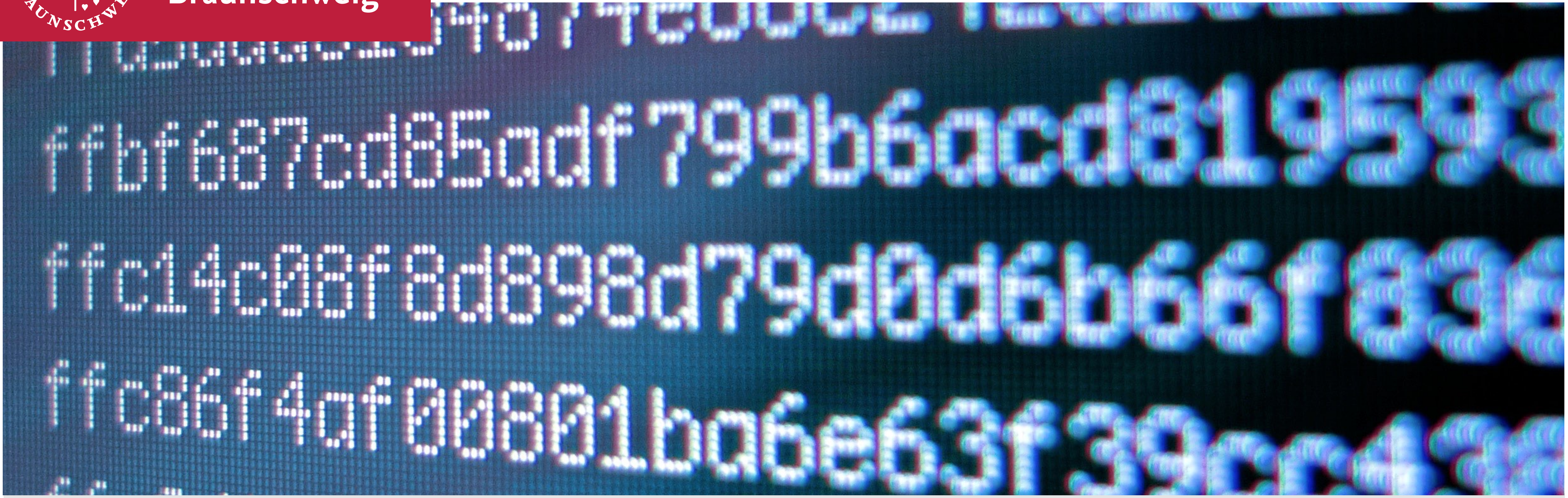
Technische Universität Braunschweig

INSTITUTE FOR APPLICATION SECURITY



Web Security

Vorlesung “Einführung in die IT-Sicherheit”

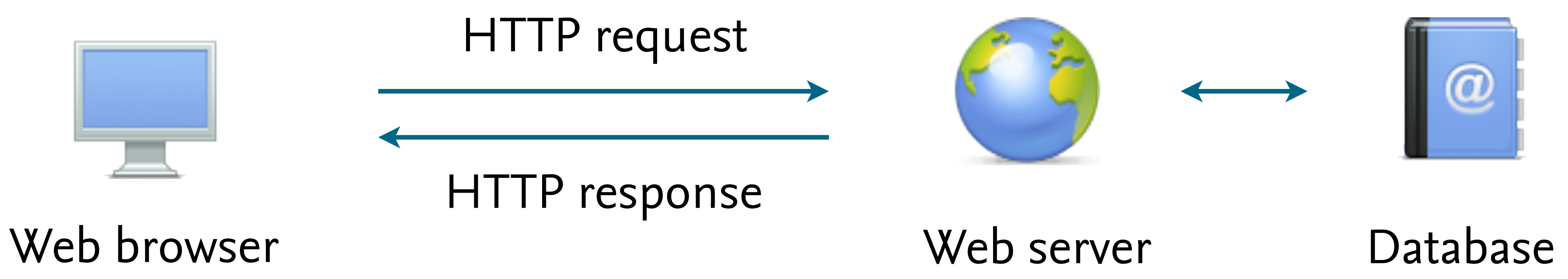
Prof. Dr. Martin Johns

Overview

- **Topic of the unit**
  - Web Security
- **Parts of the unit**
  - Part #1: Server-side attacks
  - Part #2: Web sessions
  - Part #3: Client-side attacks

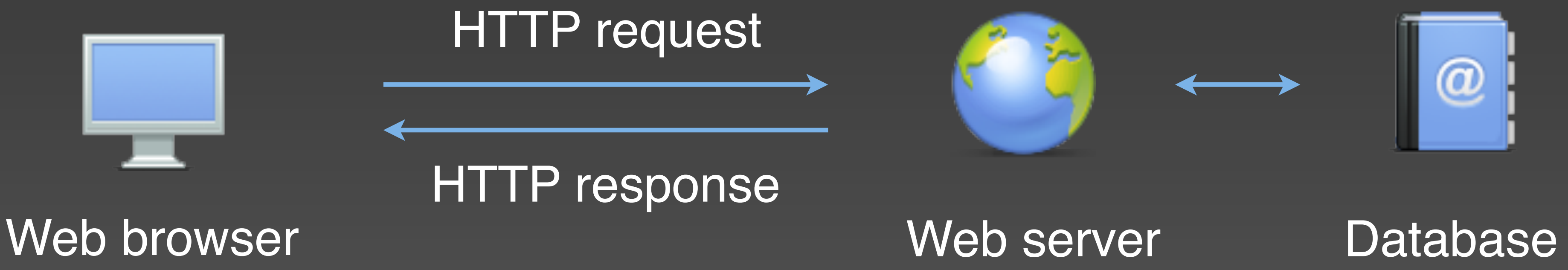
Web Applications

- **Applications implemented using "Web technologies"**
  - Client/server model using HTTP for communication
  - Code at client: HTML, JavaScript, WASM, CSS, ...
  - Code at server: PHP, JSP, ASP, ... and SQL
- **Some examples**
  - Facebook, Ebay, Youtube, ... it's just great!



Security Problems

- **All typical security problems**
  - Weak cryptography; weak authentication
  - Faulty and incorrect implementations
- **Problems specific to web applications:**
  - Client: Cross-site scripting and related attacks
  - Server: Code injection and path traversal

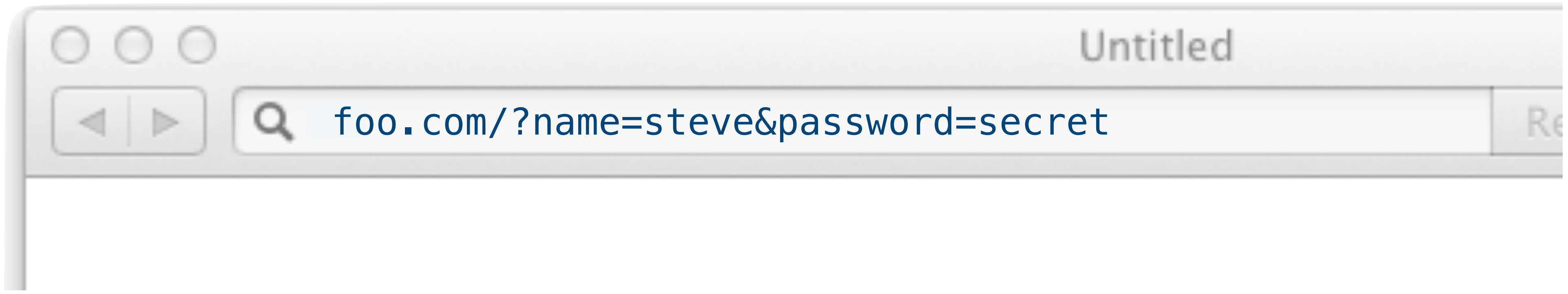


SQL Injection

- A simple example: **password check in PHP**

```
$name = $_GET ["name"];
$password = $_GET ["password"];
$query = "SELECT * FROM users WHERE name = '$name'
                        AND password = '$password'";
```

- **Let’s send a HTTP request ...**



foo.com/?name=steve&password=secret

SQL Injection

- A simple example: **password check in PHP**

```
$name = $_GET ["name"];
$password = $_GET ["password"];
$query = "SELECT * FROM users WHERE name = '$name'
                        AND password = '$password'";
```

- **Let's send a HTTP request ...**

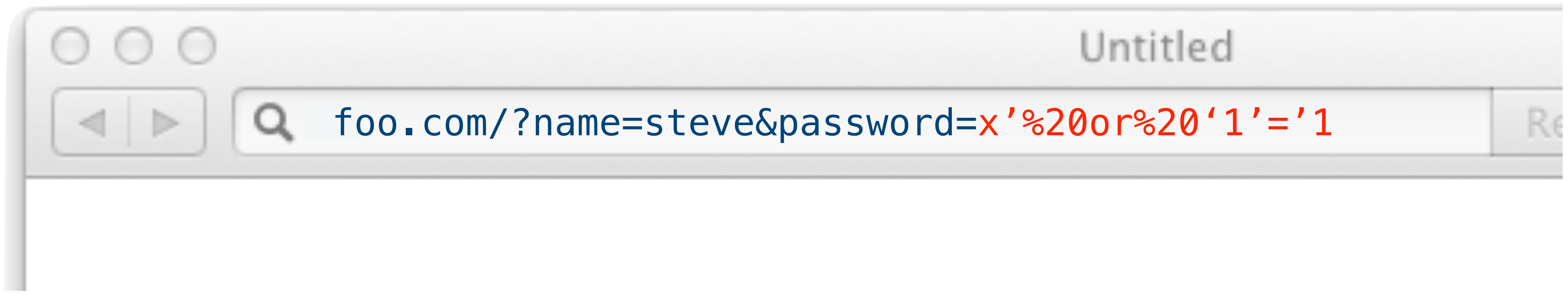
```
$name = "steve";
$password = "secret";
$query = "SELECT * FROM users WHERE name = 'steve'
                        AND password = 'secret'";
```

SQL Injection

- A simple example: **password check in PHP**

```
$name = $_GET ["name"];
$password = $_GET ["password"];
$query = "SELECT * FROM users WHERE name = '$name'
                        AND password = '$password'";
```

- **Let's send another HTTP request ...**



```
foo.com/?name=steve&password=x'%20or%20'1'='1
```

SQL Injection

- A simple example: **password check in PHP**

```
$name = $_GET ["name"];
$password = $_GET ["password"];
$query = "SELECT * FROM users WHERE name = '$name'
                        AND password = '$password'";
```

- **Let's send another HTTP request ...**

Condition is always true (password irrelevant)

```
$name = "steve";
$password = "x' or '1'='1";
$query = "SELECT * FROM users WHERE name = 'steve'
                 AND password = 'x' or '1'='1'";
```

Impact & Defense

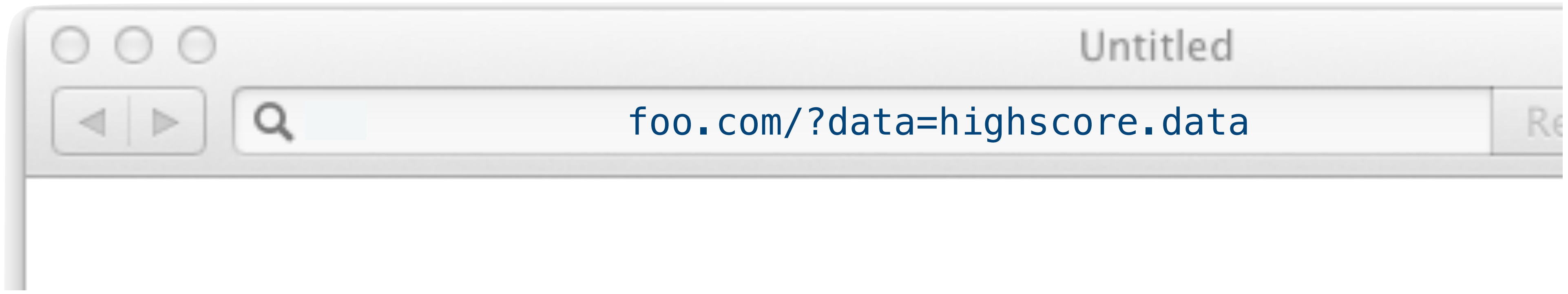
- **Attack impact**
  - Information leakage (`‘; SELECT * FROM` ...)
  - Data manipulation (`‘; INSERT` ... and `‘; DELETE` ...)
  - Code execution (depends on SQL interface of database)
- What is the problem? Insufficient validation of input data
- **Countermeasures**
  - Escaping of control and syntax characters
  - Prepared SQL statements (no mixture of data and code)

Remote Code Execution

- Another example: **local helper program called via PHP**

```
<?php
   $data = $_GET ["data"];
   $output = shell_exec("cat "+ $data);
   echo "<pre>$output</pre>";
?>
```

- **Let's send a HTTP request ...**



foo.com/?data=highscore.data

Remote Code Execution

- Another example: **local helper program called via PHP**

```
<?php
   $data = $_GET ["data"];
   $output = shell_exec("cat "+ $data);
   echo "<pre>$output</pre>";
?>
```

- **Let’s send a HTTP request …**

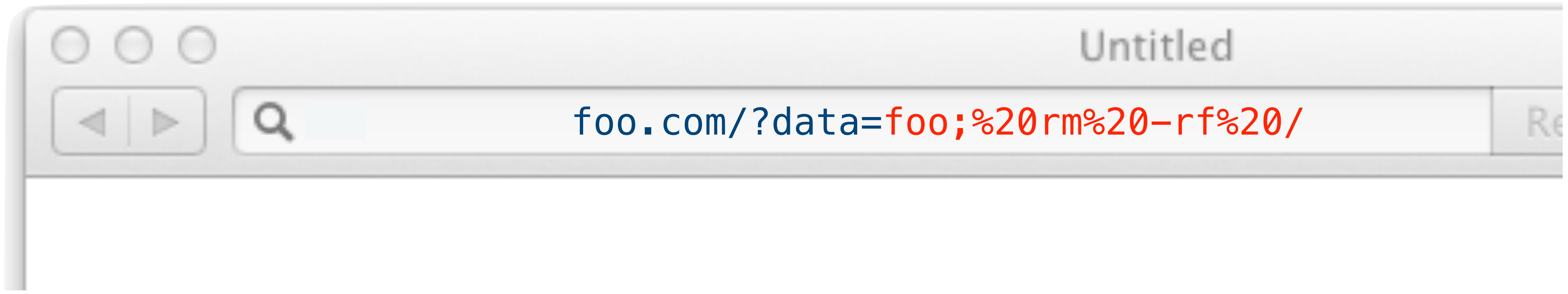
```
<?php
   $file = "highscores.data";
   $output = shell_exec("cat highscores.data ");
   echo “<pre>Peter: 13 points</pre>";
?>
```

Remote Code Execution

- Another example: **local helper program called via PHP**

```
<?php
   $data = $_GET ["data"];
   $output = shell_exec("cat "+ $data);
   echo "<pre>$output</pre>";
?>
```

- **Let's send another HTTP request ...**



foo.com/?data=foo;%20rm%20-rf%20/

Remote Code Execution

- Another example: **local helper program called via PHP**

```
<?php
   $data = $_GET ["data"];
   $output = shell_exec("cat "+ $data);
   echo "<pre>$output</pre>";
?>
```

- **Let’s send another HTTP request ...**

```
<?php
   $data = "foo; rm -rf /";
   $output = shell_exec("cat foo; rm -rf /");
   echo "<pre>file not found</pre>";
?>
```

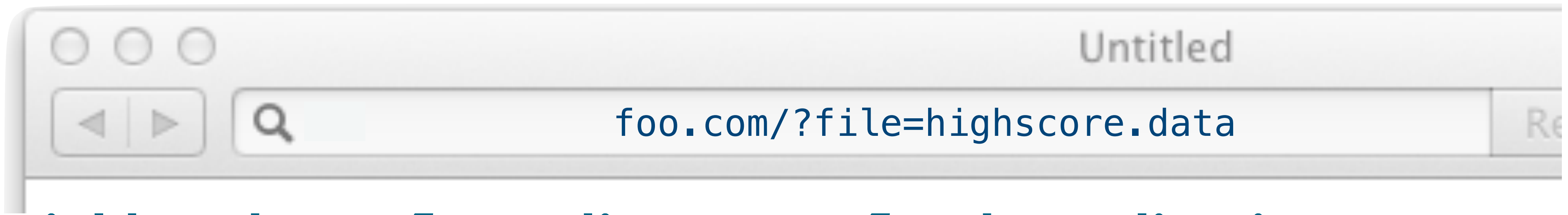
Execution of arbitrary shell commands

Impact & Defense

- **Attack impact**
  - Execution of arbitrary code on server system
  - Privileges of user running web server (e.g. www)
  - Attack variants using injected PHP or ASP code
- What is the problem? Insufficient validation of data again!
- **Countermeasures**
  - Escaping of control and syntax characters
  - No execution of shell commands from web applications

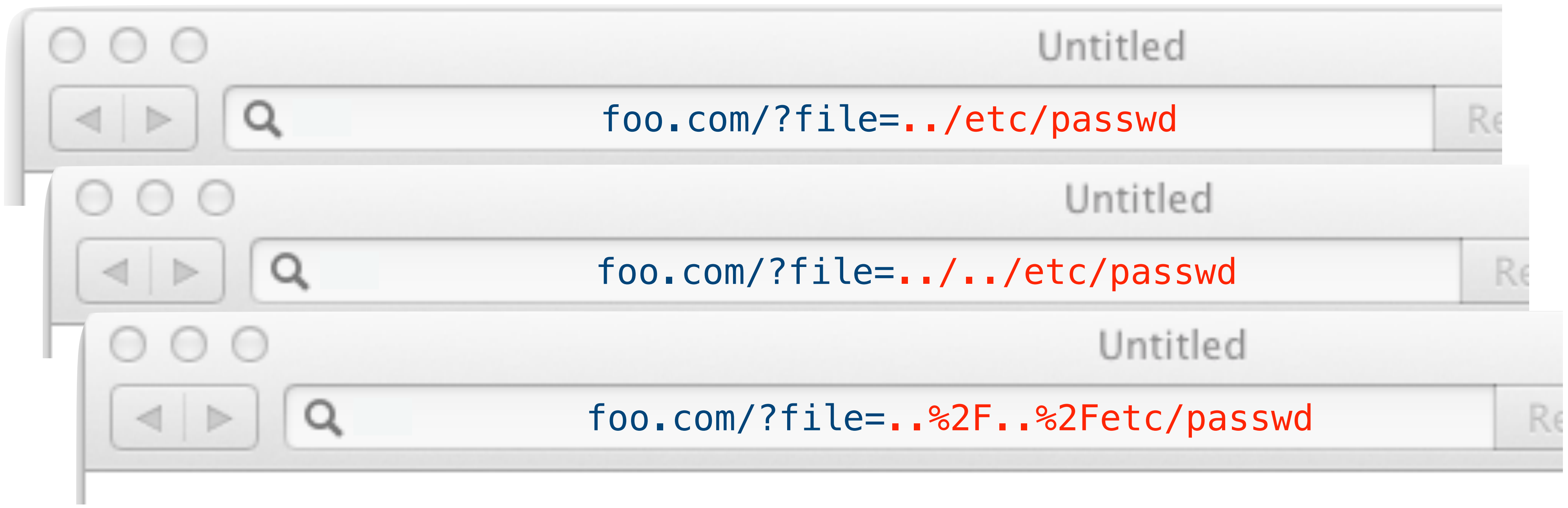
Path Traversals

- Example: **Access to local file of web application**



foo.com/?file=highscore.data

- **Potential break out from directory of web application**



foo.com/?file=../etc/passwd

foo.com/?file=../../etc/passwd

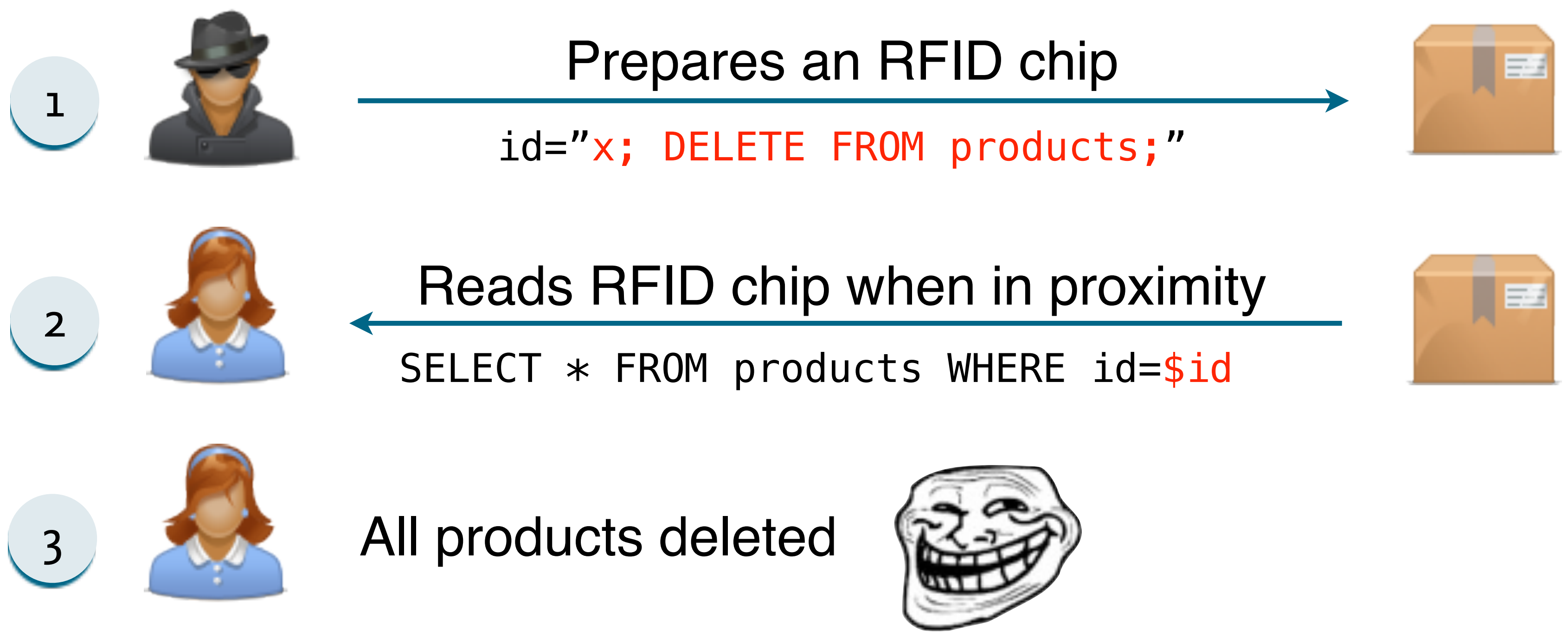
foo.com/?file=..%2F..%2Fetc/passwd

Impact & Defense

- **Attack impact**
  - Access to files outside the scope of the web application
  - Potential access to configuration files and source code
  - Combination with remote code execution (e.g. Nimda)
- What is the problem? Insufficient validation of data again!
- **Countermeasures**
  - Sanitization and normalization of file paths
  - Whitelisting of allowed files and directories

Weird Code Injection

- **Injection of code not limited to traditional user input**
  - Any type of input data potential source for attacks
- Example: **SQL injection over RFID chips**



```
id="x; DELETE FROM products;"
```

```
SELECT * FROM products WHERE id=$id
```

(Rieback et al., PERCOM 2006)

What is wrong?

- **Root cause of most vulnerabilities**
  - Dynamic construction of code using string operations
  - Separation of code and data not enforced
- **Dangerous code elements in strings**
  - SQL injection: ‘ (termination of strings)
  - Remote code execution ; (termination of commands)
  - Path traversal ../ (move to upper directory)
- Several other attack surfaces: **JSON, LDAP, XPATH,… injection**

Code vs. Data

- The developer's view

```
$sql = “SELECT ∗ FROM users WHERE passwd = ‘“ + $pass + ”’”;
```

Code Data

- The database's view

```
$sql = “SELECT ∗ FROM users WHERE passwd = ‘“ + $pass + ”’”;
```

Code Data ?

- An attack mixing code and data

```
$sql = “SELECT ∗ FROM users WHERE passwd = ‘x‘ or ‘1’ = ’1’”;
```

Code Data Attack

Overview

- **Topic of the unit**
  - Web Security
- **Parts of the unit**
  - Part #1: Server-side attacks
  - Part #2: Web sessions
  - Part #3: Client-side attacks

Sessions in HTTP

- **HTTP stateless by design**
  - Sessions prerequisite for online shopping, gaming, ...
  - Web applications need to implement session tracking
  - Session IDs track user activity across HTTP requests
- **Three common implementations**
  1. URL rewriting
  2. Form-based session IDs
  3. Cookies

URL Rewriting

- **Session management using URL rewriting**
  - Session IDs automatically appended to URLs
  - Developer needs to fix all URLs in application
  - Problem: Leakage of ID, e.g. in HTTP referrer header

```
GET /shop.php?SESSIONID=0d3adc0d3b3 HTTP/1.1
Host: www.foobar.com
Referrer: http://www.foobar.com
User-Agent: Firefox 3.1337
```

Session ID in URL

```
GET /untrusted/webpage HTTP/1.1
Host: www.dangerous.com
Referrer: http://www.foobar.com/shop.php?SESSIONID=0d3adc0d3b3
User-Agent: Firefox 3.1337
```

Leakage in referrer

Form-based Sessions IDs

- **Session management using HTML forms**
  - Session IDs stored in hidden form fields
  - Navigation of web application required to use forms
  - Transport of session IDs in body of POST request
  - Problem: back button of web browser

```
POST /shop.php HTTP/1.1
Host: www.foobar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 43

USER=john.doe&SESSIONID=0d3adc0d3b3
```

Session ID in body

Cookies

- **Session management using cookies**
  - Persistent storage maintained by web browser
  - Data set using HTTP headers or scripting, e.g. JavaScript
  - Cookies automatically added for originating (sub-)domain

```
GET /shop/login.php HTTP/1.1
Host: www.foobar.com
```

Request

```
HTTP/1.1 200 OK
Set-Cookie: 0d3adc0d3b3; path=/shop
```

Response

```
GET /shop/order.php HTTP/1.1
Host: www.foobar.com
Cookie: 0d3adc0d3b3
```

Request with cookie

Sessions and Security

- **Session IDs used for tracking authenticated users**
  - Login process marks sessions as authenticated
  - Problem: Risk of a comprise if session ID leaks
- **Securing session IDs**
  - Hard to guess IDs → Long and random strings
  - Hard to eavesdrop IDs → Encrypted transport if possible
- **Are we safe now ...**
  - ... client-side attacks may still expose session IDs

Overview

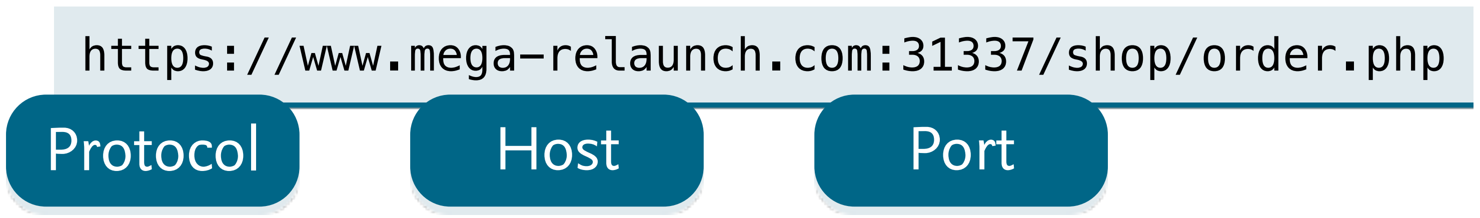
- **Topic of the unit**
  - Web Security
- **Parts of the unit**
  - Part #1: Server-side attacks
  - Part #2: Web sessions
  - Part #3: Client-side attacks

JavaScript

- **JavaScript scripting language**
  - Introduced by Netscape in 1995 as ECMAScript
  - Implemented in every modern web browser
- **Powerful scripting interface**
  - Alteration of web pages
  - Access to URLs and form fields
  - Access and modification of cookies
  - Initiating of new HTTP requests
  - Execution of dynamic code

Same-Origin Policy

- **Main security measure for JavaScript**
  - Access restricted to elements from “same origin”
  - Also applied to ActionScript (Flash) and CSS stuff
- **Two elements share the same origin if**
  - … the protocol is identical
  - … the host is identical
  - … the port is identical



```
https://www.mega-relaunch.com:31337/shop/order.php
```

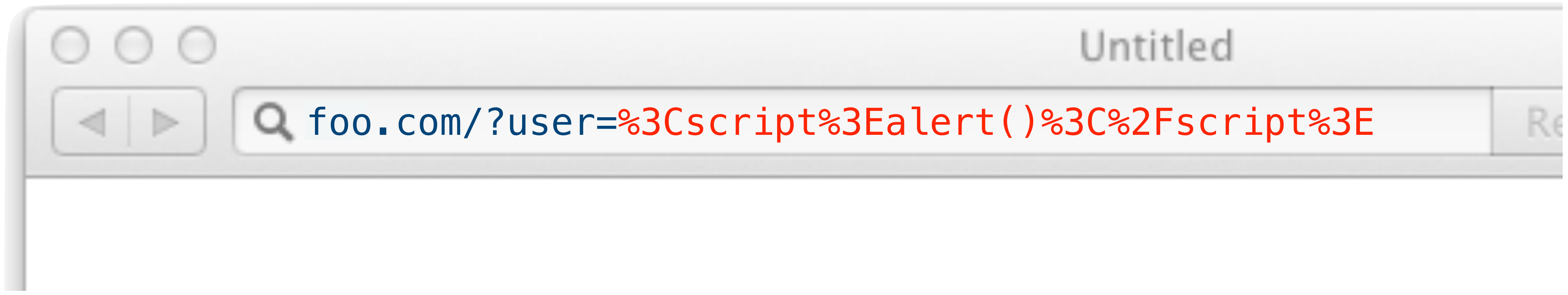
Cross-site Scripting

- **Cross-site Scripting (XSS)**
  - Injection of (JavaScript) code into webpages
  - Bypass of same-origin policy due to injection
  - Root-cause: display of insufficiently validated data
- **Simple example**

```
<?php
   $user = $_GET ["user"];
   echo "Welcome home, $user!";
?>
```

Cross-site Scripting

- **Cross-site Scripting (XSS)**
  - Injection of (JavaScript) code into webpages
  - Bypass of same-origin policy due to injection
  - Root-cause: display of insufficiently validated data
- **Simple example**



```
foo.com/?user=%3Cscript%3Ealert()%3C%2Fscript%3E
```

Cross-site Scripting

- **Cross-site Scripting (XSS)**
  - Injection of (JavaScript) code into webpages
  - Bypass of same-origin policy due to injection
  - Root-cause: display of insufficiently validated data
- **Simple example**

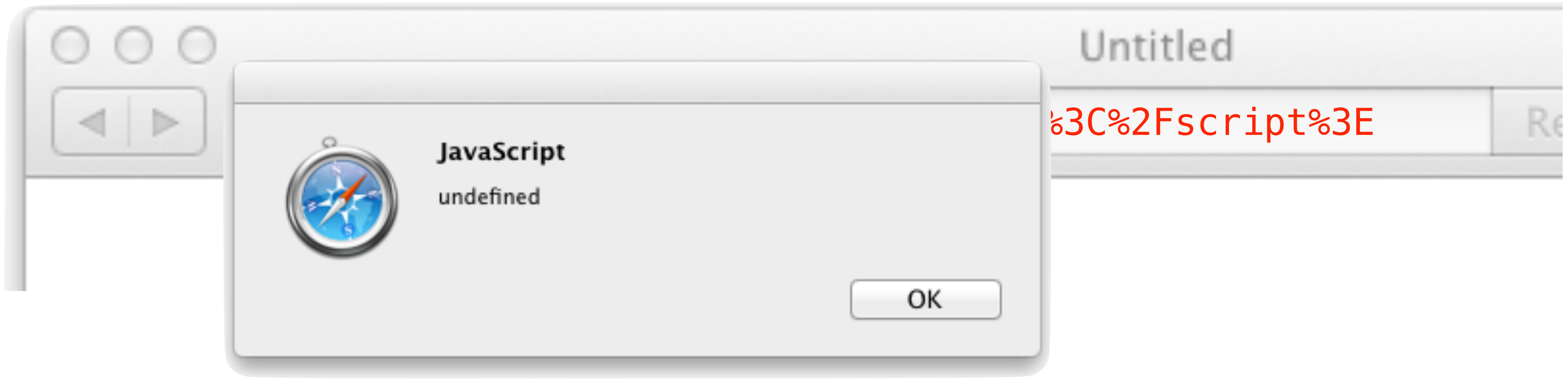
```
<?php
   $user = $_GET ["user"];
   echo "Welcome home, <script>alert()</script>!";
?>
```

Technische Universität Braunschweig

IAS INSTITUTE FOR APPLICATION SECURITY

Cross-site Scripting

- **Cross-site Scripting (XSS)**
  - Injection of (JavaScript) code into webpages
  - Bypass of same-origin policy due to injection
  - Root-cause: display of insufficiently validated data
- **Simple example**



```
%3C%2Fscript%3E
```

JavaScript

undefined

Capabilities of XSS

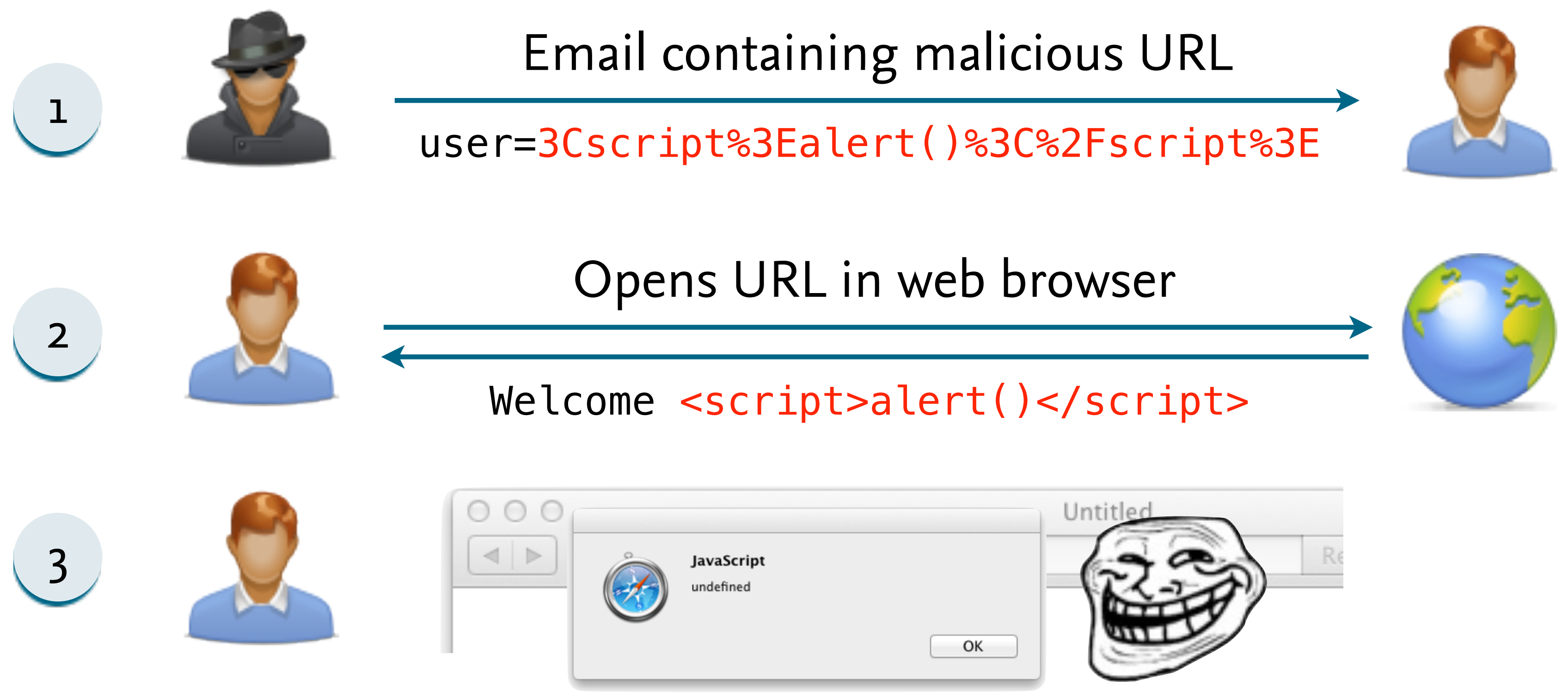
- Injection of JavaScript code = compromise of web browser
- Forgery of web content
    - Attacker manipulates content on the web page
- Spoofing of login dialogues
    - Attacker uses JavaScript code to phish login credentials
- Session and browser hijacking
    - Attacker steals session IDs and acts as the user
- Further web-based attacks, e.g. JavaScript worms

Entry Points and Types

- **XSS entry points**
  - Injection of tags, e.g. <script>...
  - Breaking out of attributes, e.g. <img alt="...
  - JavaScript URLs on some browsers, javascript:...
  - Media files containing JavaScript, e.g. SVG
- **Common types of XSS**
  - Reflected cross-site scripting (non-persistent)
  - Stored cross-site scripting (persistent)
  - DOM-based cross-site scripting

Reflected XSS

- **Reflected cross-site scripting**
  - Attacker sends malicious JavaScript code directly to user
  - Malicious JavaScript code is reflected by web server

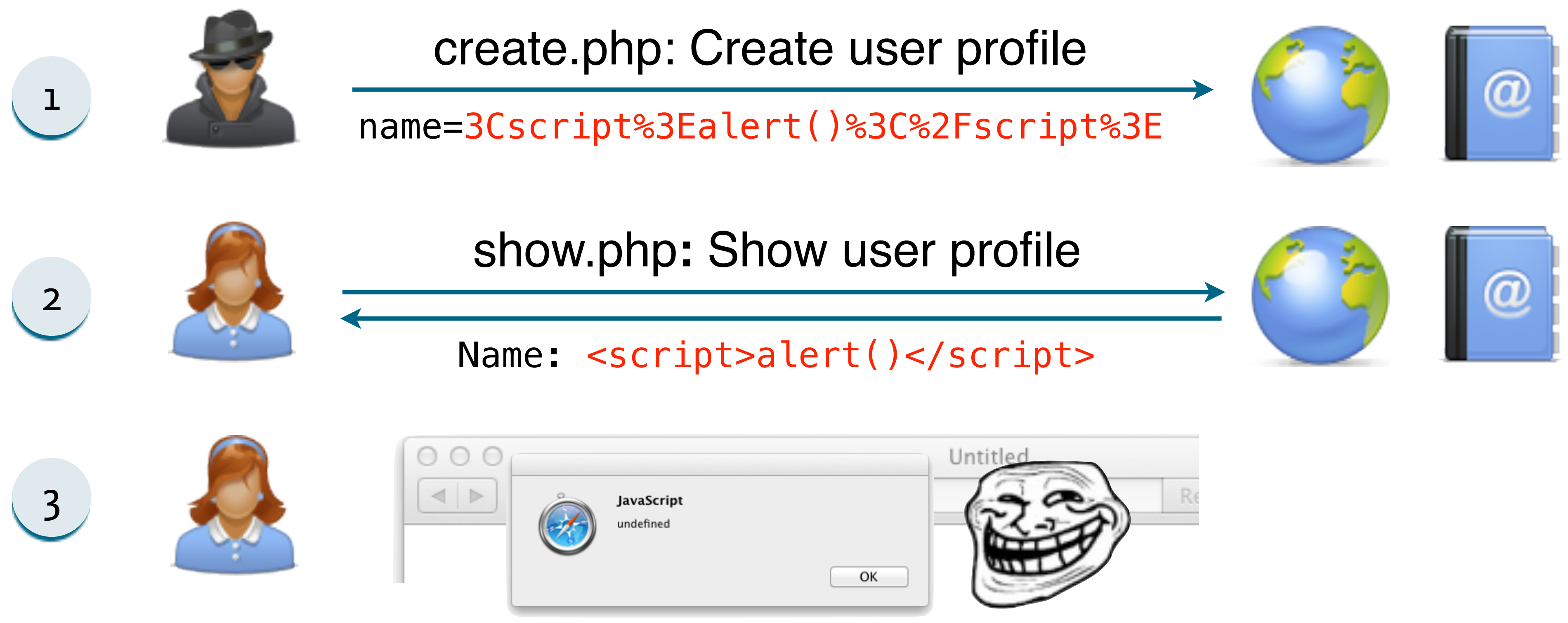


```
user=3Cscript%3Ealert()%3C%2Fscript%3E
```

```
Welcome <script>alert()</script>
```

Stored XSS

- **Stored cross-site scripting**
  - Malicious JavaScript code is injected at the web server
  - User queries trigger delivery of malicious code

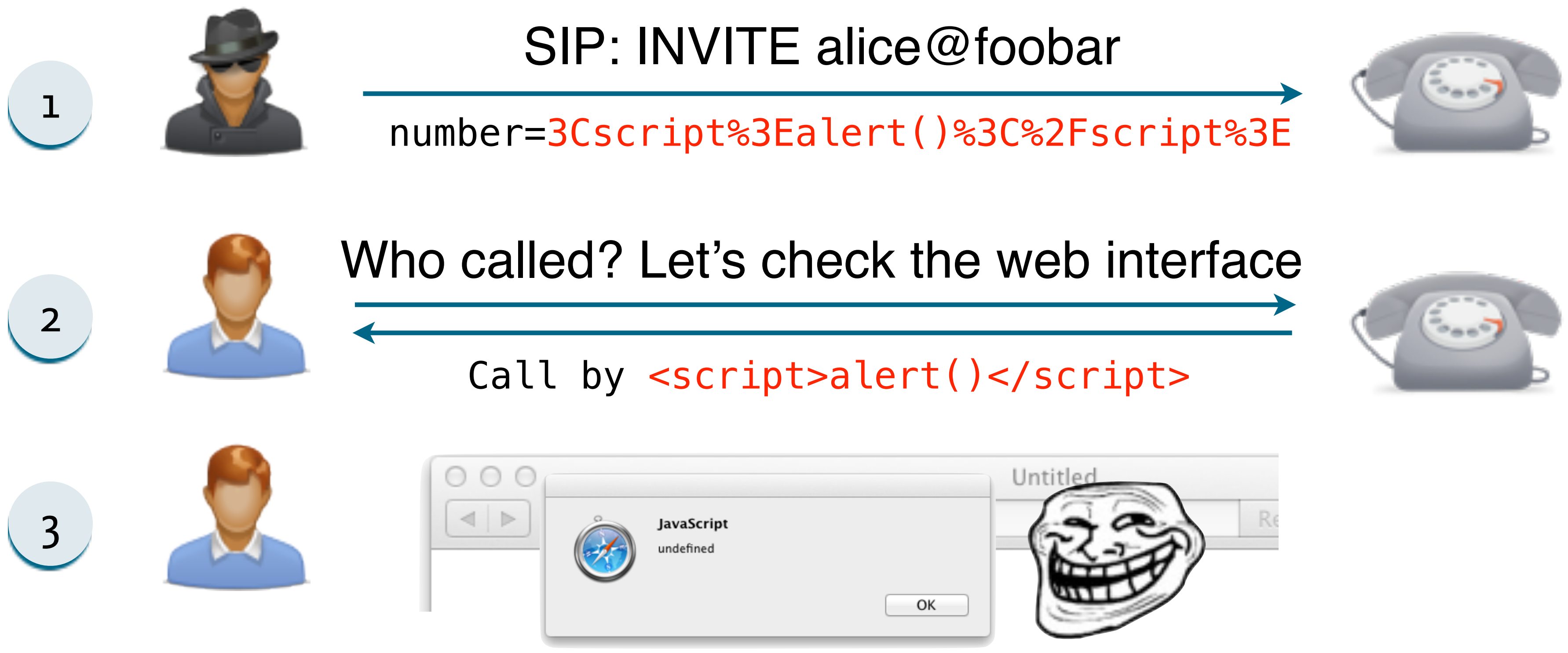


```
name=3Cscript%3Ealert()%3C%2Fscript%3E
```

```
Name: <script>alert()</script>
```

Weird XSS

- **Injection of code not limited to traditional user data**
  - Any type of input data potential source for attacks
- Examples: **XSS over SIP (Internet telephony)**



```
number=3Cscript%3Ealert()%3C%2Fscript%3E
```

```
Call by <script>alert()</script>
```

Again: Code vs. Data

- The developer's view

```
<?php echo “<b>welcome $user</b> posted <em>$comment</em>” ?>
```

HTML

Data

- The web browser's view

```
<?php echo “<b>welcome $user</b> posted <em>$comment</em>” ?>
```

Tags

?

Data

?

- Cross-site scripting

```
<?php echo “<b>welcome <script>alert()</script></b> ...” ?>
```

Avoiding XSS

- **Validation of any user-supplied data**
  - Escaping of GET and POST parameters
  - Validation of externally stored data, e.g. databases
  - Checking of other sources, e.g. cookies, referer
- **Comprehensive input validation non-trivial**
  - Output validation often easier
  - Developer knows where JavaScript code is located
- In general: **whitelisting favorable over blacklisting**

Summary

...many other attacks

- **Cross-site Request Forgery (CSRF)**
  - Indirect access to a web application via forged requests
  - Example: <img src=”http://foo/?delete-user”>
- **HTTP Parameter Pollution (HPP)**
  - Exploitation of inconsistent HTTP parameter parsing
  - Example: a=x&a=y may be interpreted as a=x or a=y or a=xy
- **SSL Striping**
  - Man-in-the-middle attack removing SSL encryption
  - Example: web proxy replacing https:// with http://

Conclusions

- **Web Security**
  - Developing secure web applications non-trivial
  - Several attack vectors using injected code
  - Never, never, never trust user-supplied data
- **Client-side and server-side attacks**
  - Server-side: SQL and other code injection
  - Client-side: Cross-site scripting and friends