  - Prüfe gezielt Eigenschaften des Gesamtsystems
  - Üblich: CIA-Paradigma (Confidentiality, Integrity, Availability)
- **Kryptographischer Zugang:**
  - Vergleiche reales System mit vereinfachtem, idealem System
  - Reales System sicher wenn „so sicher wie“ Idealisierung

## 1 Analyse größerer Systeme

- Motivation
- **Der Security-Zugang**
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- Grundidee: überprüfe gezielt Eigenschaften des Gesamtsystems
- Üblicherweise drei entscheidende Eigenschaften (CIA):
  - Confidentiality: Geheimhaltung der Daten im System  
(Beispiel: Kontostand bleibt bei Online-Banking geheim)
  - Integrity: Integrität/Konsistenz von Daten im System  
(Beispiel: Angreifer kann Überweisungsdaten nicht ändern)
  - Availability: Verfügbarkeit des Systems  
(Beispiel: Angreifer kann Online-Banking-System nicht lahmlegen/Überweisungen kommen an)
- Manchmal auch weitere Eigenschaften betrachtet  
(Beispiel: Nicht-Abstreitbarkeit)

# Grenzen des CIA-Paradigmas

- CIA-Paradigma legt keine *konkreten* Schutzziele fest
- Beispiel (Confidentiality Online-Banking):
  - Sollte Angreifer wissen dürfen, *dass* Online-Banking stattfindet?
  - Sollte Angreifer wissen dürfen, *ob* man Überweisungen tätigt?
  - Sollte Angreifer wissen dürfen, *wie viele* Überweisungen man tätigt?
- Konkrete Schutzziele abhängig von Anwendung/Wünschen



# Grenzen des CIA-Paradigmas

- CIA-Paradigma sehr anwendungsspezifisch, stellt sicher, dass nichts Grundsätzliches vergessen wurde
- Beweis/formalere Analyse nur bei spezifischeren Schutzzielen
- **Aber:** Sicherheit größerer Systeme überhaupt erst mit CIA-Paradigma beherrschbar

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- **Der kryptographische Zugang**
- Zusammenfassung

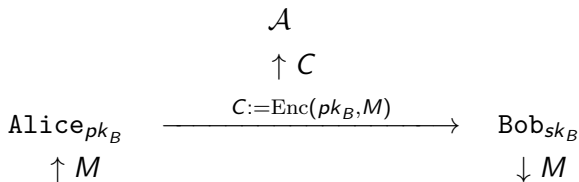
## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- **Beobachtung:** Manchmal Betrachtung von einzelnen Eigenschaften (CIA) problematisch
  - Manche Eigenschaften (Nicht-Abstreitbarkeit) nicht berücksichtigt
  - Wann ist „Liste der wünschenswerten Eigenschaften“ vollständig?
- **Ziel:** genereller Sicherheitsbegriff, der nicht auf einzelne Eigenschaften aufbaut

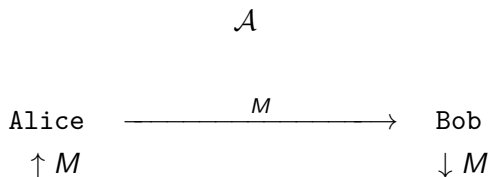
- **Idee:** Ziel wird nicht durch Eigenschaften, sondern durch idealisiertes System spezifiziert
- **Beispiel:** „sicherer Kanal“ kann Ziel von Verschlüsselung sein
- Verschlüsselung sicher gdw. sie sicheren Kanal implementiert
- **Zentrales Element:** Relation „ $\geq$ “ auf Protokollen
  - Intuition:  $\pi_1 \geq \pi_2$  heißt „ $\pi_1$  mindestens so sicher wie  $\pi_2$ “ ...
  - ... oder „ $\pi_1$  implementiert/realisiert  $\pi_2$ “
  - Dabei üblicherweise  $\pi_1$  reales Protokoll,  $\pi_2$  Idealisierung

- **Beispiel:** Reales Protokoll  $\pi_1$  nutzt PKE-Verfahren



**Wichtig:** Kanal unsicher (d.h. Angreifer  $\mathcal{A}$  erhält  $C$ ),  
Angreifer  $\mathcal{A}$  nur passiv (d.h. lauscht nur)

- Ideales Protokoll  $\pi_2$  modelliert sichere Kommunikation



Angreifer  $\mathcal{A}$  erhält keine Information über Kommunikation

- **Frage:** Ist  $\pi_1$  so sicher wie  $\pi_2$ ? (Gilt  $\pi_1 \geq \pi_2$ ?)
- Was bedeutet „ $\geq$ “ eigentlich? Was sollte  $\geq$  bedeuten?
- **Intuition:**  $\pi_1 \geq \pi_2$  wenn...
  - ...  $\pi_1$  so sicher wie  $\pi_2$
  - ... jede Schwäche von  $\pi_1$  auch schon in  $\pi_2$
  - ... für jeden Angreifer  $\mathcal{A}_1$  auf  $\pi_1$  ein Angreifer („Simulator“)  $\mathcal{A}_2$  auf  $\pi_2$  existiert, so dass die Effekte in beiden Fällen gleich
  - ... aber was heißt „... so dass die Effekte gleich“?

## Definition (Simulierbarkeit, informell)

**Protokoll**  $\pi_1$  ist *so sicher wie* **Protokoll**  $\pi_2$  (kurz:  $\pi_1 \geq \pi_2$ ), falls für jeden **effizienten** Angreifer  $\mathcal{A}_1$  auf  $\pi_1$  ein **effizienter** Angreifer  $\mathcal{A}_2$  auf  $\pi_2$  existiert, so dass nicht **effizient** zwischen  $(\pi_1, \mathcal{A}_1)$  und  $(\pi_2, \mathcal{A}_2)$  **unterschieden** werden kann.

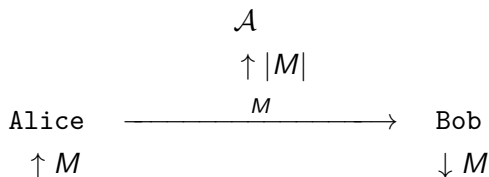
- (**Rot** bedeutet: so noch nicht genau definiert)
- **Ununterscheidbare** Protokolle: können „von außen“ (aus Sicht eines Protokollbenutzers) nicht unterschieden werden



- **Frage:** gilt für obige Protokolle  $\pi_1 \geq \pi_2$ ?
- **Antwort:** nein
  - Grund:  $\pi_1$  lässt Angreifer wissen, dass Kommunikation stattfindet
  - Aber:  $\pi_2$  lässt Angreifer dies nicht wissen
  - Konsequenz: Angreifer  $\mathcal{A}_1$  auf  $\pi_1$ , der nur zum Benutzer signalisiert, wenn Kommunikation stattfindet, kann nicht durch ein  $\mathcal{A}_2$  simuliert werden

# Beispiel Simulierbarkeit

- Geändertes Protokoll  $\pi'_2$ :



Angreifer  $\mathcal{A}$  erhält Nachrichtenlänge  $|M|$

# Beispiel Simulierbarkeit

- Für obige  $\pi_1, \pi'_2$  gilt  $\pi_1 \geq \pi'_2$ , falls das in  $\pi_1$  verwendete PKE-Verfahren IND-CPA-sicher ist
- Achtung: selbst ideales Protokoll gibt  $|M|$  an Angreifer
  - Grund: sonst würde nicht  $\pi_1 \geq \pi'_2$  gelten können
  - In  $\pi_1$  erhält  $\mathcal{A}$  das Chiffre  $C$
  - Tatsächlich gibt  $C$  Aufschluss über  $|M|$  (Nachrichtenlänge kann nicht komplett verborgen werden)
  - Steht diese Information nicht auch in  $\pi'_2$  zur Verfügung, erlaubt  $\pi_1$  mehr „Angriffe“ als  $\pi'_2$

- Vorteile von „ $\geq$ “:
  - Erlaubt intuitive Formulierung von Sicherheit
  - **Und:** erlaubt auch modulare Analyse von Protokollen
- Grundsätzliche Idee bei modularer Analyse:
  - Entwirf komplexes Protokoll mit idealisierten Bausteinen, ersetze idealisierte Bausteine später durch sichere Implementierungen
- Zentrales Werkzeug: Kompositionstheorem

## Theorem (Kompositionstheorem (informell))

Sei  $\pi^\tau$  ein *Protokoll*, das ein *Unterprotokoll*  $\tau$  benutzt. Sei weiter  $\rho$  ein *Protokoll* mit  $\rho \geq \tau$ , und sei  $\pi^\rho$  das *Protokoll*, welches  $\rho$  statt  $\tau$  als *Unterprotokoll* benutzt. Dann gilt  $\pi^\rho \geq \pi^\tau$ .

- Modulare Analyse (genauer):
  - Formuliere größeres Protokoll  $\pi^\tau$  mit idealisierten Bausteinen  $\tau$   
(Bsp.:  $\pi^\tau$  nutzt sicheren Kanal  $\tau$  für Kommunikation)
  - Beweise Sicherheit von  $\pi^\tau$  im Sinne von  $\pi^\tau \geq \pi'$   
( $\pi'$  Idealisierung von Protokollziel, z.B. sicheres Online-Banking)
  - Ersetze idealisierte Bausteine  $\tau$  in  $\pi^\tau$  durch reale Implementierungen  $\rho$  mit  $\rho \geq \tau$   
(Bsp.:  $\rho$  sicherer Kanal,  $\rho$  Verschlüsselungsprotokoll wie oben)
  - Kompositionstheorem: dann gilt schon  $\pi^\rho \geq \pi^\tau \geq \pi'$
- Erlaubt, Teilsysteme von größerem System getrennt zu analysieren

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- **Zusammenfassung**

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- Größere Systeme können auf mehrere Arten analysiert werden
- Security-Herangehensweise: Eigenschaften nachweisen (CIA)
- Kryptographische Herangehensweise: Simulierbarkeit
  - Drückt Sicherheitsziel(e) durch Vergleich mit Idealisierung aus
  - Intuitive Definition, modulare Analyse möglich
  - **Aber:** technisch schwerer zu behandeln

- Formalisierung von (CIA-)Eigenschaften in formalem Kalkül
  - Nachweis durch formale (maschinengestützte) Verifikation
  - Extrem erfolgreich für Protokolle, deren Bausteine gut formalisierbar/abstrahierbar sind
  - Oft „Computational-Soundness-Resultate“ nötig
- Simulierbarkeit („ $\geq$ “) von Standardbausteinen
  - Was ist eine gute Idealisierung/Abstraktion von ...
  - ... Verschlüsselung, Signaturen, Commitments, ...?
  - Was *bringen* einem diese Bausteine in größerem Protokoll, und unter welchen technischen Annahmen?



## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- 1 Analyse größerer Systeme
  - Motivation
  - Der Security-Zugang
  - Der kryptographische Zugang
  - Zusammenfassung
  
- 2 Kurzüberblick häufige Sicherheitslücken
  - **Motivation**
  - Buffer Overflows
  - Denial of Service
  - Code Execution
  - Cross-Site Scripting

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

- Datenbank für Sicherheitsprobleme in Software: CVE (Common Vulnerabilities and Exposures)

<http://cve.mitre.org/cve/>

- Ziel der Datenbank: *eine* Anlaufstelle für Sicherheitsprobleme
  - Die fünf häufigsten Typen von Sicherheitsproblemen:
    - (Buffer) Overflows
    - Denial of Service
    - Code Execution
    - Cross-Site Scripting
    - SQL Injection
- (werden noch näher erläutert)
- **Bonus:** schlechte Zufallsgeneratoren, schlechte APIs

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- **Buffer Overflows**
- Denial of Service
- Code Execution
- Cross-Site Scripting

# Buffer Overflows

- **Problematik:** viele Programmiersprachen (z.B. C) reservieren statisch bestimmten Speicherplatz für Variablen
- Beispiel (in C):

```
char name[48], adresse[64];
```

reserviert 48 Zeichen für Variable name und 64 für adresse

- Üblicherweise wird Größe von Variable nicht überprüft
- Beispiel (in C):

```
scanf("%s", name);
```

parst STDIN-Eingabe in name

- **Eingabe länger als 48 Zeichen ⇒ adresse wird überschrieben**

# Buffer Overflows

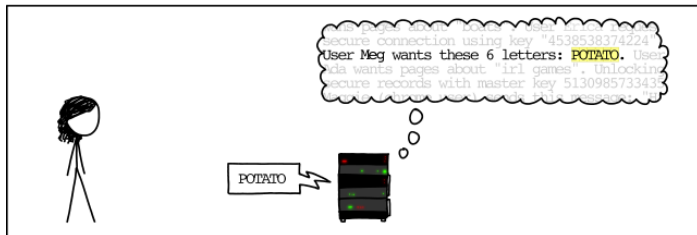
- Bei ungeschickter Programmierung kann Angreifer (z.B. durch Eingaben) andere Variablen/Speicherbereiche überschreiben
- Klingt schon beunruhigend...
- ... aber es kommt noch schlimmer
- **Problem:** bei Unterprogrammaufruf lokale Variablen auf Stack
- ... und direkt hinter Variablen Rücksprungadresse
- **Konsequenz:** Angreifer kann auch Rücksprungadresse durch eigene Adresse auf Stack überschreiben/ersetzen...
- ... und so eigenen Code ausführen (lassen)

- **Gegenmaßnahmen?**
- Benutzung sichererer Programmiersprache (Java, Python, ...)
- Wenn nicht möglich: Benutzung von Routinen, die explizite Längengrenzen von verlangen/überprüfen
- DEP (Data Execution Prevention): verhindern, dass Code auf Stack ausgeführt wird (Trennung von Code- und Datenbereichen)
  - Beispiel: OpenBSD erzwingt `write-xor-execute`-Regel
- ASLR (Address Space Layout Randomization): Adressraum randomisieren (Adresse von Code im Stack unvorhersagbar)



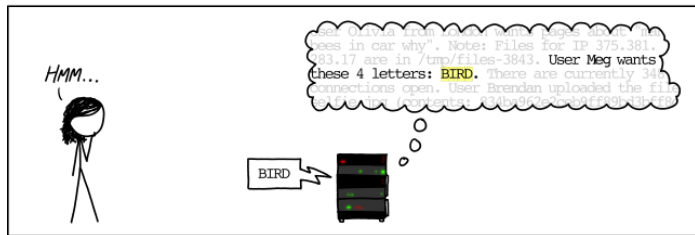
# Buffer Overflows (Variante, Beispiel Heartbleed)

## HOW THE HEARTBLEED BUG WORKS:



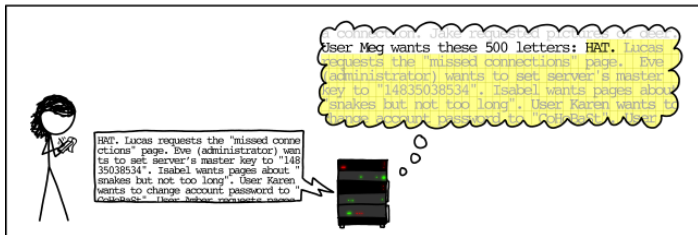
(xkcd.com)

# Buffer Overflows (Variante, Beispiel Heartbleed)



(xkcd.com)

# Buffer Overflows (Variante, Beispiel Heartbleed)



(xkcd.com)

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- Code Execution
- Cross-Site Scripting

- **Ziel:** Verfügbarkeit eines Systems angreifen
  - Schädigt insbesondere Systeme, bei denen Verfügbarkeit finanziell kritisch ist (Online-Banking, Online-Shopping)
- Naheliegend: System durch viele Anfragen überlasten
  - Beispiel: Botnetz (d.h. Menge von Viren infizierter und „gekaperter“ Rechner) greift ständig auf Online-Dienst zu
  - Lässt sich prinzipiell nur schwer verhindern (vor allem bei Botnetz-Angriffen)
- Aber auch geschicktere Angriffe möglich

# Hash Table Collisions

- **Szenario:** geschickte Anfragen an Webserver/-dienst stellen, die möglichst viel Zeit zur Verarbeitung brauchen
- HTTP-GET-/-POST-Anfragen von Anwender setzen Variablen
  - Beispiel: GET-Anfrage von Benutzer  
`http://www.google.com/search?q=mad+magazine`  
setzt Variable q auf „mad magazine“
- Programm (z.B. in PHP, Python) verarbeitet Anfrage und liest Variablen in Dictionary-Datenstruktur ein
- **Frage:** was könnte dabei schiefgehen?
- Zunächst genauer verstehen, was da passiert

- **Datenstruktur Dictionary:** assoziatives Array

anfrage['q'] = 'mad magazine'  
anfrage['client'] = 'ubuntu'  
... = ...

- Wird intern in Array fixer Länge gespeichert:

$\overline{\text{anfrage}}[h('q')] = ('q', 'mad magazine')$   
 $\overline{\text{anfrage}}[h('client')] = ('client', 'ubuntu')$

für (nicht-kryptographische) Hashfunktion  $h : \{0, 1\}^* \rightarrow \mathbb{Z}_n$

- Dictionaries intern in „normales“ Array konvertiert:

$$\overline{\text{anfrage}}[h('q')] = ('q', 'mad magazine')$$
$$\overline{\text{anfrage}}[h('client')] = ('client', 'ubuntu')$$

für (nicht-kryptographische) Hashfunktion  $h$

- **Frage:** was passiert bei  $h$ -Kollisionen?
- **Antwort:** ersetze Key-Value-Paar durch *Liste* von Key-Value-Paaren



- Kosten von  $n$  Zugriffen auf Dictionary  $\mathbf{O}(n)$ ,  
wenn keine  $h$ -Kollisionen auftreten
- Kosten von  $n$  Zugriffen auf Dictionary  $\mathbf{O}(n^2)$ ,  
wenn nur  $h$ -Kollisionen auftreten
- Bei „normalen“ Eingaben treten wenige  $h$ -Kollisionen auf  
⇒ üblicherweise Dictionaries sehr effizient
- **Aber:** bei „ungünstigen“ Dictionary-Keys (Variablennamen)  
mit vielen  $h$ -Kollisionen Dictionaries sehr ineffizient

# Dictionaries und Denial of Service

- **Angriff:** wähle „ungünstige“ Variablennamen in HTTP-Anfrage  $\Rightarrow$  lange Verarbeitungsdauer, weil Variablen-Wert-Paare in Dictionary eingelesen werden
- **Beachte:**  $h$  nicht kollisionsresistent
- **Beispiel:** (Quelle: 27C3-Vortrag „Effective Denial of Service attacks against web application platforms“)
  - PHP: Anfragen mit 70-100kb/s lasten i7-Kern voll aus
  - ASP.NET: Anfragen mit 30kb/s lasten Core2-Kern voll aus
  - Java (Tomcat), Python ähnlich, Ruby katastrophal
- **Was tun?**
  - $h$  randomisieren (kryptographische Hashfunktion zu langsam)
  - Anzahl Variablen bei Anfragen einschränken

## 1 Analyse größerer Systeme

- Motivation
- Der Security-Zugang
- Der kryptographische Zugang
- Zusammenfassung

## 2 Kurzüberblick häufige Sicherheitslücken

- Motivation
- Buffer Overflows
- Denial of Service
- **Code Execution**
- Cross-Site Scripting

- **Ziel:** lasse eigenen Code beim Opfer ausführen
- **Beispiel:** Buffer Overflow auf Stack (siehe letzte VL)
- **Warum ist das schlimm?**
  - (Lese-/Schreib-)Zugriff auf anderes System erlangen
  - Anderes System kontrollieren (→ Botnetze)
  - Andere impersonieren
- Überlappungen mit anderen Softwarefehlern
- Deshalb hier nicht weiter behandelt

- 1 Analyse größerer Systeme
  - Motivation
  - Der Security-Zugang
  - Der kryptographische Zugang
  - Zusammenfassung
- 2 Kurzüberblick häufige Sicherheitslücken
  - Motivation
  - Buffer Overflows
  - Denial of Service
  - Code Execution
  - Cross-Site Scripting

- **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)