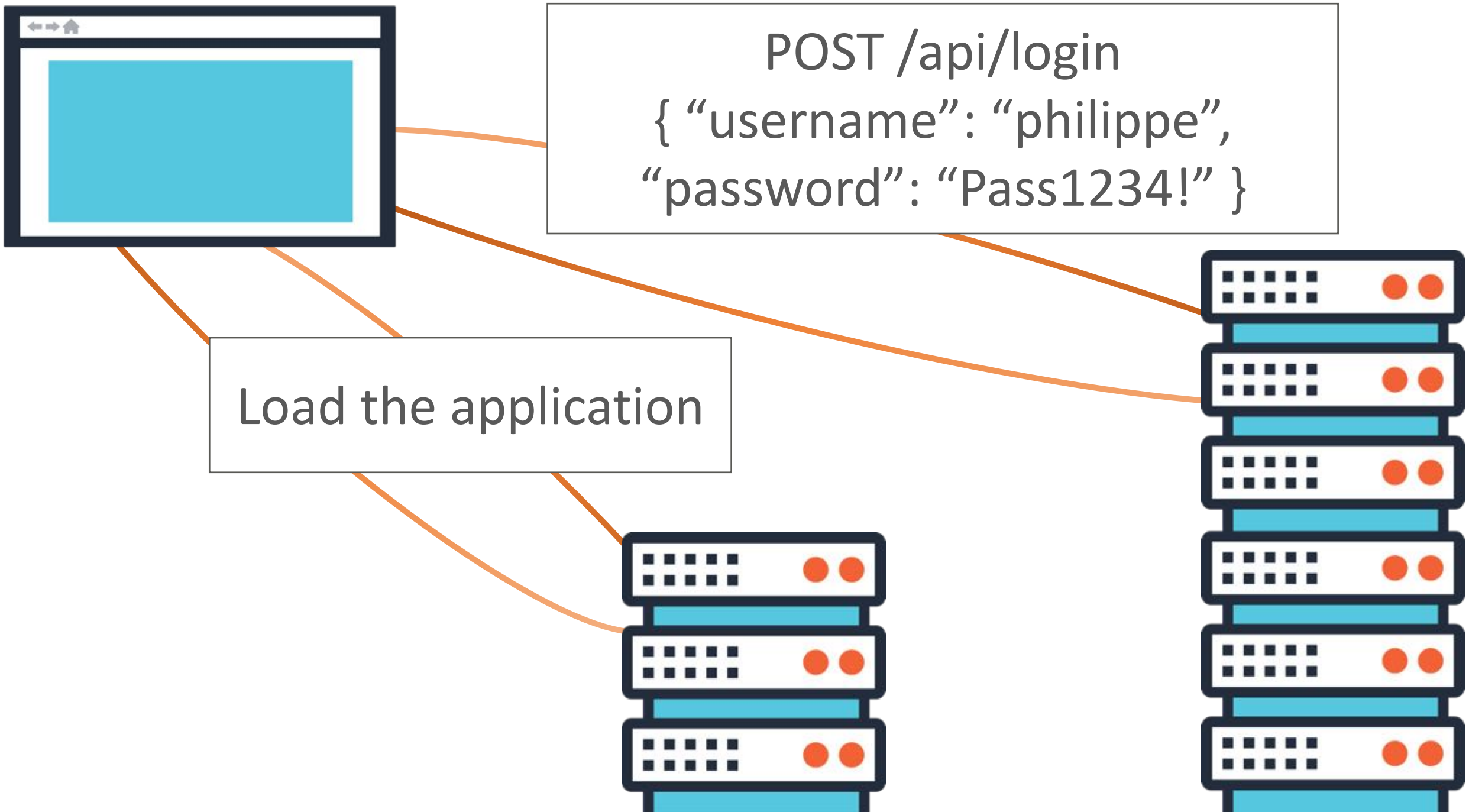
Common REST API Security Pitfalls

Philippe De Ryck

OWASP BeNeLux days 2017


POST /api/login
{ “username”: “philippe”,
“password”: “Pass1234!” }
Load the application

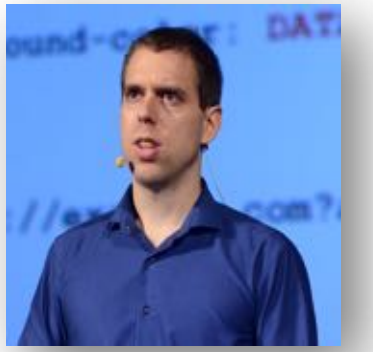
A10 Underprotected APIs

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific
Consider anyone with the ability to send requests to your APIs. Client software is easily reversed and communications are easily intercepted, so obscurity is no defense for APIs.	Attackers can reverse engineer APIs by examining client code, or simply monitoring communications. Some API vulnerabilities can be automatically discovered, others only by experts.	Modern web applications and APIs are increasingly composed of rich clients (browser, mobile, desktop) that connect to backend APIs (XML, JSON, RPC, GWT, custom). APIs (microservices, services, endpoints) can be vulnerable to the full range of attacks. Unfortunately, dynamic and sometimes even static tools don't work well on APIs, and they can be difficult to analyze manually, so these vulnerabilities are often undiscovered.		The full range of negative outcomes is possible, including data theft, corruption, and destruction; unauthorized access to the entire application; and complete host takeover.	Consider the impact of an API attack on the business. Does the API access critical data or functions? Many APIs are mission critical, so also consider the impact of denial of service attacks.

https://github.com/OWASP/Top10/blob/master/2017/drafts/OWASP%20Top%2010%20-%202017%20RC1-English.pdf

About me – Philippe De Ryck

- My goal is to help you build secure web applications
  - Courses and training programs
  - Talks at various developer conferences
  - Slides, videos and blog posts on **https://www.websec.be**



- Author of the Web Security Fundamentals course
  - Free online course on the edX platform
  - All info on ***https://mooc.websec.be***

- Course curator for the *SecAppDev course*
  - Security course targeted towards developers, architects, …
  - Week-long course taught by international experts in their domain

secappdev.org

HTTPS

Offer your API over HTTPS

- There is no valid excuse to not use HTTPS anymore
  - Let's Encrypt offers free certificates for all
  - Performance is no longer an issue
- APIs are accessed directly from within an application
  - Makes setting up HTTPS easier, as you do not need to support a redirect from HTTP
  - Simply disable HTTP for your API endpoints altogether
- Network-based attacks can still attempt a fallback to HTTP
  - Configure ***HTTP Strict Transport Security (HSTS)*** to prevent this from happening
  - HSTS will tell the browser to use HTTPS for every request, regardless of the scheme

```
Strict-Transport-Security: max-age=31536000
```

SECURITY PITFALL

Allowing access to your API over HTTP

APIs are accessed from code, so there is no need to support a redirect from HTTP to HTTPS. Lock your API further down by enabling HSTS

T-Mobile Website Allowed Hackers to Access Your Account Data With Just Your Phone Number

The bug exposed customers' email addresses, their billing account numbers, and the phone's IMSI numbers. T-Mobile has patched the bug.

https://motherboard.vice.com/en_us/article/wjx3e4/t-mobile-website-allowed-hackers-to-access-your-account-data-with-just-your-phone-number

```
exports.read_a_task = function(req, res) {
  Task.findById(req.params.taskId, function(err, task) {
    if (err)
      res.send(err);
    res.json(task);
  });
};
```

```
exports.delete_a_task = function(req, res) {
  Task.remove({
    _id: req.params.taskId
  }, function(err, task) {
    if (err)
      res.send(err);
    res.json({ message: 'Task successfully deleted' });
  });
};
```

https://www.codementor.io/olatundegaruba/nodejs-restful-apis-in-10-minutes-q0sgsfhbd

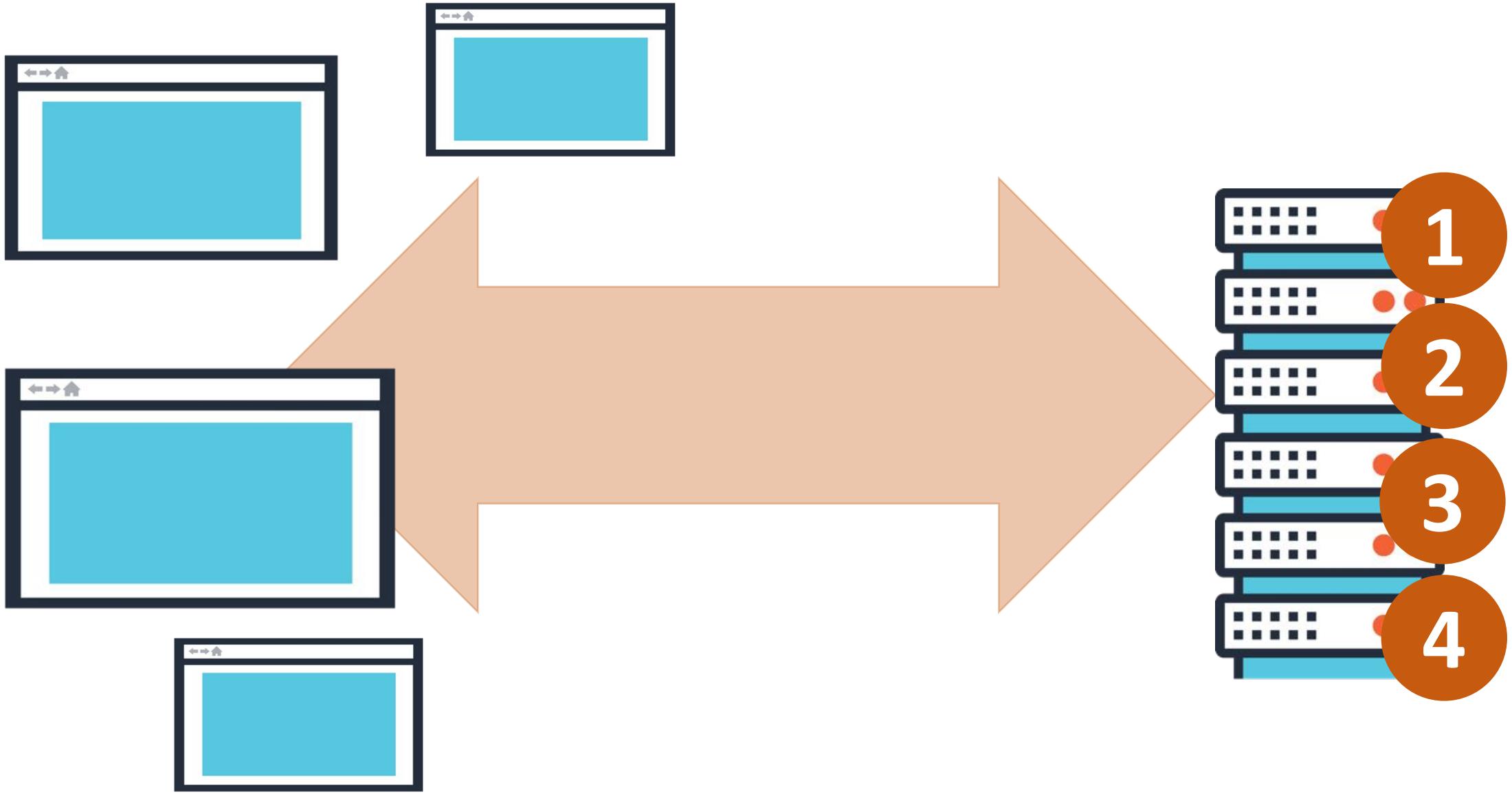
Insecure direct object references

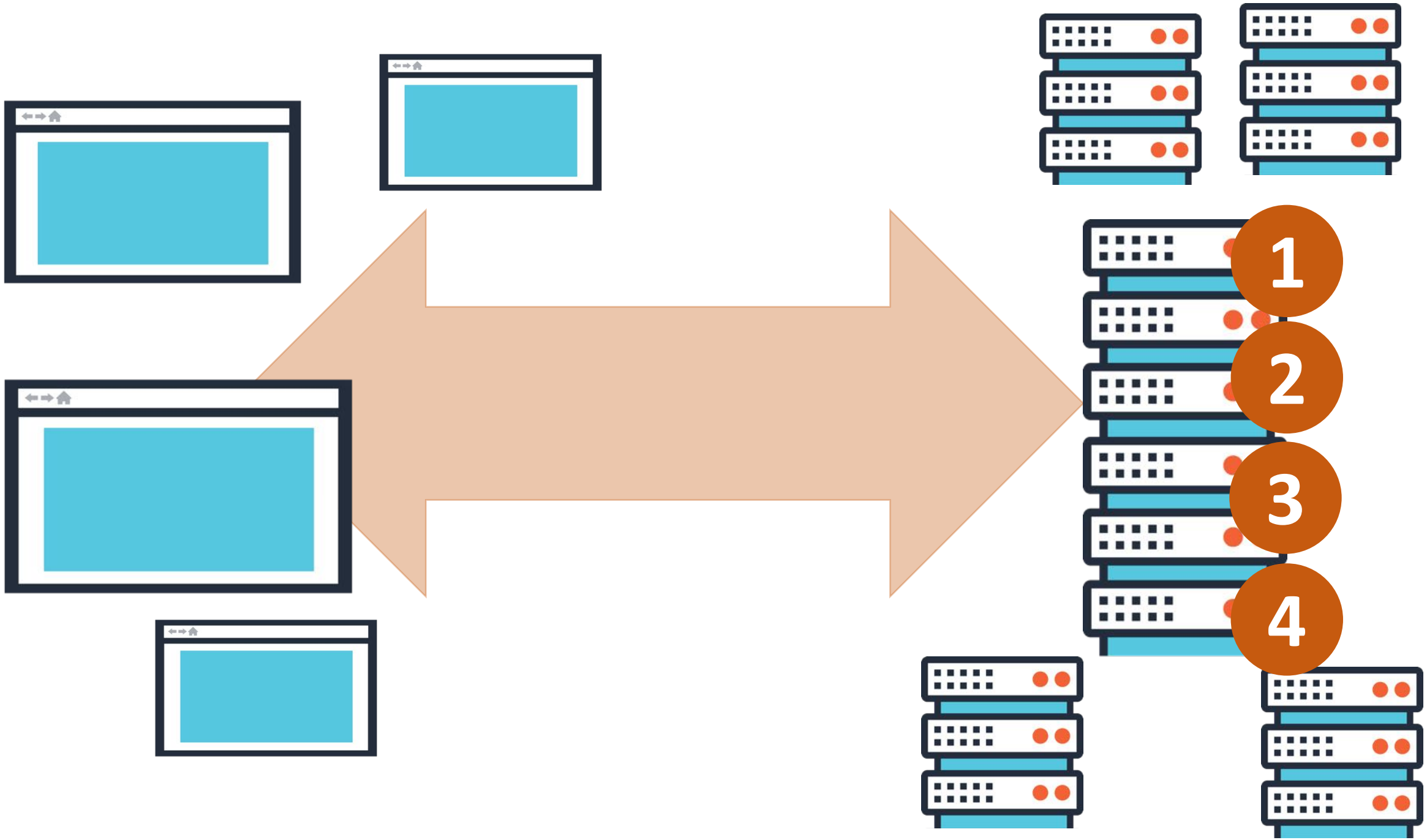
- Predictable identifiers enable the enumeration of resources
  - Dangerous if resources are not shielded by strict authorization checks
  - Many APIs only check authentication status, but not ***which*** user is authenticated
- The only proper mitigation is implementing proper authorization checks
  - E.g. checking if the current user is the owner of the resource
- The use of non-predictable identifiers is a complementary strategy
  - UUIDs are a good example of such an identifier
  - Just be careful about using them as primary keys in the database

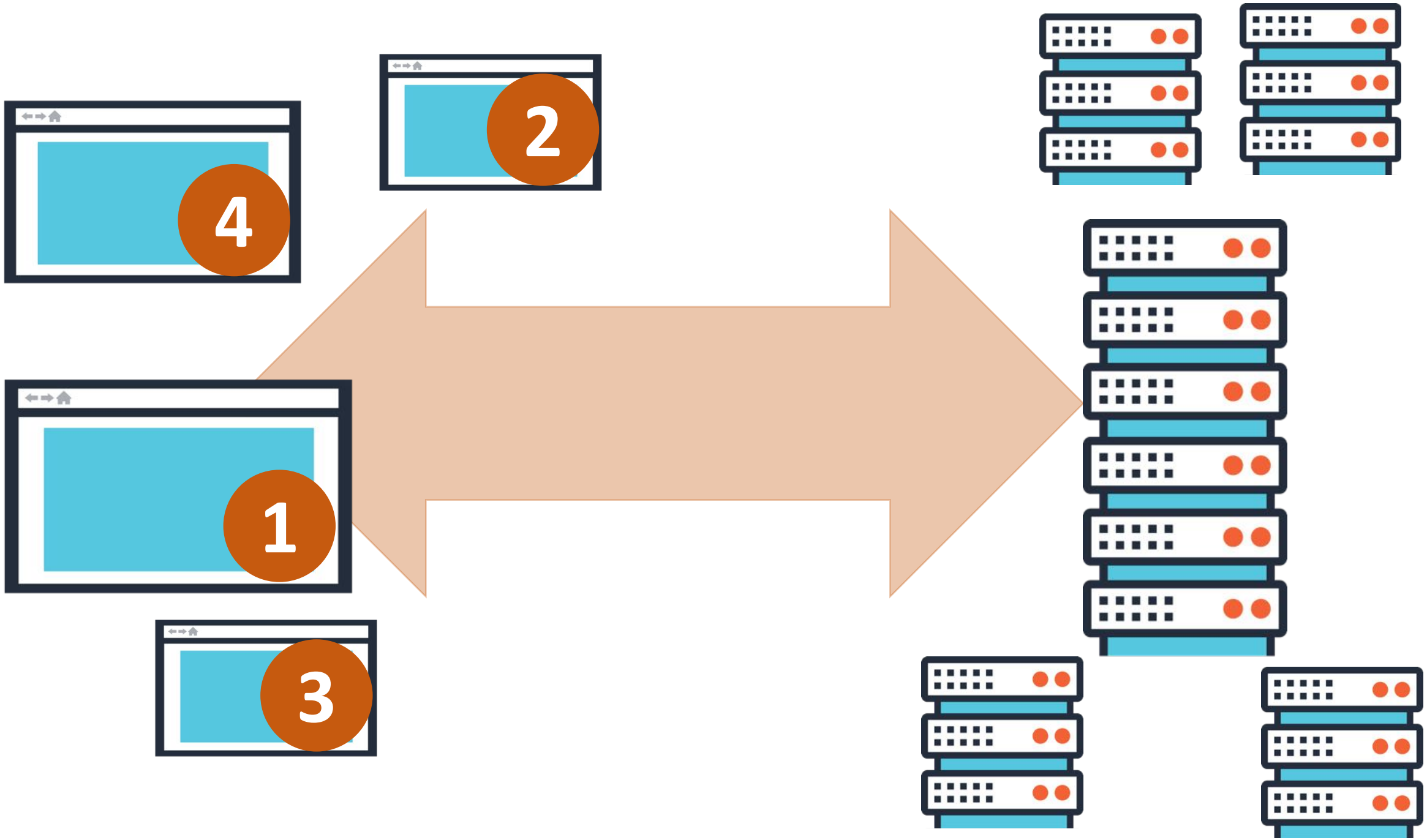
SECURITY PITFALL

Using insecure direct object references

Always complement a basic authentication check with appropriate authorization checks (e.g. ownership of a resource)


1
2
3
4


1
2
3
4


4
2
1
3

The trust levels of session data

- Server-side sessions share an ID with the client and store data on the server
  - Attacks on session management focus on guessing or stealing the ID
  - The data stored in the server-side session object can be considered trusted
- Client-side sessions are a completely different paradigm
  - The actual data is stored on the client, so it can be easily accessed
  - The data comes in from the client, and is untrusted by default
- Client-side sessions require additional data protection measures
  - Mandatory integrity checks to detect tampering with the data
  - Optional confidentiality mechanisms to prevent disclosure of information

SECURITY PITFALL

Mishandling client-side session data

Client-side session data can be read and manipulated, so you need to ensure confidentiality and integrity

JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ3ZWJzZWMuYmUiLCJ1c2VybmFtZSI6InBoaWxpcHBlIiwicm9sZSI6InRyYWluZXIifQ.Wj99INABfmWnm5CNaqoFwDPGFxDVq7PX8UktTFdrRtQ

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "iss": "websec.be",
  "username": "philippe",
  "role": "trainer"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  APPSECRET_HERE
) secret base64 encoded
```

https://jwt.io/

JWT TOKENS IN PRACTICE

- JWT tokens only represent claims to be exchanged securely
  - The data is base64-encoded, which offers no protection at all
  - The JWT specs support integrity (signing) and confidentiality (encryption)
- The default mode of operation is signing JWTs
  - The signature is part of the token, and can only be generated by the issuer
  - A valid signature indicates that the data of the JWT token has not been changed
- Many libraries offer decode functions that do not check integrity
  - Failing to fully understand the importance of integrity will cause misuse
  - Decoding is also a lot easier than verifying the integrity

```java
String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXUyJ9.eyJpc3MiOiJhdXRoMCJ9.AbIJTDMFc7yUa5MhvcP03nJPyCPzZtQc
try {
    Algorithm algorithm = Algorithm.HMAC256("secret");
    JWTVerifier verifier = JWT.require(algorithm)
        .withIssuer("auth0")
        .build(); //Reusable verifier instance
    DecodedJWT jwt = verifier.verify(token);
} catch (UnsupportedEncodingException exception){
    //UTF-8 encoding not supported
} catch (JWTVerificationException exception){
    //Invalid signature/claims
}
```

```java
String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXUyJ9.eyJpc3MiOiJhdXRoMCJ9.AbIJTDMFc7yUa5MhvcP03nJPyCPzZtQc
try {
    DecodedJWT jwt = JWT.decode(token);
} catch (JWTDecodeException exception){
    //Invalid token
}
```

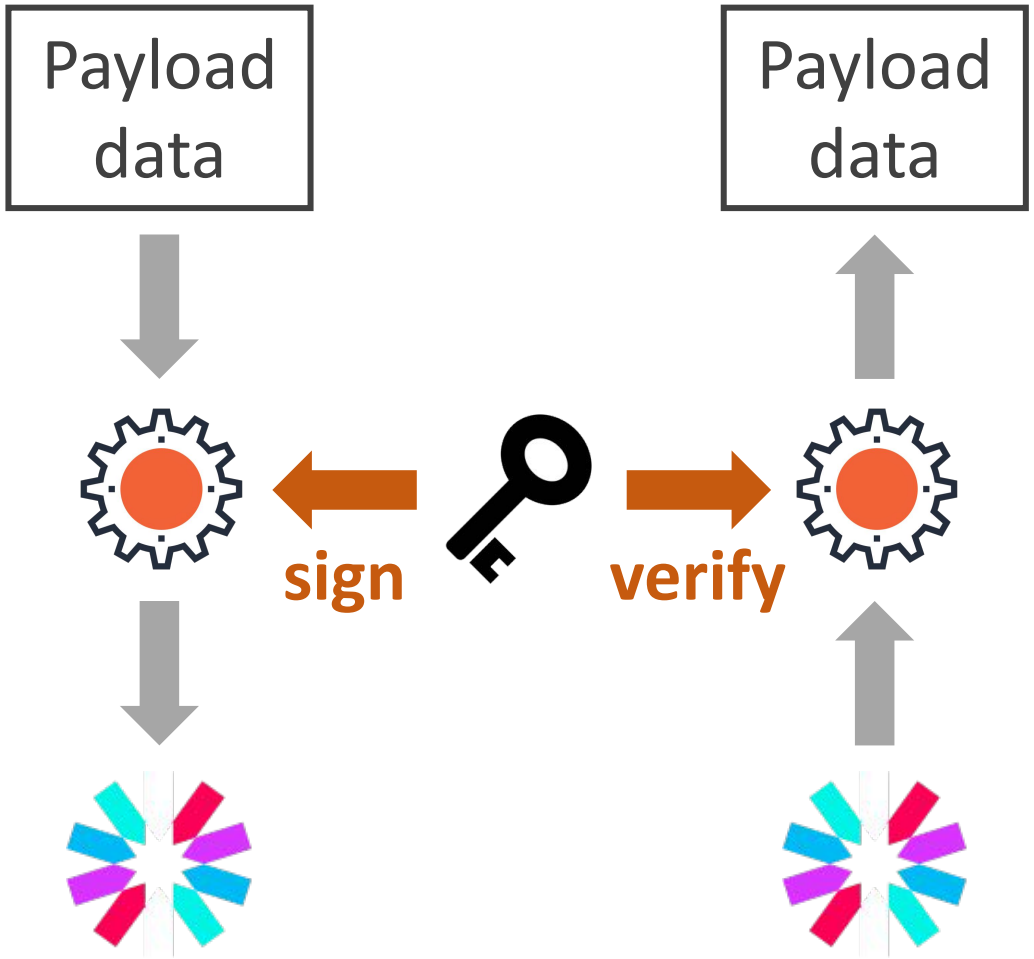
https://github.com/auth0/java-jwt

SECURITY PITFALL

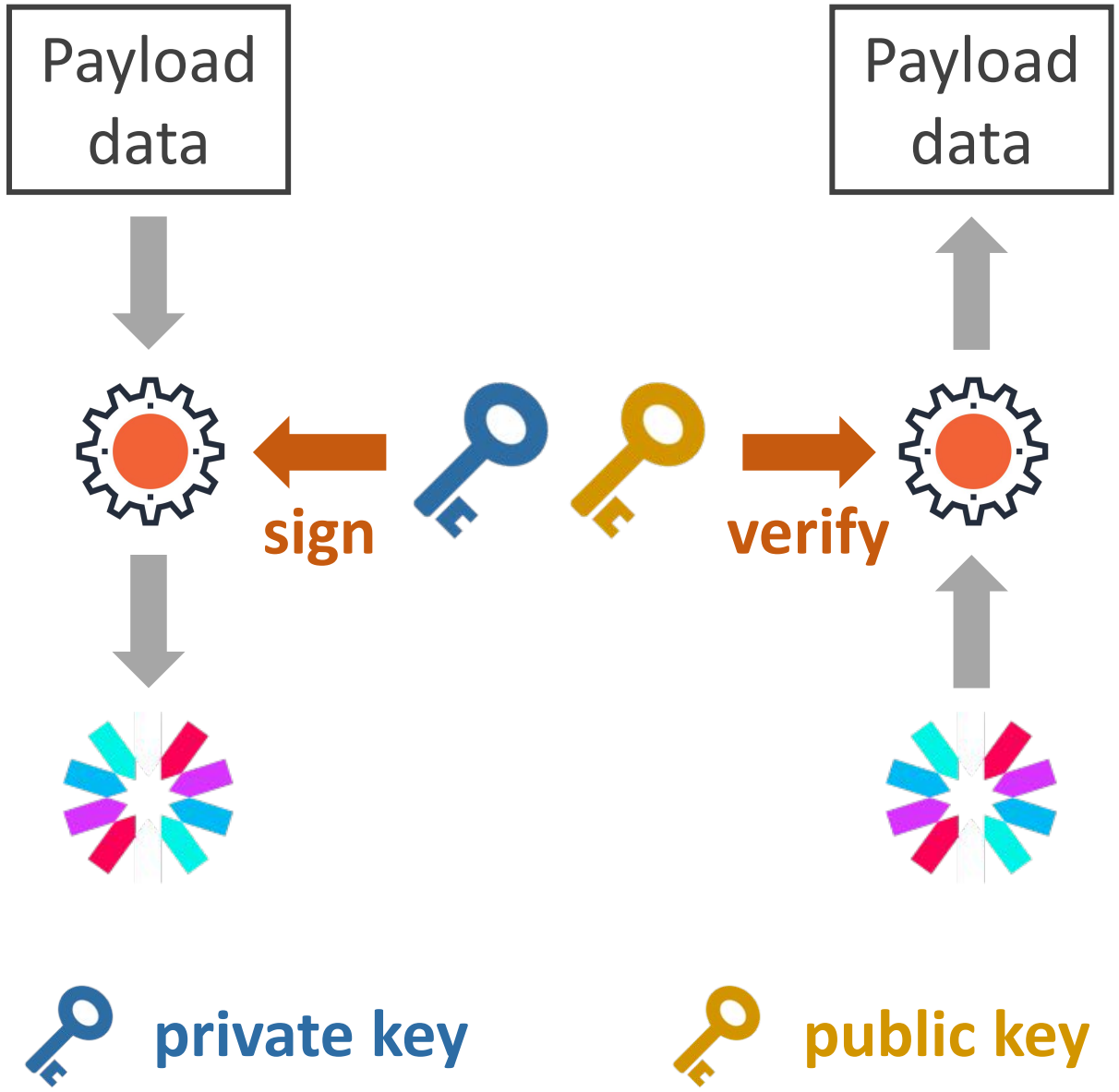
Not verifying the integrity of your JWT tokens

Many JWT libraries offer functions to get the data from a token without verifying its integrity. Never use them in the backend

Signing with a shared secret



Signing with a public/private key pair



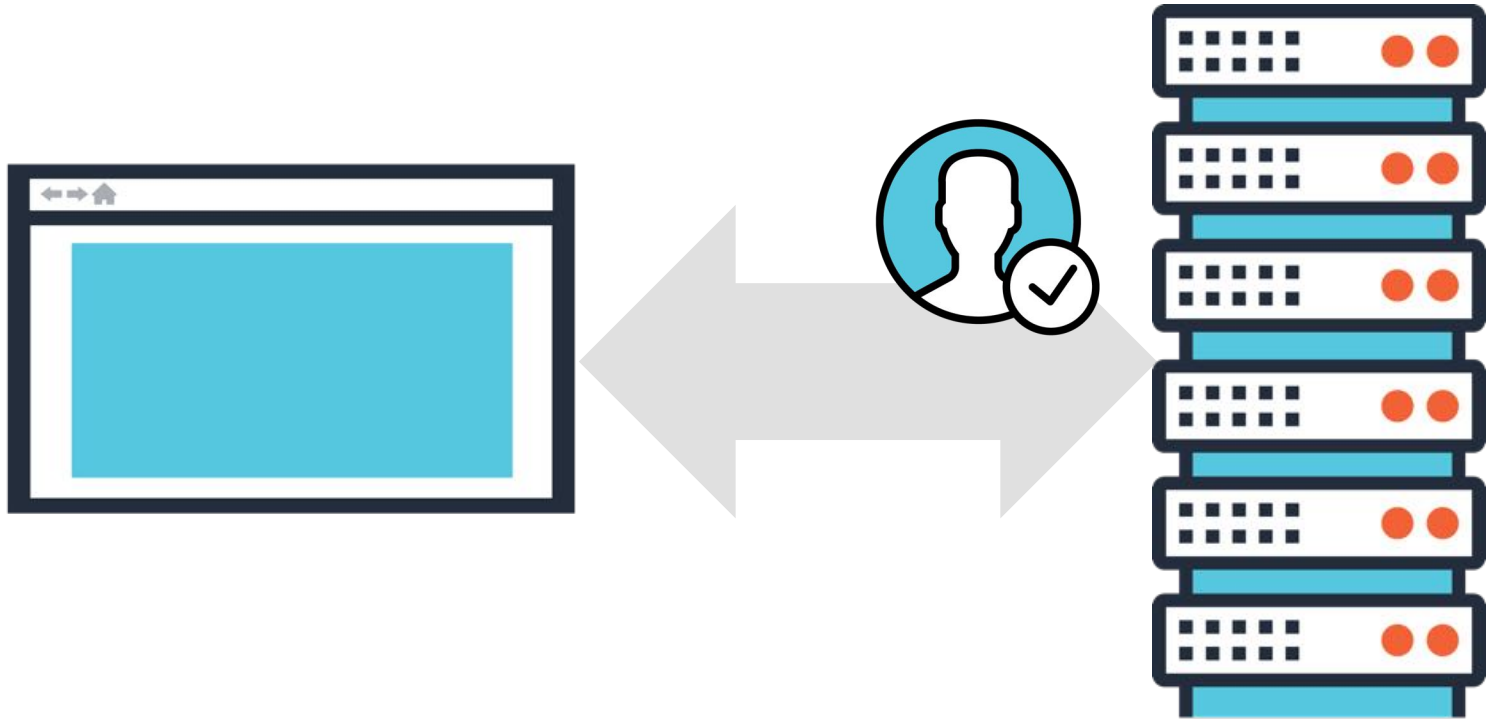
Signature schemes for JWT tokens

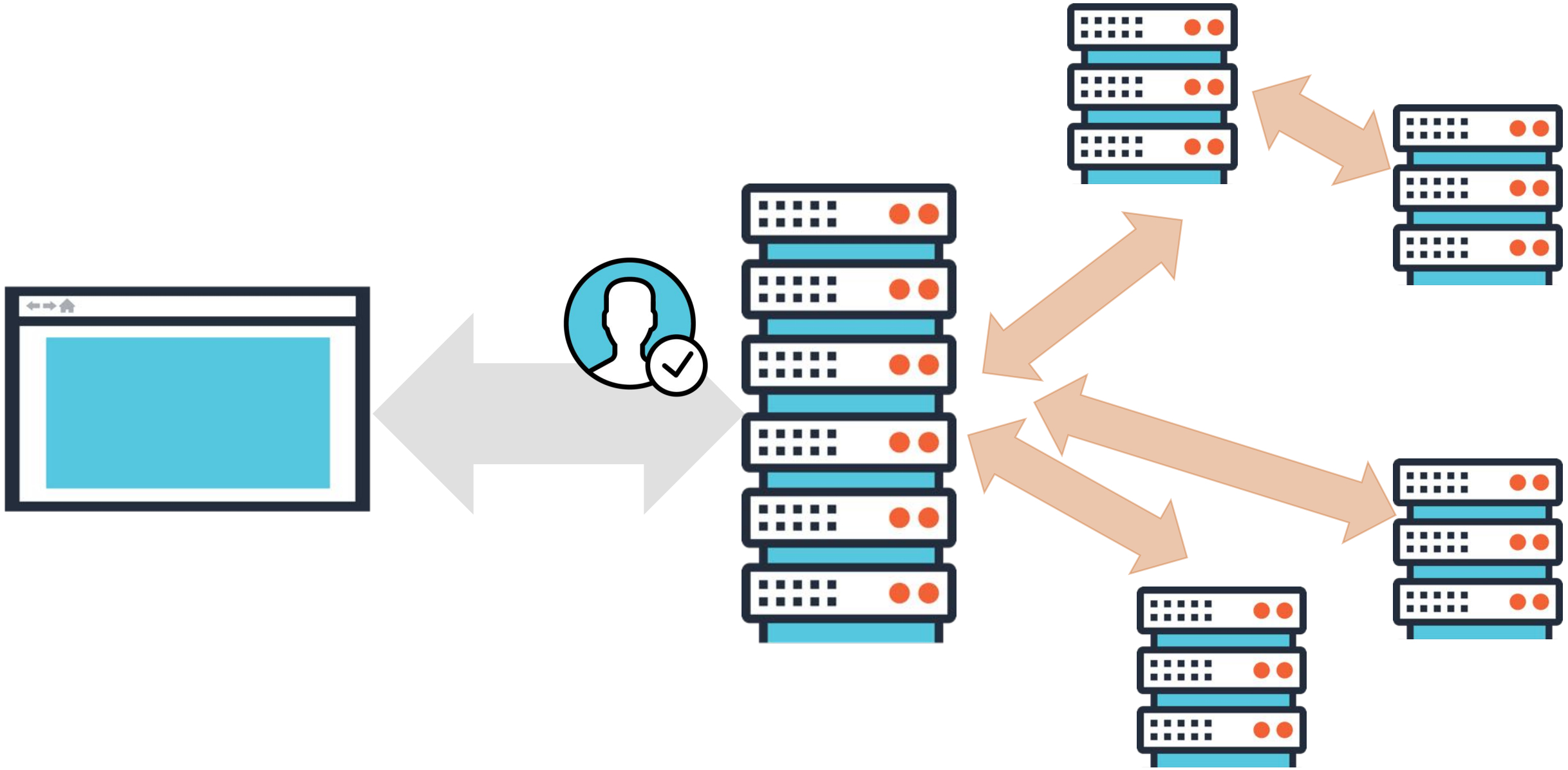
- Many developers only know about signing JWTs with a shared secret
  - This is perfectly valid within one application or even within one trust boundary
  - Breaks down when tokens need to be verified outside of your trust boundary
- The shared secret can never leave your backend application
  - Do not share it with your client application, or “friendly” APIs
  - If you need verification in those cases, sign the JWT with a private key instead
- The issuer should be the only one knowing the private key
  - The public key can be distributed to anyone
  - Tokens are signed with the private key, and verified with the public key

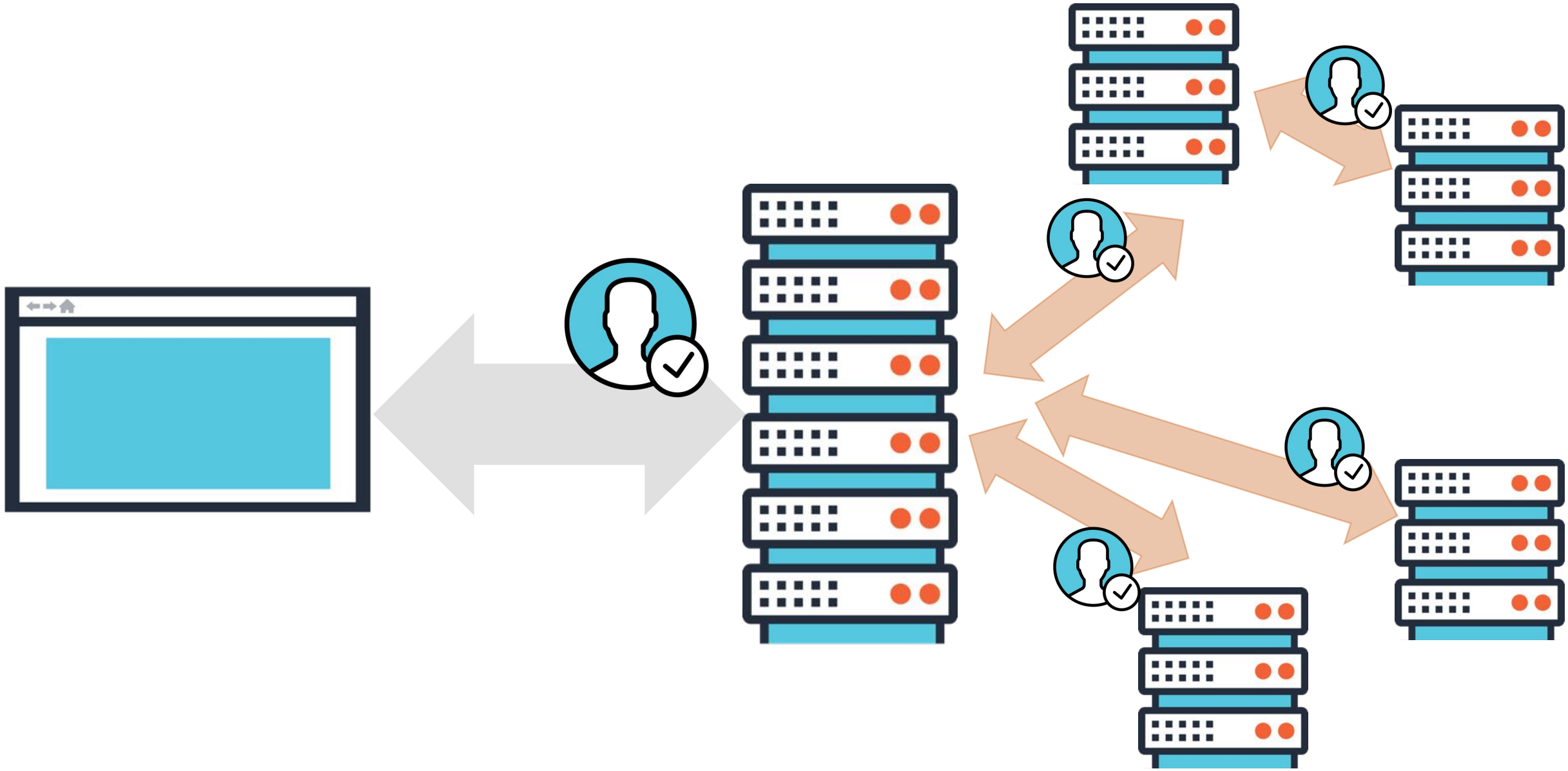
SECURITY PITFALL

Using the wrong signature scheme on JWT tokens

Shared secrets for verifying JWT tokens are for use within the boundaries of the application. Otherwise, use a public/private key pair







```
{
  "iss"       : "https://openid.wonderland.net",
  "sub"       : "alice",
  "aud"       : "s6BhdRkqt3",
  "nonce"     : "n-0S6_WzA2Mj",
  "exp"       : 1311281970,
  "iat"       : 1311280970,
  "auth_time" : 1311280969,
  "acr"       : "urn:acr:2fa",
  "name"      : "Alice Adams",
  "email"     : "alice@wonderland.net",
  "roles"     : [ "admin", "audit" ]
}
```

https://connect2id.com/blog/using-openid-connect-to-make-assertions-about-end-users

SECURITY PITFALL

Not propagating identity information

Calls are often delegated to internal systems or services. Ensure that these services possess all relevant identity information for making authorization decisions and creating an audit trail

Cookie: JWT=eyJhbGciOiJIUzI1Ni…

Authorization: Bearer yJhbGciOiJIUzI1Ni…

The properties of cookies

- Cookies are a mess, but they are compatible with the web
  - Browsers store and send cookies automatically
  - Cookies are present on all requests, including those coming from DOM elements
  - Cookies are compatible with web mechanisms such as CORS, SSE, WebSockets, …
- Securing cookie-based mechanisms requires a lot of effort
  - Cookie security flags need to be configured correctly
  - Cookie prefixes offer additional security, but require modifying the name
  - Cookies enable a nasty attack called Cross-Site Request Forgery (CSRF)
- Cookies are a nightmare to support in non-web applications

The properties of custom headers

- Custom headers are straightforward, but can be hard to use
  - Not handled automatically, so the application needs to store and send the value
  - The browser will not attach it to requests coming from DOM elements
  - The use of mechanisms such as CORS, SSE, WebSockets, … becomes more difficult
- Securing header-based mechanisms is also surprisingly difficult
  - You have to decide where to store the data in the client application
  - You're likely to mess up attaching the header to outgoing requests
  - But the good news is that custom headers do not suffer from CSRF
- Custom headers are a breeze to use in non-web applications

```
'request': function (config) {
    config.headers = config.headers || {};
    if ($localStorage.token) {
        config.headers.Authorization = 'Bearer ' + $localStorage.token;
    }
    return config;
},
```

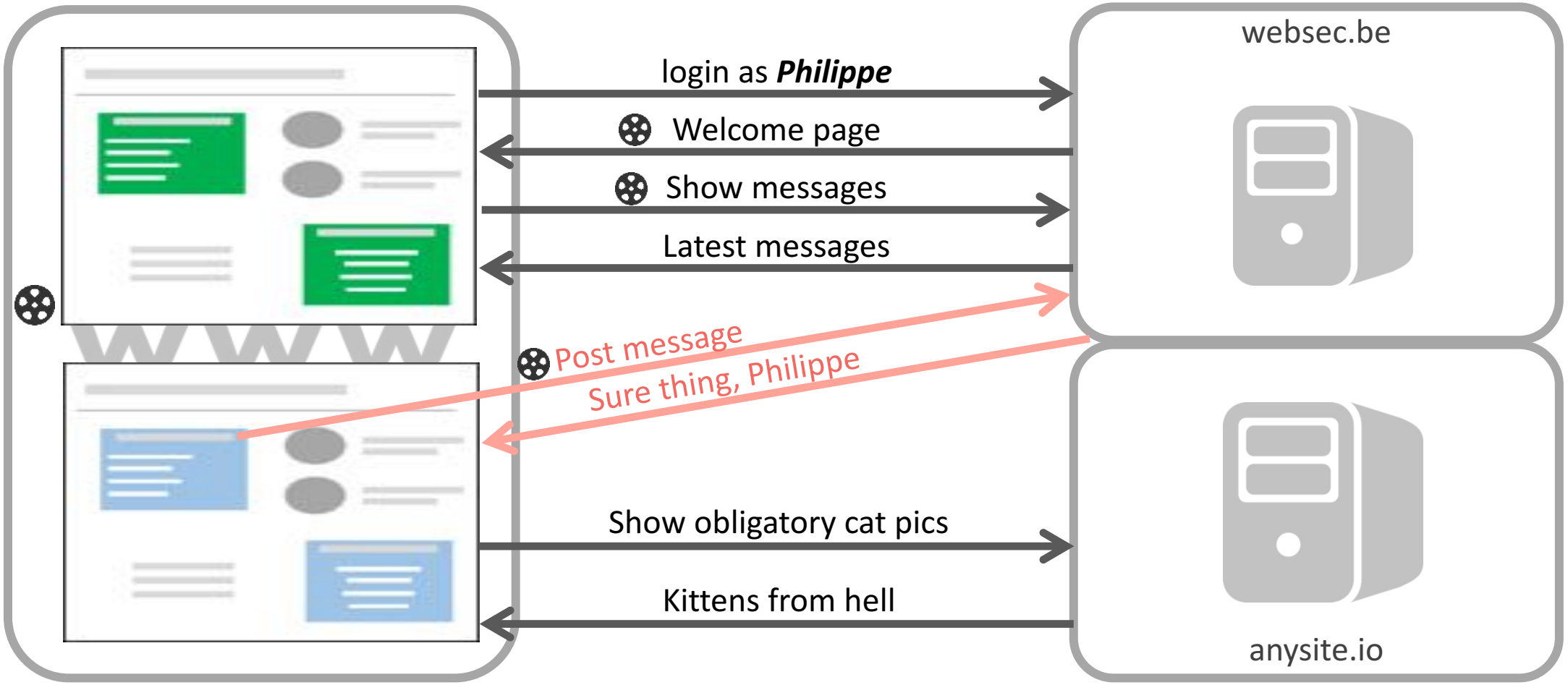
https://www.toptal.com/web/cookie-free-authentication-with-json-web-tokens-an-example-in-laravel-and-angularjs

SECURITY PITFALL

Minimizing the impact of the transport mechanism

Cookies are often frowned upon in an API world, and custom headers are preferred. Both have vastly different security properties, so make sure you understand them fully

THE UNDERESTIMATED THREAT OF CSRF



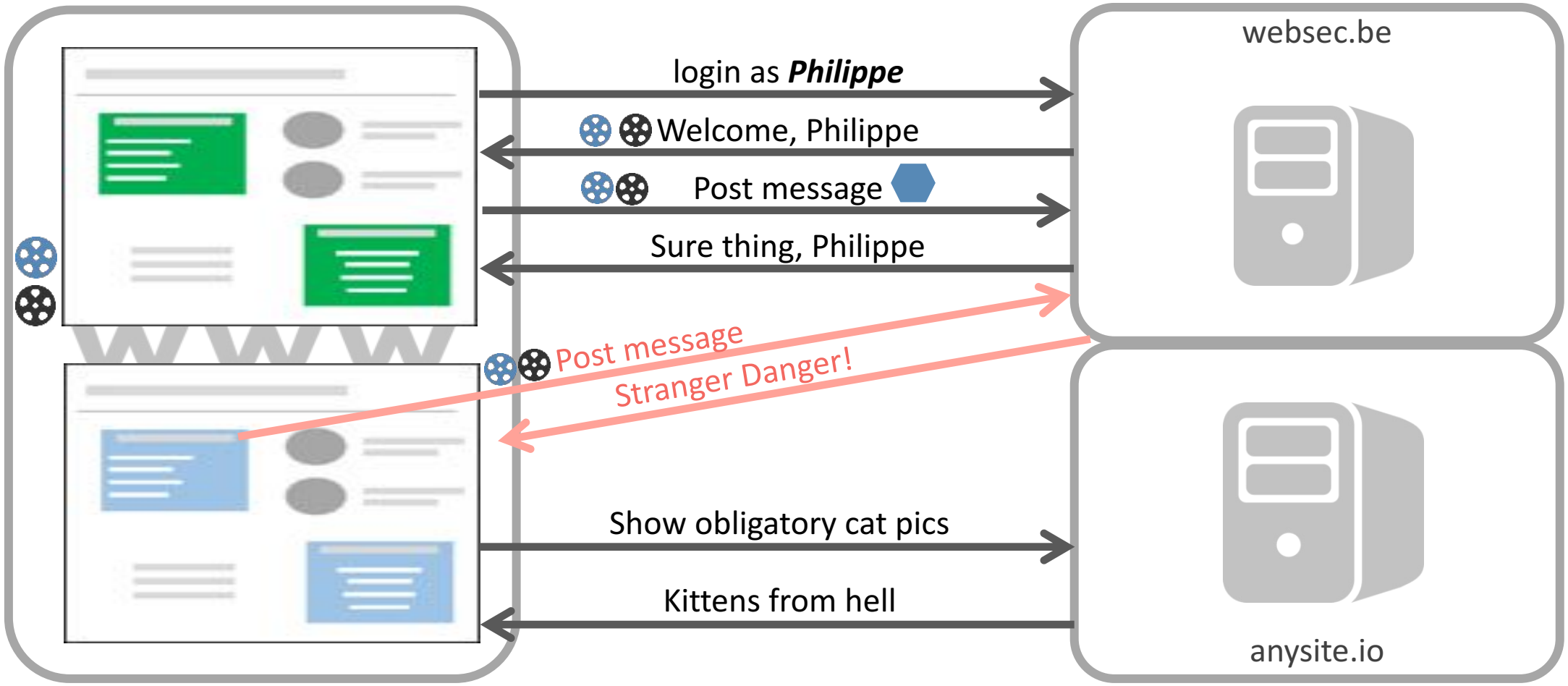
CSRF SOHO ROUTER ATTACK

https://arstechnica.com/information-technology/2014/03/hackers-hijack-300000-plus-wireless-routers-make-malicious-changes/

Cross-Site Request Forgery

- CSRF exists because the browser handles cookies very liberally
  - They are automatically attached to any outgoing request
  - By default, there's no mechanism to indicate the source or intent of a request
- Many APIs are unaware that any context can send requests
  - GET and POST requests are easy to trigger using DOM elements or XHR
  - PUT and DELETE requests are a different story
  - Defending against CSRF requires explicit action by the developer
- A traditional CSRF defense is using hidden form tokens

Defending your API against CSRF



Cookie value is copied to a header by JavaScript code

```
POST …
Cookie: SID=123, XSRF-TOKEN=abc
X-XSRF-TOKEN: abc
```

The relation between CSRF and CORS

- Cross-origin HTTP requests have always existed in the web
  - Examples are loading images from other origins, or submitting forms across origins
- CSRF matters in an API supporting “traditional” HTTP requests
  - GET/POST requests with traditional content types and no custom headers
  - These requests can easily be forged using traditional HTML elements
- APIs using “non-traditional” HTTP requests fall under the protection of CORS
  - Such a request can only be sent from JavaScript using XMLHttpRequest
  - Such a request triggers the ***Cross-Origin Resource Sharing (CORS)*** security policy
  - Such a request will only be allowed if the server explicitly approves it

X-Show-Me: The Money

Content-Type: application/json

SECURITY PITFALL

Underestimating the prevalence of CSRF

CSRF attacks exist when cookies are used for keeping session state. Verify if you're vulnerable and implement appropriate defenses.

If you do not use cookies, you do not need to worry about CSRF

/users/1’%20OR%20’1’=‘1

```
statement = conn.prepareStatement("SELECT * FROM Beers
  WHERE name LIKE ?");
statement.setString(0, parameter);
```

Input validation is an important first line of defense

- Limiting the number of valid inputs reduces the attack surface
  - Untrusted data should be validated before using it
  - The restrictions that can be imposed depend on the type of content

- Best practices for input validation
  - Only accept content types that you expect, and reject everything else
  - Validate every input against its expected data type
  - Impose sensible length restrictions, and always set a strict upper bound
  - Always use a secure parser to process input

But input validation only gets you so far

- Input validation targets symptoms, not the root cause of the issue
  - Injection needs to be addressed in the code, not at the input level
- Once the data is complex enough, validation bypasses will exist
  - Validation or sanitization is hard to get right, so do not solely rely on them
  - A good example are the huge XSS filter evasion cheat sheets
- And sometimes, it's just not the API's responsibility
  - Cross-site scripting in web applications is the perfect example
  - The API has no idea where the data will be used, so it cannot render it safe
  - The client-side application needs to handle this, as e.g. Angular does out of the box

SECURITY PITFALL

Over or underestimating input validation

Even though input validation is a good first line of defense, it will fail as the only defense. Do not rely on input validation alone

Question Everything

How is this different from what we used to do?

Do we really understand what we're doing?

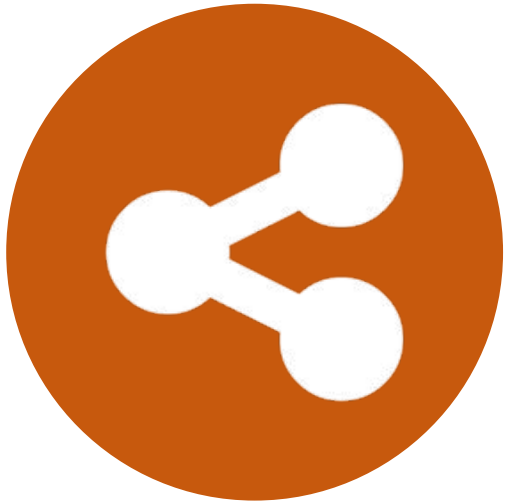
Have we validated the integrity and format of that data?

…

Now it's up to you …

Secure

@PhilippeDeRyck



Share