



# Sicherheit von Ruby On Rails

**Heiko Webers**

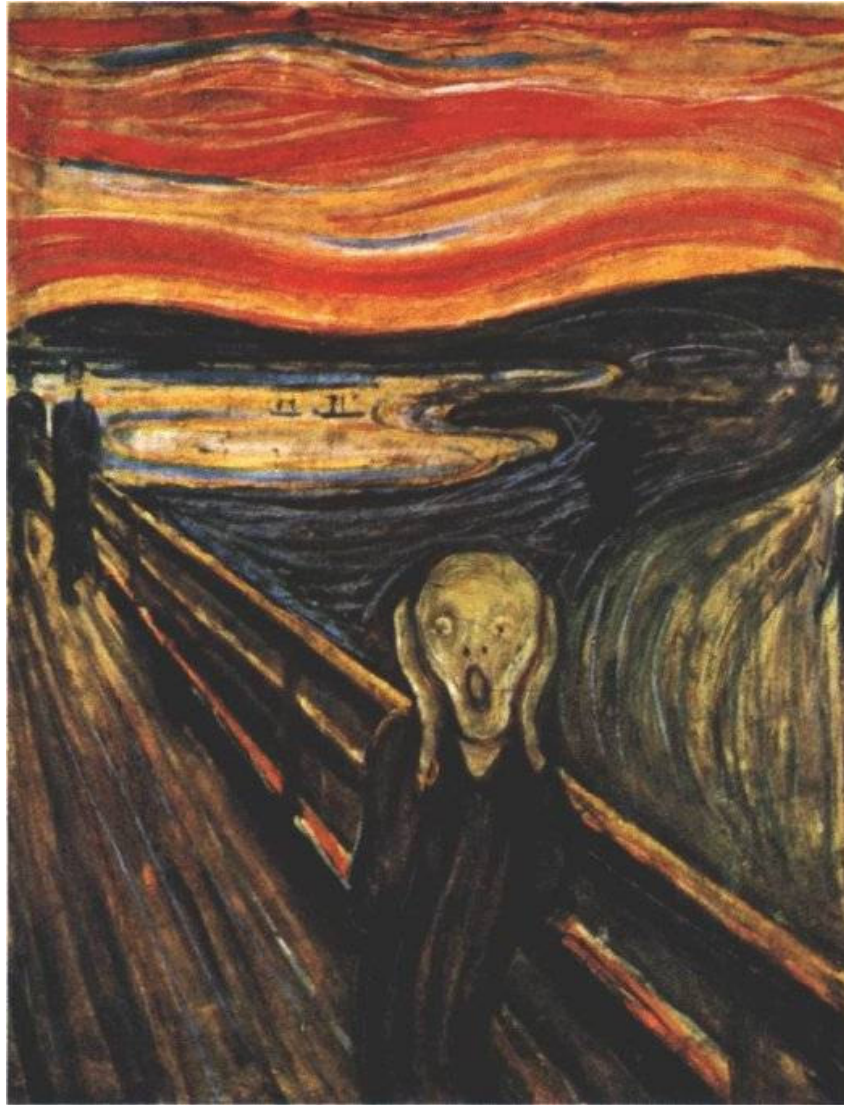
# Heiko Webers

- Ruby On Rails Security Project: [www.rorsecurity.info](http://www.rorsecurity.info)
- Autor von „Ruby On Rails Security“, ab August 2007 bei der OWASP
- Ruby On Rails Beratung mit Sicherheit
- Softwarefirma

## Warum Sicherheit?

- „Warum sollte gerade meine Seite ...?“, weil große Firmen immer sicherer werden, Teil eines großen Angriffs, nicht alle Angriffe kommen von außen
- 2002: 90% der Firmen und Regierungsstellen melden Sicherheitsverletzungen, 80% davon mit finanziellen Folgen
- Wiederherstellung kostet Zeit und Geld

# Angst



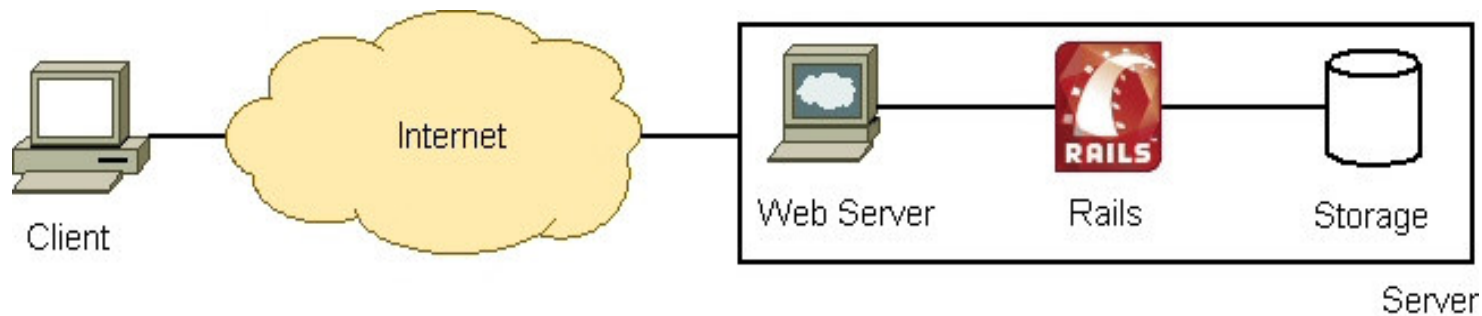
# Gefahren für Web Anwendung

- **Marktforscher Gartner Group: 75% der Angriffe ereignen sich auf der Ebene der Web Anwendungen, von 300 analysierten Seiten waren 97% anfällig für Angriffe**
- **Nicht erst handeln, wenn es zu spät ist**

# Gefahren für Web Anwendung

- CardSystems 2004: 263,000 Kreditkartennummern gestohlen
- Online-Vertragsverwaltung der Telekom 2004: Zugriff auf fremde Verträge über Monate, Administratorzugriff, Vollzugriff auf Server im Telekomnetz
- CNBC 2007: \$1,000,000 Börsenwettbewerb gehackt
- eBay 2004: Sicherheitsdienstleister testet 220 weibliche Vornamen als Passwörter bei 2600 eBay-Accounts, 27 mal erfolgreich

# Schichtenmodell



- Sicherheit von Ruby On Rails Anwendungen abhängig von den typischen drei Schichten:
  - Webserver, hier Apache
  - Datenbank Management System, hier MySQL
  - Ruby On Rails

# Sicherheitsmaßnahmen für Apache

- Nur die tatsächlich benötigten Module aktivieren
- Apache mit den Zugriffsrechten eines speziellen Unix Benutzers ausführen: Im Falle einer Kompromittierung kein Vollzugriff auf das Betriebssystem
- Verzeichnisse und Dateien: „Im Allgemeinen schützen, im Besonderen freigeben“

# Sicherheitsmaßnahmen für MySQL

- MySQL-Server ebenfalls mit den Zugriffsrechten eines speziellen Unix Benutzers ausführen
- Mit `bind-address = 127.0.0.1` Verbindungen nur von localhost erlauben (Rails auf dem Server)
- Speziellen MySQL-Benutzer erstellen, der nur eingeschränkten Zugriff auf die Datenbank der Rails Anwendung hat

# Sicherheit von Ruby On Rails



# Profiling

- Ziel: Wie funktioniert die Web Anwendung, wie ist sie aufgebaut
- Betriebssystem, Web Server, Datenbank Server, Programmiersprache + Framework, Verzeichnisstruktur, Controller, Actions, URL-Parameter, Datenbanktabellen und -felder, ...

# Profiling: Hilfsmittel

- Analysetools, Kommentare im Quelltext, vergessene Dateien und Controller, Debug-Actions
- Robots.txt

```
User-agent: *  
Disallow: /admin/  
Disallow: /catalog/admin  
Disallow: /private
```
- Google, Google Hacking Database, The Wayback Machine

# Profiling: Hilfsmittel

- URL-Parameter:

`http://www.domain.com/project/1/show?userId=1&returnTo=www.domain.com&file=/docs/project1.doc`

- `userId=42`
- `returnTo=www.attacker.com`
- `file= ../../../../etc/passwd`

# Interpreter Injection

- Schadhaften Code in die Anwendung einschleusen (injizieren), damit er im Sicherheitskontext der Anwendung ausgeführt wird
- Plätze 1 und 2 in der OWASP Top Ten

---

# Interpreter Injection: Übersicht

- **User Agent Injection**
- **SQL Injection**

# User Agent Injection

- Auch: Browser Injection, Cross Site Scripting (XSS)
- Injektion: HTML, meist in Verbindung mit JavaScript, aber auch andere Formate, die der Browser interpretiert
- Wo: Forum, Kommentare, aber auch Überschriften, Benutzernamen, Suchergebnisse der Web Anwendung
- In der URL für einen unmittelbaren Effekt, in Parametern, die gespeichert werden für einen mittelbaren Effekt
- Wie kommt das Opfer auf die manipulierte Seite: E-Mail, Foreneintrag, Kommentare etc.

# User Agent Injection: Ziele

- Defacement: Hier Austausch von Elementen, um Opfer auf Seiten des Angreifers zu locken
- Original: [Sign Up](#) | [My Account](#) | [History](#) | [Help](#) | [Log In](#)
- Fälschung: [Sign Up](#) | [My Account](#) | [History](#) | [Help](#) | [Log In](#)
- Mittel: CSS und HTML-Injektion

## Exkurs: Cookies

- HTTP Protokoll zustandslos
- Zustand: Benutzereinstellungen, eingeloggter Nutzer
- Zustand wird im Daten-Hash `session` gespeichert:  
`session[:user_id]`
- Identifiziert durch Session ID
- Cookie: `_session_id=16d5b78abb28e3d6206b60f22a03c8d9`
- Cookies können gestohlen werden!

# User Agent Injection: Cookies stehlen

- Cookies: In Javascript `document.cookie`, Same Origin Policy, durch Injektion wird Skript Teil des Dokuments und kann auf alles zugreifen

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

- Ganz normaler Image-Tag? `<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#83;&#83;&#83;&#39;&#41;>`

# User Agent Injection: Gegenmaßnahmen

- Falls Textformatierung benötigt: RedCloth benutzt eigene Auszeichnungssprache, `_test_` wird umgewandelt in HTML `<em>test<em>`
- Falls HTML: Markdown (sehr begrenzte Tag-Auswahl)
- Falls volles HTML
  - Blacklist-Filter, aber: `<script/src=..., onclick=...`
  - Whitelist-Filter: Rails Plugin
- Gar kein HTML: `h (@page.name)`

---

# Interpreter Injection: Übersicht

- User Agent Injection
- SQL Injection

# SQL Injection

- Injizierung von SQL Befehlen in Parameter, mit dem Ziel Datenbankabfragen zu manipulieren
- Zugangsschutz umgehen
- Unbefugter Zugriff
- Manipulation von Daten
- Lahmlegen des Servers

# SQL Injection: Unbefugter Zugriff

```
Project.find(:all,  
  :conditions => "name = '#{params[:name]}' AND  
                 user = 3")
```

```
SELECT * FROM projects WHERE (name = 'abrechnung'  
AND user = 3)
```

```
' OR 1=1)--
```

```
SELECT * FROM projects WHERE (name = '' OR 1=1)--'  
AND user = 3)
```

# SQL Injection: Eingabefilter

- Kein Suchen & Ersetzen!
- SELECT ersetzen, aber SELSELECTECT
- '1'=1' ersetzen, aber '2'='2'
- Besser: Überprüfung und ggf. Zurückweisung, dazu später mehr
- Und: ...

# SQL Injection: Gegenmaßnahmen

- SQL Injection steht und fällt mit ' , " , NULL und Zeilenumbrüchen
- Rails wendet Filter normalerweise automatisch an und wandelt diese Zeichen um. Nicht aber in `:conditions => "..."`, `connection.execute()`, `find_by_sql()`
- Hier nicht `string1 + string2` und `#{Variable}` verwenden, sondern Array:

# SQL Injection: Gegenmaßnahmen

- Syntax: [*String mit Platzhaltern, Variablen*]

```
User.find(:first, :conditions => ["login = ? AND  
password = ?"], params[:name], params[:password]))
```

- Seit Rails 1.2: Conditions Hash

```
User.find(:first, :conditions => {:login =>  
params[:name], :password => params[:password]})
```

# Interpreter Injection: Eingabefilter

- Wie wir sehen: **Alle** Parameter, die vom Benutzer kommen sind als schädlich und manipuliert anzusehen bis das Gegenteil bewiesen wurde
- Überprüfung auf der Client-Seite per JavaScript sinnlos
- Model kann geprüft werden *validates\_numericality\_of*
- Aber auch im Controller Prüfung notwendig

# Interpreter Injection: Eingabefilter

```
If params[:id] && params[:id].to_i > 0 Then ...
```

- Oder einfachen Parseparam() Filter: prüft Datentyp, erlaubte Werte, minimale und maximale Länge, Format

```
parseparam(vpstr, vdefault, vtype, [vpositivelist],  
           [vmatchregexpr], [vmin], [vmax])
```

# Interpreter Injection: Eingabefilter

```
gender = parseparam( params[:gender], "f", "str",  
  ["m", "f"])
```

```
file = parseparam( params[:file], "", "str", nil,  
  /^[^w\.\-\+]+$/)
```

```
params[:file] =  
  "file.txt%0A<script>alert('hello')</script>"
```

```
/\A[^\w\.\-\+]+\z/
```

# Logic Injection

- Interpreter Injection injiziert Code in einen Interpreter (Browser, DBMS etc.)
- Logic Injection: Versuch die Logik der Anwendung zu manipulieren

# Logic Injection: Unerlaubter Zugriff

- Rails URLs: `http://www.domain.com/project/show/1`
- Unterschiedliche Berechtigungen: nicht auf fremde Projekte zugreifen
- Niemand kann verhindern, dass Benutzer `http://www.domain.com/project/show/2` eingibt

# Logic Injection: Unerlaubter Zugriff

- Meist `@project = Project.find(params[:id])`

```
@project = Project.find_by_id_and_user_id(  
  params[:id], session[:user_id])
```

# Logic Injection: Unerlaubter Zugriff

```
def current_user
  User.find(session[:user_id])
end
```

```
class User < ActiveRecord::Base
  has_many :projects
end
```

```
@project = current_user.projects.find(params[:id])
```

# Logic Injection: Benutzerverwaltung

- Generatoren: LoginGenerator, LoginSugar, Restful Authentication, ...
- Password Cracker: über 30,000 Passwörter pro Minute
- Weitere Sicherheitshinweise dazu im Rails Security Project

# Logic Injection: Massenzuweisung

```
<input id="user_first_name" name="user[first_name]"  
  size="30" type="text" />
```

```
@user = User.new(params[:user])
```

```
<input id="user_verified" name="user[verified]"  
  type="hidden" value="1" />
```

```
<input id="user_role" name="user[role]" type=  
  "hidden" value="admin" />
```

# Logic Injection: Massenzuweisung

- Werte einzeln zuweisen:

```
User.new(:first_name =>
  params[:user][:first_name])
```

- Im Model:

```
attr_accessible :first_name
```

```
user = User.new(:first_name => "Heiko",
  :verified => true)
```

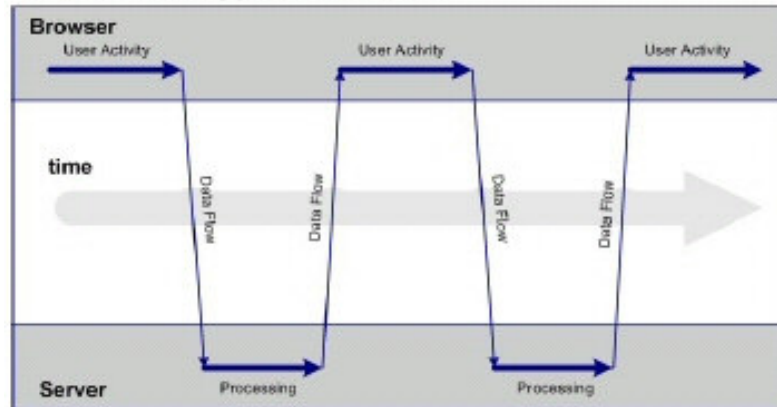
```
user.verified # => false
```

```
user.verified = true
```

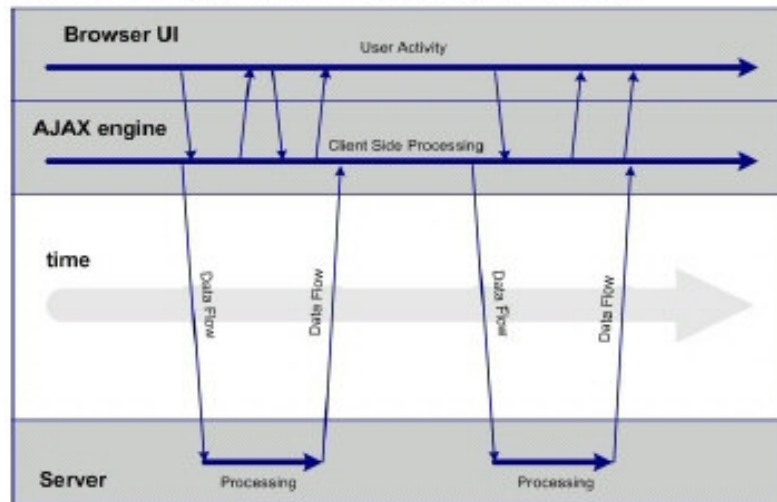
```
user.verified # => true
```

# Ajax: Asynchronous Javascript And XML

Classic WEB application Model



AJAX WEB application Model (asynchronous)



Von: Stefano Di Paola,  
Giorgio Fedon

Rails-Konferenz 2007

# Ajax

- Angriffsarten weithin die selben: Überprüfung der Eingabe- und Ausgabeparameter, Berechtigungsprüfung bei jedem Request
- Eingabeparameter auch im Hinblick auf das Ausgabeformat überprüfen: HTML, XML, JSON
- To\_json XSS Sicherheitslücke bis Rails 1.2.3: JSON-Notation kann eingeschleust werden

# Ajax

- Bisher Maßnahmen gegen XSS in Rails View:  
`<%= h @project.title %>`
- Bei (Ajax-) Funktionen, die nur Text zurückliefern und kein View rendern, z.B. `in_place_editor()`, muss vorher gefiltert werden  
`name = parseparam(params[:name], "empty", "htmlescape")`

# Fazit

- Sicherheit hängt von allen 3 Schichten ab
- Vom Benutzer kontrollierte Parameter sind **IMMER** schädlich und manipuliert, bis das Gegenteil bewiesen ist
- Sicherheit ist kaum Mehrarbeit
- Thema angeschnitten, weitere Angriffsvektoren und deren Gegenmaßnahmen unter [www.rorsecurity.info](http://www.rorsecurity.info), bald in meinem Buch „Ruby On Rails Security“, laden Sie mich ein