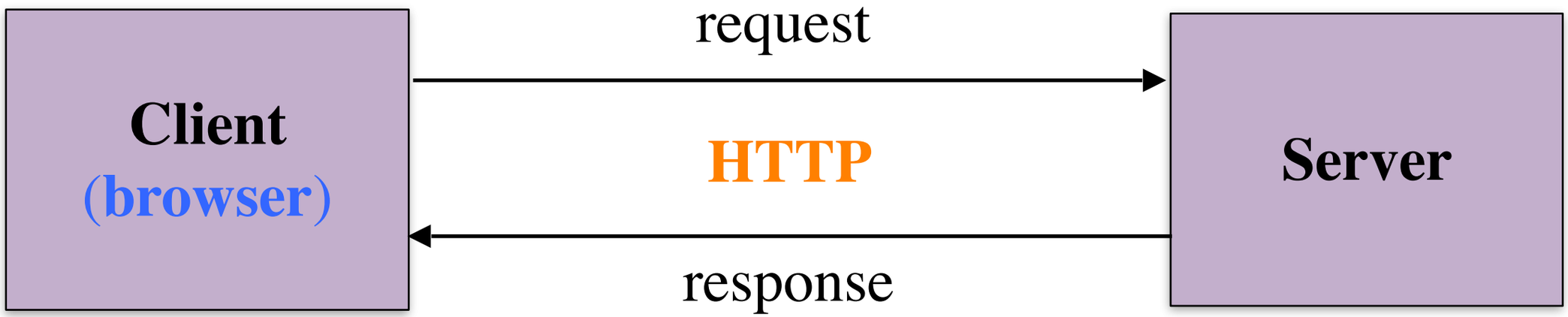
Vulnerabilities in web applications

Web = Client + Server



HTTP request contains the URL of the resource and the header
HTTP response contains a status code, the header, data

Client-server interaction

- Web pages (resources) are identified by **URL**

`http://securityfans.com/forum/viewtopic.php?t=4841378`

protocol **hostname** **path to a resource** **arguments**

Client request:

GET / HTTP/1.1

Host: satoss.uni.lu

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

Accept-Encoding: gzip,deflate,sdch

Accept-Language: en-US,en;q=0.8,it;q=0.6,ru;q=0.4

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122 Safari/537.36

Client-server interaction

- Web pages (resources) are identified by **URL**

`http://securityfans.com/forum/viewtopic.php?t=4841378`

protocol **hostname** **path to a resource** **arguments**

Server response:

HTTP/1.1 200 OK

Connection: keep-alive

Content-Encoding: gzip

Content-Length: 3161

Content-Type: text/html; charset=utf-8

Date: Tue, 25 Nov 2014 13:56:31 GMT

Server: ATS/3.2.4

Vary: Accept-Encoding

X-Powered-By: PHP/5.4.34-0+deb7u1

Vulnerabilities in web applications

- **Many security holes in corporate IT are due to vulnerabilities in the code of web applications**
    - These are often used as the “weakest link principle” by the attackers
- **Differences between web apps and client-server apps open enterprises to significant risks**
    - JavaScript has diffused boundaries between client and server
    - Web apps are easier to deploy but harder to maintain securely
- **Equifax credit bureau had leaked critical personal data of 143 million customers because of a patchable vulnerability in a third-party component of one of their web app (CVE-2017-5638)**
    - The attack was launched at the **end of July**, but **the patch for CVE-2018-5638 was already available in March**!

Practical approaches in vulnerability discovery

- **Software security is a problem that is very hard to define**
  - Bell-LaPadula model: *“a system is secure iff it starts in a secure state and cannot enter an insecure state”*
  - *“I don’t want my emails to be read by others”* - this is easy to express, but quite difficult to formalize
  - It is nearly impossible to analyze software behavior conclusively
    - Turing’s halting problem, Rice’s theorem
- **The complexity of software systems continue to grow -> more and more vulnerabilities are introduced**
  - Sometimes, we must fall back to a set of empirical recipes

Practical approaches in vulnerability discovery (continued)

- **Plan your actions as if everything is already compromised**

- **Rely on tools that detect SPECIFIC problems, but do not rely on tools completely**

    - Tools can help in finding certain vulnerabilities, but they are nothing without human knowledge (same problem as with signature-based intrusioin detection).

- **Learn from (preferably) others' mistakes**

    - There are many vulnerability taxonomies, databases, case studies. Don't forget about Open Source Sotfware

A quick look at vulnerability taxonomies

- **All vulnerabilities are related to flaws in source code**
    - Design and implementation errors
    - Many of them are language/framework independent
- **Categories, classifications, and databases**
    - Open Web Application Security Project (OWASP)
    - Common Weakness Enumeration (CWE)
    - The National Vulnerability Database (NVD)
    - ~~Open-sourced Vulnerability Database (OSVDB)~~
    - IARPA Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP

OWASP Top 10 (2013)

A1: Injection

A2: Broken Auth. and Session Management

A3: Cross-site Scripting (XSS)

A4: Insecure Direct Object References

A5: Security Misconfiguration

A6: Sensitive Data Exposure

A7: Missing Function Level Access Control

A8:Cross-site Request Forgery (CSRF)

A9: Using Component With Known Vulns.

A10: Unvalidated Redirects and Forwards

Injection vulnerabilities

- **Assume, an app is written in multiple languages: Java, JavaScript, HTML, SQL, …**

- **An app accepts user inputs and does not check them**

- **Problem: some inputs that look like String in Java, might be valid instructions in SQL, JavaScript, …**

- **Consequences?**

  - From website defacement …

  - … to complete control over a vulnerable server

SQL/NoSQL injection

- **Due to insufficient input filtering (or output escaping), attacker-controlled input may be interpreted as code by a database interpreter and executed**

- **Related threats:** Information Disclosure, Data Modification/Deletion, Elevation of Privileges

- **Technical impact:** Moderate/Severe

SQL injection: example

```
UserData data = getDataFromUser();
String userId = data.getUserId();
String passwd = data.getPasswd();

SomeDB.executeQuery("SELECT * FROM users WHERE users.userId =
'"+ userId + "' AND users.passwd ='" + passwd + "'");
```

```
userid <- "John Doe"
passwd <- "qweJk@#4kw"
query <- "SELECT * FROM users WHERE users.userId =
'John Doe' AND user.passwd = 'qweJk@#4kw'"
```

```
userId <- "Batman' OR '1' == '1'; DROP TABLE users; --"
passwd <- ""

query <- "SELECT * FROM users WHERE users.userId =
'Batman' OR '1' == '1'; DROP TABLE users;
–' AND users.passwd= ''"
```

NoSQL injection: example

```
var login = request.body.userid;
var passwd = request.body.passwd;

var query = eval("({ _id: '" + login + "', pword : '" +
Passwd + "'})");
if (dbprovider.findOne(query)) authenticate(login);
```

```
Login <- "John Doe"
passwd <- "qweJk@#4kw"
query <- ({ _id : 'John doe', pword: 'qweJk@#4kw'})
```

```
login <- "Batman'})//"
passwd <- ""

query <- "({ _id : 'Batman'})//, pword : ''})"
query <- "({ _id : 'Batman'})"
```

SQL/NoSQL injection

DEMO

Finding db injection

- **Symptoms:**
  - App gets user input and does not check it
  - App uses user input to construct database queries, uses string concatenation

Language	Keywords
Java (+JDBC)	`sql, java.sql`
Python	`pymssql,`
C#	`Sql, SqlClient, OracleClient, SqlDataAdapter`
PHP	`mysql_connect`
Node.js	`require("mysql”), require(”mssql"), require("mongodb")`

Preventing db injection

- **Validate user inputs on server side before processing**

- **In JavaScript, do not use the *eval( )* function to parse user inputs, do not use String concatenation**

- **Use special library functions (a.k.a. prepared statements in Java, or *JSON.parse( )* in JavaScript) for constructing database queries with user input**

Cross-site Scripting (XSS)

- **Insufficient input validation can allow attackers to plant own HTML or scripts on a vulnerable website.**

- **The reflected variant takes the advantage of the input when it is being incorrectly “echoed” back to a browser**

- **The stored (persistent) variant takes the additional advantage of the lack of sanitization of the data that goes to a DB (and is displayed to users later)**

- **Related threats:** Information Disclosure, Elevation of Privileges

- **Technical impact:** Moderate/Severe

Cookies

- **Cookies are key-value pairs that are set up in a web browser**
- **Cookies are mostly used for site personalization and session management**
- **Cookies can be used by advertisement engines to track users**
- **Stealing valid session cookies allows to impersonate legitimate users**

XSS scenario

- **Every JavaScript program can be written as a string that gets evaluated**

- **Attacker injects a malicious script to yourbank.com**

- **When a victim access yourbank.com, her browser assumes that the script is being executed under yourbank.com and should be trusted**

- **The malicious script can access cookies (unless the "HttpOnly" flag is set) or any other sensitive information saved by the browser and used for yourbank.com**

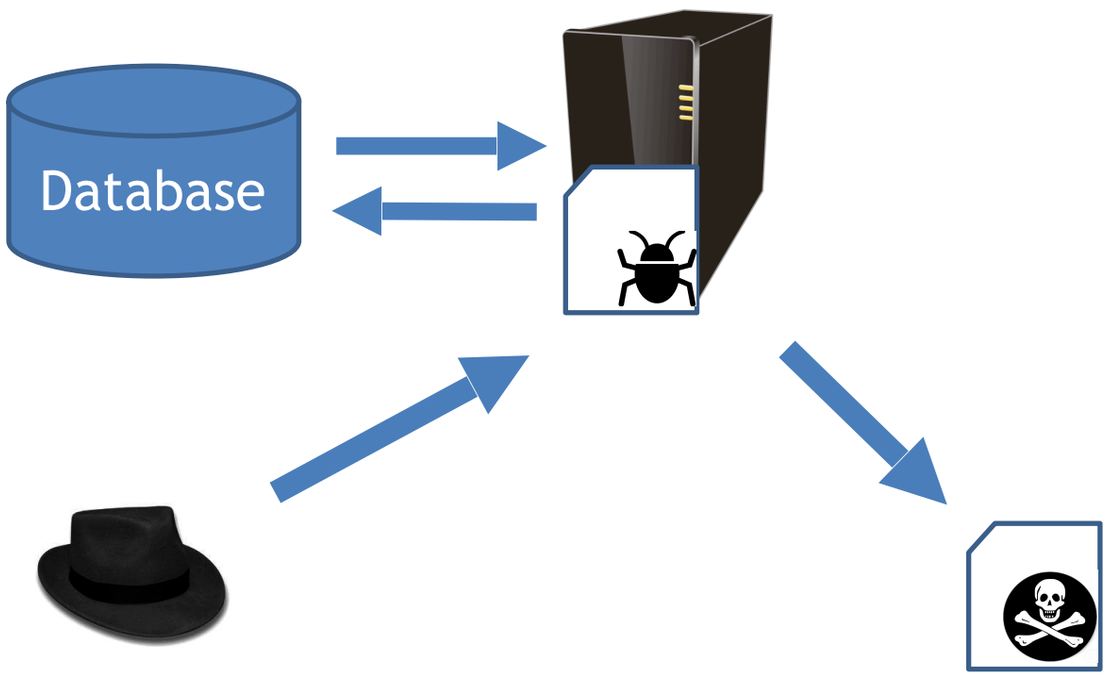
- **The script can also rewrite the contents of a HTML web page to trick users into giving up their personal data**

XSS reflected

DEMO

XSS stored

Step 0: developer writes vulnerable pages:
1st one stores invalidated input;
2nd one reads it from a database and with no validation.



Step 1:
Attacker sends malformed
input (code) to a vulnerable web page.

Step 2: User browses the site.

Step 3: Web site reads unchecked data and sends it along with attacker's code to the user's browser.

Step 4: User's browser renders the web page and runs the attacker's code
(every time the page is requested!)

XSS stored

DEMO

Finding XSS

Language	Keywords
Java (JSP)	`addCookie`, `getRequest`, `request.getParameter` followed by `<jsp:setProperty` or `<%=` or `response.sendRedirect`
Python	`form.getvalue`, `SimpleCookie` when the data is not validated correctly.
C#	`Request.*`, `Response.*`, and `<%=` when the data is not validated correctly.
PHP	Accessing `$_REQUEST`, `$_GET`, `$_POST`, or `$_SERVER` followed by `echo`, `print`, `header`, or `printf`.
Node.js	`request`, `response`, …

Preventing XSS

- **Validate user input on both client- and server-side**
- **Set the “HttpOnly” for cookies explicitly**
- **Use output encoding for correct contexts**
- **Implement Content Security Policy (CSP)**

HTML Attribute Encoding	<input type="text" name="fname" value=" UNTRUSTED DATA">
URI Encoding	clickme
JavaScript Encoding	<script>var currentValue=' UNTRUSTED DATA';</script> <script>someFunction(' UNTRUSTED DATA');</script>
CSS Encoding	Selection

```
Content-Security-Policy: default-src 'self' *.mydomain.com
```

Information disclosure

- **Attacker is able to get unprotected critical data. The data itself can be the goal, or it can be used by the attacker for reaching its goal (exploit other vulnerabilities)**

- **Intentional:** developers have a mistmatch with end users on what data should be protected (privacy issues)

- **Accidental:** critical data can be accessed throug an error in the code, or a non-obvious channel (e.g., verbose error messages).

- **Technical impact:** could be anything

Information disclosure: intentonal

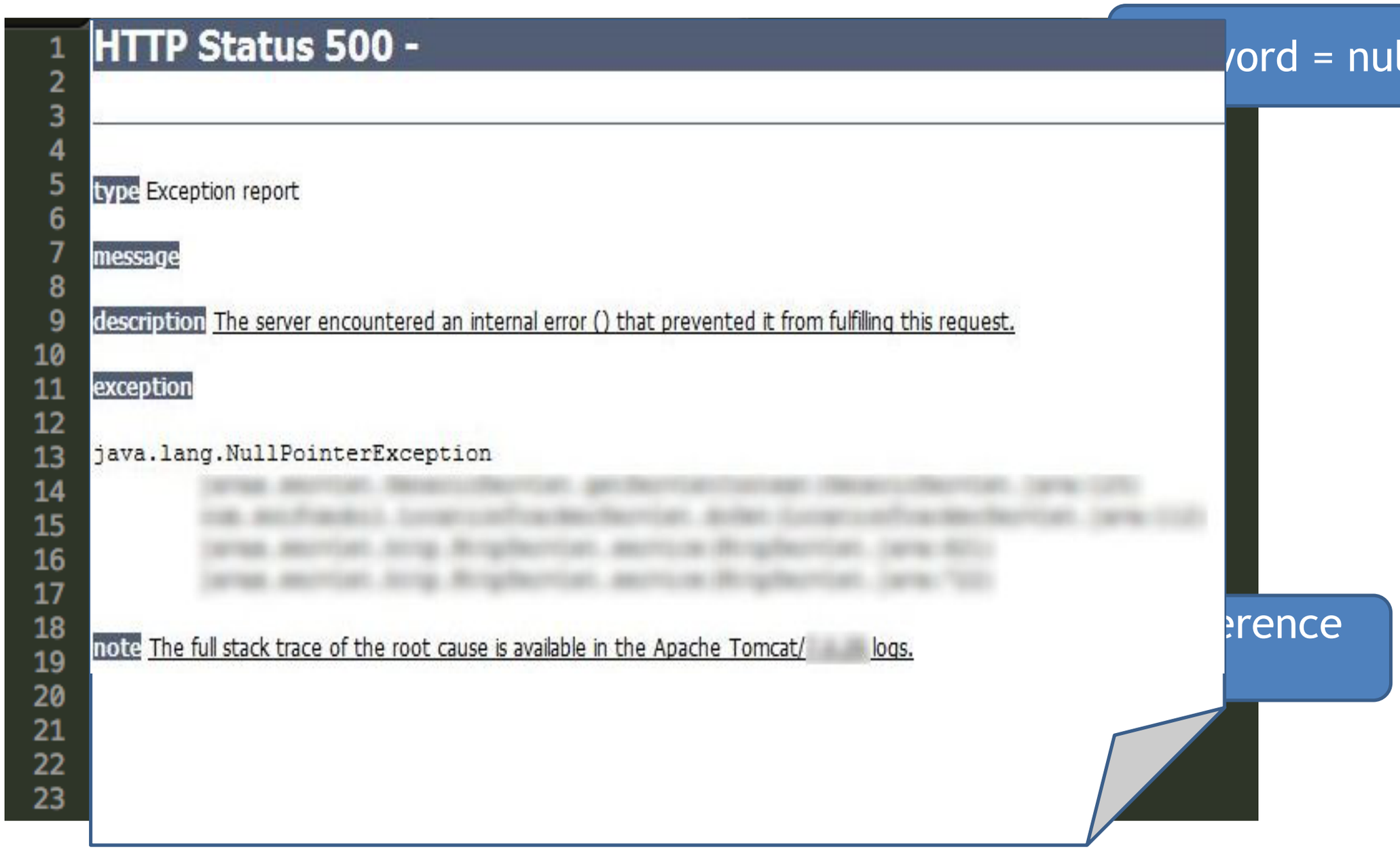
```php
<?php
  $UName = "[redacted]";
  $PWord = "[redacted]";
  $DB="[redacted]";
?>
```

```python
def authenticate(uname, pword):
    if uname == "":
        return False
    elif pword != "[redacted]":
        return False
    else:
        return True
```

```python
def authenticate(uname,pword):
    if uname=="[redacted]" and pword=="[redacted]":
        return True
    else:
        return False
```

Information disclosure: accidental 1

Information disclosure: accidental 2



Finding information disclosure

- **App returns the “default” information such as server type, configuration, ip address, hostname, etc.**

- **There are too many details in error messages (e.g., stack traces), there are unhandled exceptions; non-uniform error messages when handling user logins**

- **Look for “password”, “credentials”, “login” in the source code - you might find something interesting**

Path traversal

- **An application can be tricked into reading/writing files at arbitrary locations (despite app-level restrictions). Unconstrainted, such bugs are often used for deploying attacker-controlled scripts**

- **Related threats:** Information Disclosure, Code Injection, Denial of Service

- **Technical Impact:** Moderate/Severe

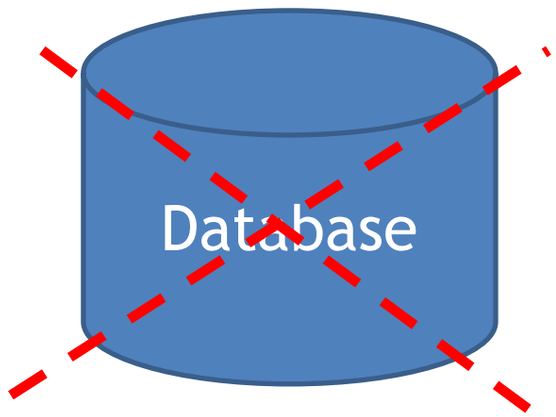
Path traversal: example

```
String path = getInputPath();
if (path.startsWith("/safe_dir/")) {
    File f = new File(path);
    f.delete();
}
```

An attacker could provide an input such as : /safe_dir/../data.db

The code attempts to validate the input by whitelisting.

If the file is within the ”/safe_dir/” folder, the file gets deleted.



Path traversal

DEMO

Finding path traversal

- **App gets an input from user that is not being checked**
- **The user input is used to constuct a path string to a file/folder (downloading/uploading files, redirects, etc.)**
- **Sanitization functions often contain errors (remember the previos example), so they have to be checked carefully**

Useful links

- *Zalewski, Michal. The tangled Web: A guide to securing modern web applications. No Starch Press, 2012.*
- *Howard, Michael, David LeBlanc, and John Viega. 24 deadly sins of software security: programming flaws and how to fix them. McGraw-Hill, Inc., 2009.*
- *OWASP: the free and open software security community https://www.owasp.org/index.php/Main_Page*
- *Secure Coding Guidelines for Java SE http://www.oracle.com/technetwork/java/seccodeguide-139067.html*