Grundlagen der Systemsoftware

Modul: InfB-GSS

Veranstaltung: 64-091

Utz Pöhlmann
4poehlma@informatik.uni-hamburg.de
6663579

Louis Kobras
4kobras@informatik.uni-hamburg.de
6658699

Marius Widmann
4widmann@informatik.uni-hamburg.de
6714203

22. Juni 2016

Zettel Nr. 5 (Ausgabe: 13. Juni 2016, Abgabe: 22. Juni 2016)

Aufgabe 1: Rechnersicherheit

1)

Aufgabe a)

Zugriffskontrolle:

- Setzt die Zugangskontrolle voraus und gewährleistet Zugriff nur, falls der Nutzer das Recht auf den Zugriff zu Objekt oder Ressource x hat
- Beispiel: Mandatory Access Control eines Unix-Systems

Zugangskontrolle:

- Zugangskontrolle stellt in der Informatik sicher, dass ein Rechner Kommunikation nur mit berechtigten Benutzern oder Rechnern erlaubt
- Beispiel: Login (Username/Passwort) an einer Workstation

Aufgabe b)

Es ist sinnvoll, sofern nur eine Person - oder nur Personen einer "Sicherheitsstufe"- Zugang zu dem IT-System haben, da die Zugangskontrolle dazu da ist, um sicherzustellen, dass es sich um einen berechtigten Nutzer handelt. Dann kann man sich allerdings unbekümmert an den Computer setzen und ausnahmslos alle Daten auslesen, editieren und/oder löschen.
Somit ist es nicht sinnvoll, falls mehrere Personen mit unterschiedlichen "Sicherheitsstufen" an einem IT-System arbeiten und/oder man verhindern will, dass Personen auf vertrauliche Daten anderer zugreifen können.

Aufgabe c)

Die Zugangskontrolle identifiziert eine Person, die Zugriffskontrolle verifiziert ob die Person auf ein bestimmtes Objekt zugreifen darf. Ohne eine Identifikation kann die Zugriffskontrolle keine Rechte überprüfen, da die Anfrage durch eine unbekannte Person erfolgt.

Aufgabe d)

Man wird über die IP Identifiziert.
Bzw. man kann nur über den Link auf ein Dokument zugreifen und der gibt man nicht einfach so weiter (vorausgesetzt, es besteht keine kriminelle Absicht der Leute, denen die Adresse anvertraut wurde).

Weitere Erklärungen:

- Man wird über die IP und das Wissen über die Link-Adresse identifiziert. Das ist die Zugangskontrolle
- Der Link ist öffentlich, daher ist jeder Zugriff ein Zugriff eines Gastes, d.h. die Zugriffskontrolle hat Rechte für Gäste spezifiziert.

2)

Aufgabe a)

Es werden Bild und Fingerabdrücke gespeichert. Anhand der erkennbaren Muster kann eine biometrische Identifizierung erfolgen.

Aufgabe b)

- Man könnte ein Foto vor die Webcam halten
- Die Überprüfung wird lokal gemacht dort könnte man in die Software eingreifen und das Signal beeinflussen
- Sicherungsmaßnahme um die Daten der Karte zu sichern: Die Karten sollte read-only sein (ROM) und zusätzlich versiegelt werden, sodass sie nur noch lesbar sind. Zudem verhindern, dass der Chip irgendwie ausgebaut werden kann.

3)

Aufgabe a)

Schwachstellen:	die Tippmuster könnten ähnlich einer einer Unterschrift durch gutes Beobachten und Zuhören von einem Dritten nachgeahmt werden.
Verhalten:	passiv
Verbreitung:	keinen Zugriff auf die Datenbank der Tipp-Profile
Rolle:	Außenstehender
Ressourcen:	begrenzt (nicht genug um um den Algorithmus, der der Verfizierung zugrundeliegt zu knacken)

Aufgabe b)

- Zugangs- und Zugriffskontrolle der Datenbank
- Verschlüsselter Datentransfer
- sicherer Handshake bei der Authentifizierung

4)

Aufgabe a)

Rolle:	Kunde
Verbreitung:	Zugriff auf den Dienst der Website
Verhalten:	passiv
Ressourcen:	Begrenzt (nicht genug, um die Website zu hacken und den Algorithmus auszulesen)

Aufgabe b)

- Barcodes werden in Kino selbst - eventuell zufällig - erzeugt (lokal)
- Daten über erzeugte Barcodes werden lokal gespeichert (an anderer Stelle)
- Prüfung des Barcodes erfolgt lokal (durch Zugriff auf den lokalen Rechner)
- Erfolgreiche Barcodes werden gelöscht

Aufgabe 2: Timing-Attack

1)

Annahmen: wir gehen OBDA davon aus, dass wir wissen,

- dass das Passwort höchstens 20 Zeichen lang ist.
- dass es nur alphanumerische Zeichen enthält.
- dass unser Passwort “password” ist.

- dass das Passwort mindestens 1 Zeichen lang ist.

Code: siehe Anhang

2)

Das Programm prüft die erste Stelle des Passworts. Ist diese Korrekt, wird die nächste geprüft, sonst nicht. Somit bricht das Programm an unterscheidlichen Stellen ab und bearbeitet die Passwörter anhand ihrer Korrektheit unterscheidlich lange.

3)

Gehe zu Schritt 1 mit X = 1 und Y leer.

Schritt 0: (nur bei manchen Implementationen sinnvoll): Länge des Passwortes prüfen, indem verschieden lange Passwörter erzeugt und geprüft werden. Das Passwort, bei dem das Programm länger zum antworten braucht hatte die richtige Länge. Braucht das Programm jeweils in etwa gleich lange, kann man an dieser Stelle keine Rückschlüsse auf die Passwortlänge ziehen.

Schritt 1: alle Möglichkeiten für Stelle X generieren und den Anfang des Passwortes Y davorhängen und das Passwort dann prüfen. Der Rest des Passwortes ist egal.

Schritt 2: bei einem Passwort hat das Programm länger zum Antworten gebraucht. Dies ist die Richtige Stelle und wird an Y angehängt.

Schritt 3: wenn das Passwort noch nicht geknackt wurde, gehe zu Schritt 1 mit dem neuen Y und X= X+1

4)

Annahme: counter setzen benötigt keine Zeit:

```
boolean passwordCompare(char[] a, char[] b)
{
    int i;
    int counter;
    for(i = 0; i<a.size(); i++)
    {
        if(a[i] == b[i])
        {
            counter++;
        }
    }
    return counter == b.size()
}
```

Annahme: counter setzen benötigt Zeit:

```
boolean passwordCompare(char[] a, char[] b)
{
    int i;
    int counter = 0;
    int fakeCounter = 0;
    for(i = 0; i<a.size(); i++)
    {
        if(a[i] == b[i])
        {
            counter++;
        }
        else
        {
            fakeCounter++;
        }
    }
```

```
17      return counter == b.size()
18  }
```

Aufgabe 3: Real-World-Brute-Force Angriff

1)

Code sieht folgendermaßen aus:

- fffjj-DEMO-XXXXX-XXXXX-XXXXX-XXXXX

Davon sind:

- 3 Zeichen Fach
- 2 Zeichen Jahr
- 1 Bindestrich Aufgabenblatt (hier 4 Zeichen: DEMO)
- 20 Zeichen Code (4 Segmente zu 5 Zeichen, getrennt mit Bindestrich)

Für jedes Zeichen gibt es 36 Möglichkeiten = 1^{36}
$\Rightarrow$ Für den gesamten Code gibt es 20 Zeichen = 20^{36} Möglichkeiten. Das sind ca. 68 Septilliarden Möglichkeiten.
Die Wahrscheinlichkeit, einen gültigen Code zu erraten liegt bei 100 gültigen Codes durch 20^{36} mögliche Codes $\approx$ 0.00%

2)

Tatsächlicher Zeichenumfang:
26 lateinische Buchstaben + 10 arabische Ziffern = 20^{36} verschiedene Passwortmöglichkeiten
Angenommen, die bereits ausgegebenen Passwörter sind gleichmäßig auf diesen Bereich verteilt, dann müsste man, um mindestens 1 Passwort zu knacken im Durchschnitt mindestens $\frac{20^{36}}{100}$ Passwörter probieren.
Das macht bei 1000 Passwörter pro Sekunde:
$\frac{\frac{20^{36}}{100}}{1000}$ ca. 687 Sextilliarden ($32 \cdot 20^{31} \approx 6.87 \cdot 10^{41}$) Sekunden $\widehat{\approx} 2.17 \cdot 10^{34} \approx$ 22 Quintilliarden Jahre

ANHANG:

```java
package Blatt5.Aufgabe3;

public class Timer
{
    private final char[] _passwort = "abcdefgh".toCharArray();
    private boolean _gefunden = false;
    private int _passwortLaenge;
    private final char[] _symbole = "abcdefghijklmnopqrstuvwxyz0123456789"
        .toCharArray();
    private final int _maxPasswortLaenge = 20;

    public static void main(String[] args)
    {
        Timer timer = new Timer();
        timer._passwortLaenge = timer.findePasswortLaenge();
        System.out.println(timer._passwortLaenge);
        System.out.println(timer.findePasswort());
    }

    /**
    * Findet das Passwort. Period.
    * @return Das Passwort.
    * @requirements: Die Passwortlaenge wurde bestimmt.
    */
    private String findePasswort()
    {
        String laufPasswort = "";
        String bisherigesPasswort = "";
        long[] zeiten = new long[_passwortLaenge];
        long startZeit;
        long endZeit;
        do
        {
            // Zaehlt einen Zaehler bis zur bestimmen Passwortlaenge
            for (int laenge = 0; laenge < _passwortLaenge; laenge++)
            {
                // Zaehlt einen Zaehler bis zur Laenge des Eingabealphabets
                for (int zaehler = 0; zaehler < _symbole.length; zaehler++)
                {
                    // haengt das aktuelle Laufsymbol an das Passwort
                    laufPasswort = bisherigesPasswort + _symbole[zaehler];
                    startZeit = System.nanoTime();
                    // Hier wird das Passwort geprueft
                    _gefunden = passwordCompare(laufPasswort.toCharArray(), _passwort);
                    endZeit = System.nanoTime();
                    // Berechnet Zeitdifferenz fuer aktuelles Passwort
                    zeiten[zaehler] = endZeit - startZeit;
                }
                // haengt an das bisherige Passwort dasjenige Symbol an, fuer welches die
                    groesste Zeit gebraucht wurde
                bisherigesPasswort += _symbole[gibIndexVonMaximum(zeiten)];
            }
        }
        while (!_gefunden);
        return bisherigesPasswort;
    }

    /**
    * Bestimmt die Laenge eines Passworts
    * @return die Laenge eines Passworts
    */
    private int findePasswortLaenge()
    {
        String passwort = "";
        long[] zeiten = new long[_maxPasswortLaenge];
        long startZeit;
```

```
66          long endZeit;
67          System.out.println("Starte Schleife...");
68          for (int zaehler = 1; zaehler < _maxPasswortLaenge; zaehler++)
69          {
70              passwort.concat("a");
71              startZeit = System.nanoTime();
72  //          System.out.println("Startzeit: " + startZeit);
73              // prueft, ob das Passwort korrekt ist
74              passwordCompare(passwort.toCharArray(), _passwort);
75              endZeit = System.nanoTime();
76  //          System.out.println("Endzeit: " + endZeit);
77              zeiten[zaehler] = endZeit - startZeit;
78              System.out.println("Zeitdifferenz: " + zeiten[zaehler]);
79          }
80          System.out.println("Nach Schleife");
81          return gibIndexVonMaximum(zeiten);
82      }
83
84      /**
85      * Gibt den Index des hoechsten Wertes zurueck
86      * @param array Eingabe, das nach dem hoechsten Wert durchsucht werden soll
87      * @return den Index des hoechsten Wertes
88      */
89      private int gibIndexVonMaximum(long[] array)
90      {
91          int index = 0;
92          for (int i = 0; i < array.length; i++)
93          {
94              if (array[i] > array[index])
95              {
96                  index = i;
97              }
98          }
99          return index;
100     }
101
102     /**
103     * Vergleicht zwei Char-Arrays
104     * @param a erstes Array
105     * @param b zweites Array
106     * @return true, wenn sie gleich sind; false sonst
107     */
108     boolean passwordCompare(char[] a, char[] b)
109     {
110         int i;
111         if (a.length != b.length) return false;
112         for (i = 0; i < a.length && a[i] == b[i]; i++)
113             ;
114         return i == a.length;
115     }
116 }
```