Vorlesung Sicherheit

Dennis Hofheinz

ITI, KIT

10.07.2014

Überblick

Überblick

Erinnerung

- Bislang Bausteine betrachtet
- Zwei Heransgehensweisen zur Analyse größerer Systeme
- **Security-Zugang:** Eigenschaften checken (CIA)
- **Kryptographischer Zugang:**
  - Vergleiche System mit Idealisierung
  - Kern: Relation „$\geq$“ („mindestens so sicher wie“) auf Protokollen

Überblick

Das CIA-Paradigma

- 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

Überblick

Motivation

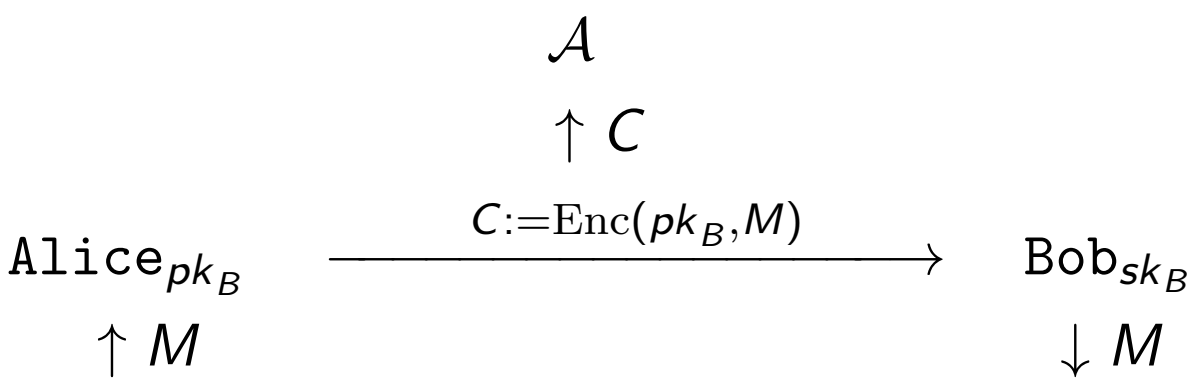
- **Beobachtung:** Manchmal Betrachtung von einzelnen Eigenschaften (CIA) problematisch
  - Manche Eigenschaften (Nicht-Abstreitbarkeit) nicht berücksichtigt
  - Wann ist „Liste der wünschenswerten Eigenschaften“ vollständig?
- **Ziel:** genereller Sicherheitsbegriff, der nicht auf einzelne Eigenschaften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

Grundidee

- **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 „π_1 mindestens so sicher wie π_2“...
  - ... oder „π_1 implementiert/realisiert π_2“
  - Dabei üblicherweise π_1 reales Protokoll, π_2 Idealisierung

Grundidee

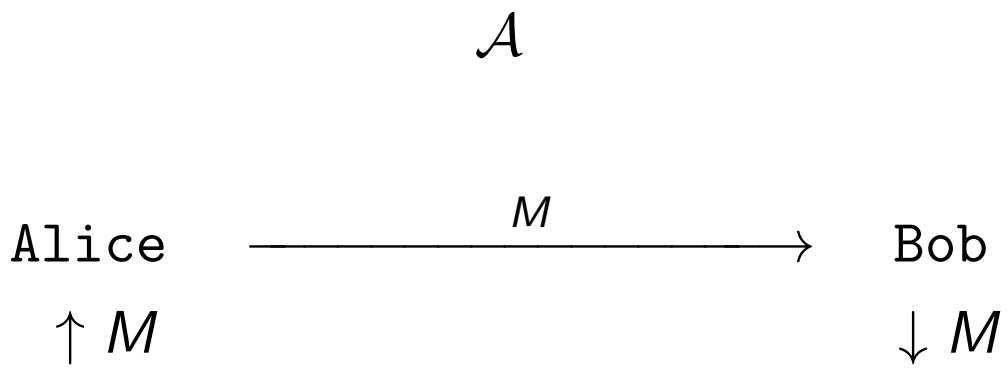
- **Beispiel:** Reales Protokoll π_1 nutzt PKE-Verfahren



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

Grundidee

- Ideales Protokoll π_2 modelliert sichere Kommunikation



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

Grundidee

- **Frage:** Ist π_1 so sicher wie π_2? (Gilt $\pi_1 \geq \pi_2$?)
- Was bedeutet „$\geq$“ eigentlich? Was sollte $\geq$ bedeuten?
- **Intuition:** $\pi_1 \geq \pi_2$ wenn...
  - ... π_1 so sicher wie π_2
  - ... jede Schwäche von π_1 auch schon in π_2
  - ... für jeden Angreifer $\mathcal{A}_1$ auf π_1 ein Angreifer („Simulator“) $\mathcal{A}_2$ auf π_2 existiert, so dass die Effekte in beiden Fällen gleich
  - ... aber was heißt „... so dass die Effekte gleich“?

Definition Simulierbarkeit

Definition (Simulierbarkeit, informell)

Protokoll π_1 ist *so sicher wie* Protokoll π_2 (kurz: $\pi_1 \geq \pi_2$), falls
für jeden effizienten Angreifer $\mathcal{A}_1$ auf π_1
ein effizienter Angreifer $\mathcal{A}_2$ auf π_2 existiert, so dass
nicht effizient zwischen $(\pi_1, \mathcal{A}_1)$ und $(\pi_2, \mathcal{A}_2)$ unterschieden
werden kann.

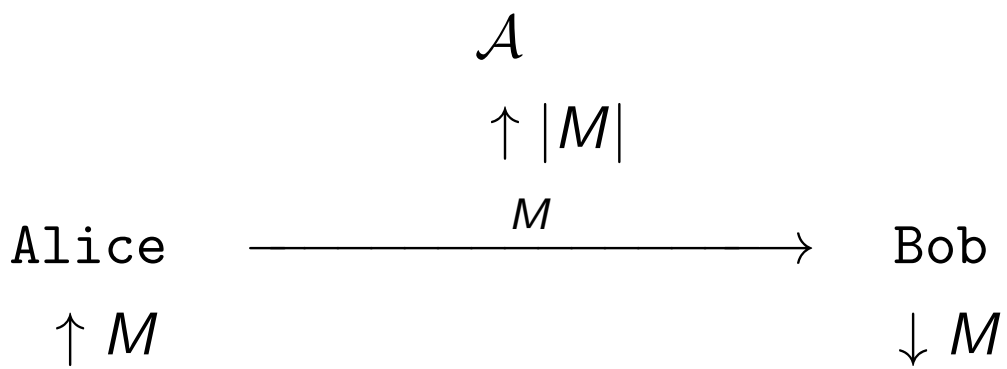
- (Rot bedeutet: so noch nicht genau definiert)
- Ununterscheidbare Protokolle: können „von außen“ (aus Sicht eines Protokollbenutzers) nicht unterschieden werden

Beispiel Simulierbarkeit

- **Frage:** gilt für obige Protokolle $\pi_1 \geq \pi_2$?
- **Antwort:** nein
  - Grund: π_1 lässt Angreifer wissen, *dass* Kommunikation stattfindet
  - Aber: π_2 lässt Angreifer dies nicht wissen
  - Konsequenz: Angreifer $\mathcal{A}_1$ auf π_1, der nur zum Benutzer signalisiert, wenn Kommunikation stattfindet, kann nicht durch ein $\mathcal{A}_2$ simuliert werden

Beispiel Simulierbarkeit

- Geändertes Protokoll π_2':



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

Beispiel Simulierbarkeit

- Für obige π_1, π_2' gilt $\pi_1 \geq \pi_2'$, falls das in π_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 π_1 erhält $\mathcal{A}$ das Chiffrat C
  - Tatsächlich gibt C Aufschluss über $|M|$ (Nachrichtenlänge kann nicht komplett verborgen werden)
  - Steht diese Information nicht auch in π_2' zur Verfügung, erlaubt π_1 mehr „Angriffe“ als π_2'

Modulare Analyse

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

Modulare Analyse

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

Überblick

Zusammenfassung

- 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

Aktuelle Forschung

- 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?

Überblick

Überblick

Motivation

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

Die CVE-Datenbank

- 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

Überblick

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 $\Rightarrow$ `adresse` wird überschrieben

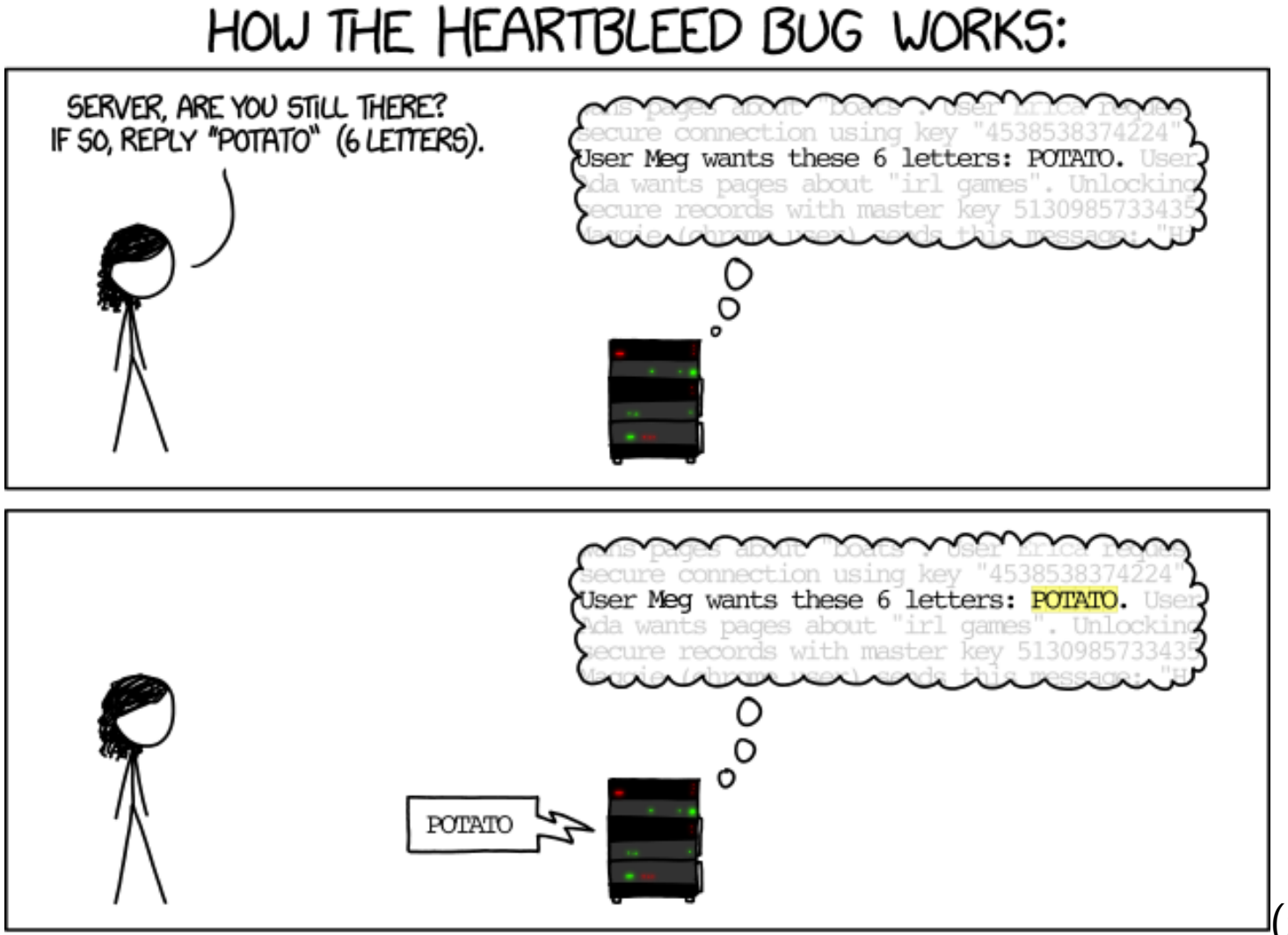
Buffer Overflows

- 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)

Buffer Overflows

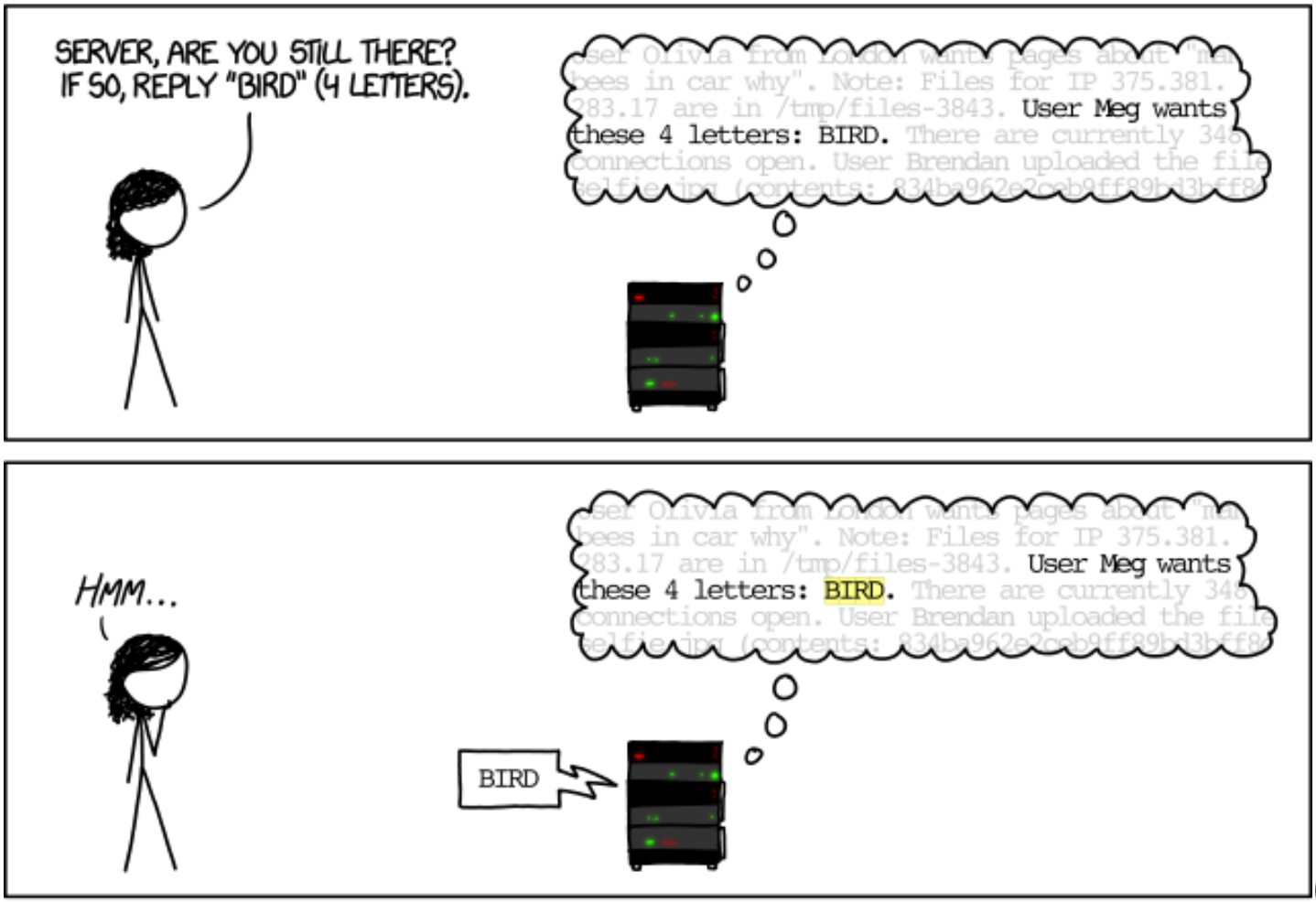
- **Gegenmaßnahmen?**
- Benutzung sichererer Programmiersprache (Java, Python, . . .)
- Wenn nicht möglich: Benutzung von Routinen, die explizite Längengrenzen von verlangen/überprüfen
- DEP (Data Execution Prevention): verhindern, dass Code auf Stack ausgeführt wird (Trennung von Code- und Datenbereichen)
  - 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)



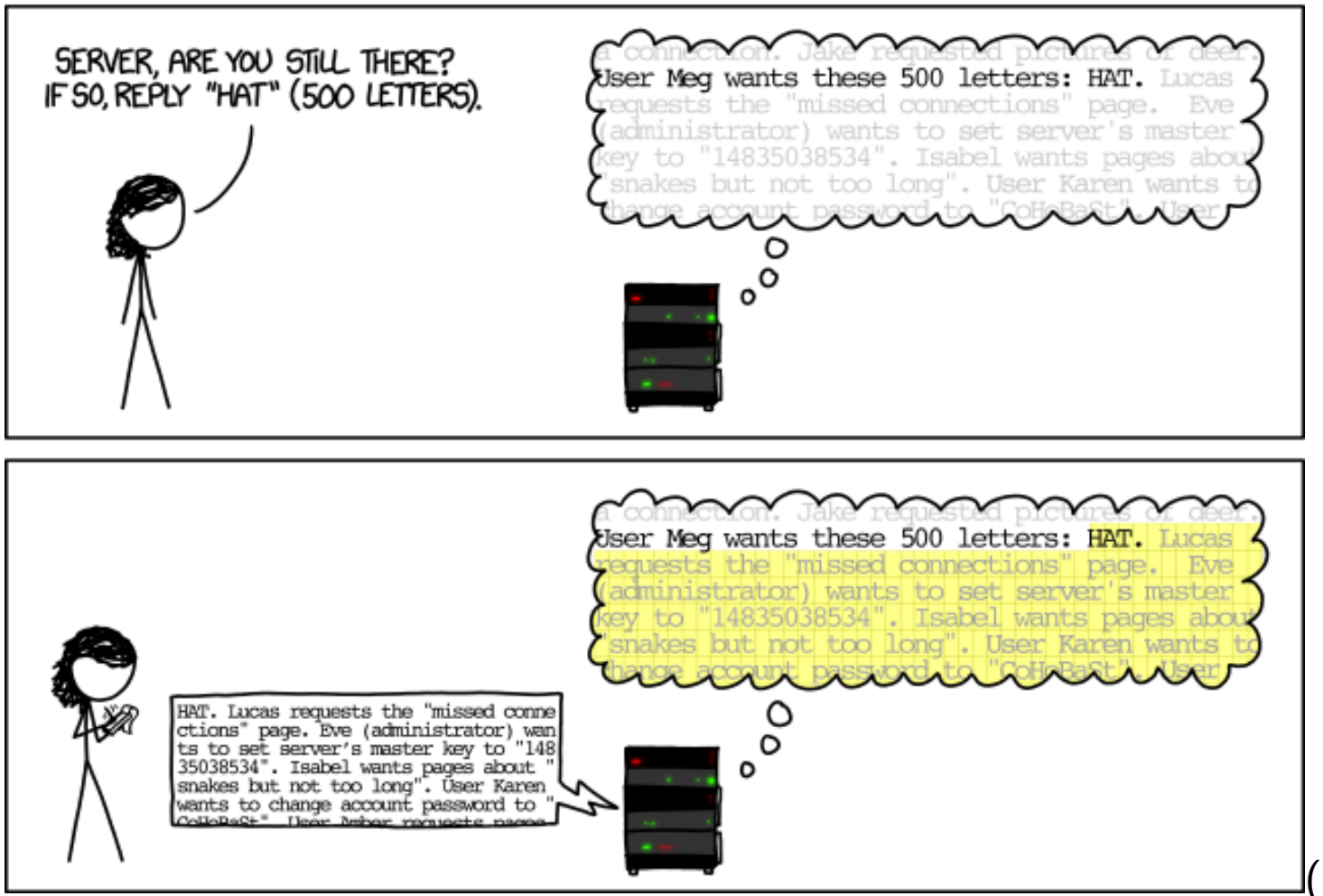
(xkcd.com)

Buffer Overflows (Variante, Beispiel Heartbleed)



(xkcd.com)

Buffer Overflows (Variante, Beispiel Heartbleed)



(xkcd.com)

Überblick

Denial of Service

- **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

Dictionaries

- **Datenstruktur Dictionary:** assoziatives Array

$$\begin{aligned} \texttt{anfrage['q']} &= \texttt{'mad magazine'} \\ \texttt{anfrage['client']} &= \texttt{'ubuntu'} \\ \ldots &= \ldots \end{aligned}$$

- Wird intern in Array fixer Länge gespeichert:

$$\begin{aligned} \overline{\texttt{anfrage}}[h(\texttt{'q'})] &= (\texttt{'q'}, \texttt{'mad magazine'}) \\ \overline{\texttt{anfrage}}[h(\texttt{'client'})] &= (\texttt{'client'}, \texttt{'ubuntu'}) \end{aligned}$$

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

Dictionaries

- Dictionaries intern in „normales“ Array konvertiert:

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

  für (nicht-kryptographische) Hashfunktion h
- **Frage:** was passiert bei h-Kollisionen?
- **Antwort:** ersetze Key-Value-Paar durch *Liste* von Key-Value-Paaren

Dictionaries

- 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
  $\Rightarrow$ 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