



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

WEB SECURITY



Content

1. Security basics
2. Secure connections with HTTPS
3. Prevent info leaks
4. Popular types of attack

Content

1. **Security basics**
2. Secure connections with HTTPS
3. Prevent info leaks
4. Popular types of attack

1. Security basics

- A vulnerability (sometimes called a security bug) is a type of bug that could be used for abuse.
- When an application is not secure, different people could be affected.

User	<ul style="list-style-type: none">• Sensitive information, such as personal data, could be leaked or stolen.• Content could be tampered with. A tampered site could direct users to a malicious site.
Application	<ul style="list-style-type: none">• User trust may be lost.• Business could be lost due to downtime or loss of confidence as a result of tampering or system shortage.
Other system	<ul style="list-style-type: none">• A hijacked application could be used to attack other systems, such as with a denial-of-service attack using a botnet.

1. Security basics

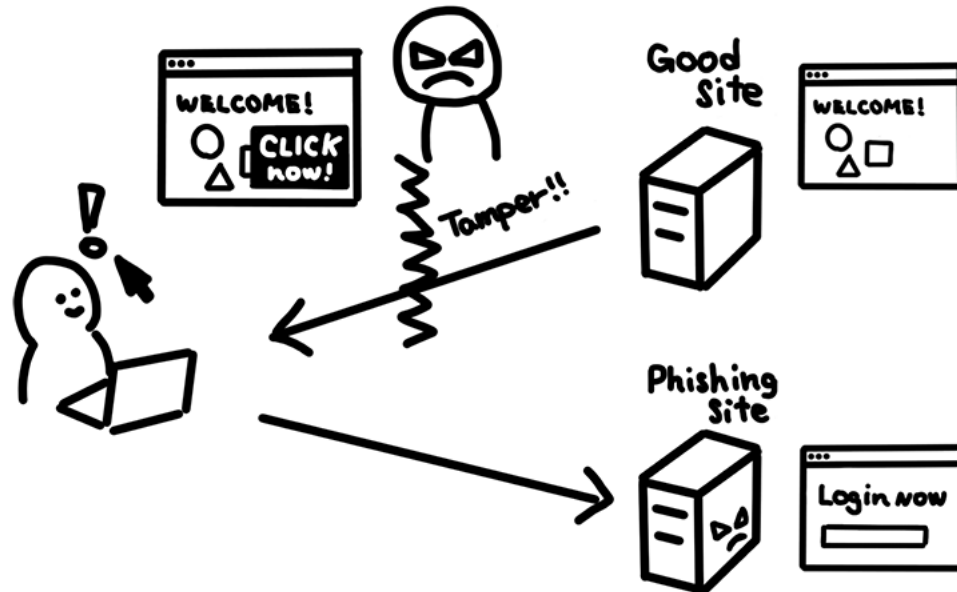
- When a malicious party uses vulnerabilities or lack of security features to their advantage to cause damage, it is called an attack. There are 2 different types: **Active** and **Passive**:

❑ **Active attacks:** the attacker tries to break into the application directly. There are a variety of ways this could be done, from using a false identity to access sensitive data (masquerade attack) to flooding your server with massive amounts of traffic to make your application unresponsive (denial of service attack).

Active attacks can also be done to data in transit. An attacker could modify your application data before it gets to a user's browser, showing modified information on the site or direct the user to an unintended destination. This is sometimes called **modification of messages**.

1. Security basics

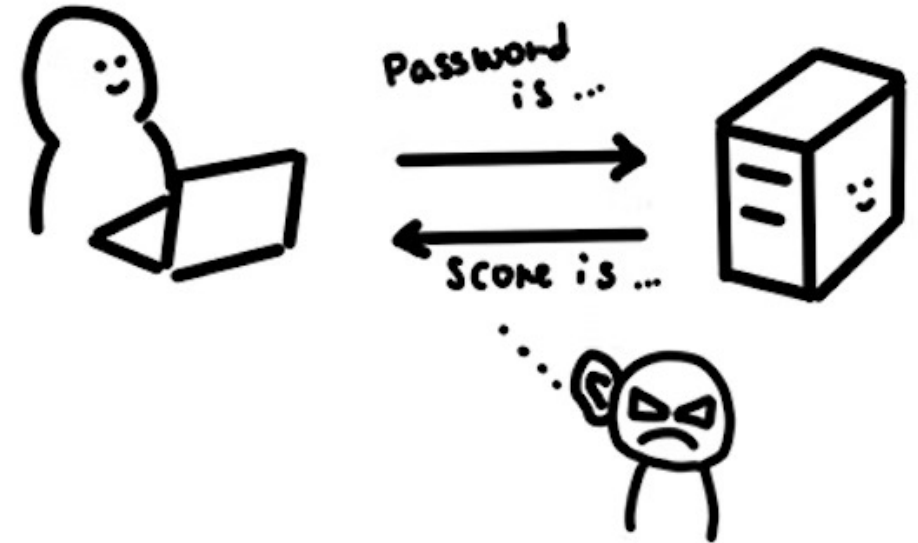
- *Active attack example*: when you logged into free public wifi and seen ads wrapped around web pages you are accessing, that's exactly what **modification of message** is! The wifi access point injected their advertising into a website before it got to your browser. In many cases, you might dismiss it as "just ads for free wifi", but imagine if the same technique is used to replace some of the javascript or link to a phishing site. Your site may be used by an attacker to misguide users without you noticing.



1. Security basics

❑ **Passive attack:** the attacker tries to collect or learn information from the application but does not affect the application itself.

On your web traffic, an attacker could capture data between the browser and the server collecting usernames, passwords, users's browsing history, data exchanged.



Content

1. Security basics
2. **Secure connections with HTTPS**
3. Prevent info leaks
4. Popular types of attack

2. Secure connections with HTTPS

- SSL, or Secure Sockets Layer, is an encryption-based Internet security protocol. It was first developed by Netscape in 1995 for the purpose of ensuring privacy, authentication, and data integrity in Internet communications. SSL is the predecessor to the modern TLS encryption used today.
- In order to provide a high degree of privacy, SSL encrypts data that is transmitted across the web. This means that anyone who tries to intercept this data will only see a garbled mix of characters that is nearly impossible to decrypt.
- SSL initiates an authentication process called a handshake between two communicating devices to ensure that both devices are really who they claim to be.
- SSL also digitally signs data in order to provide data integrity, verifying that the data is not tampered with before reaching its intended recipient.

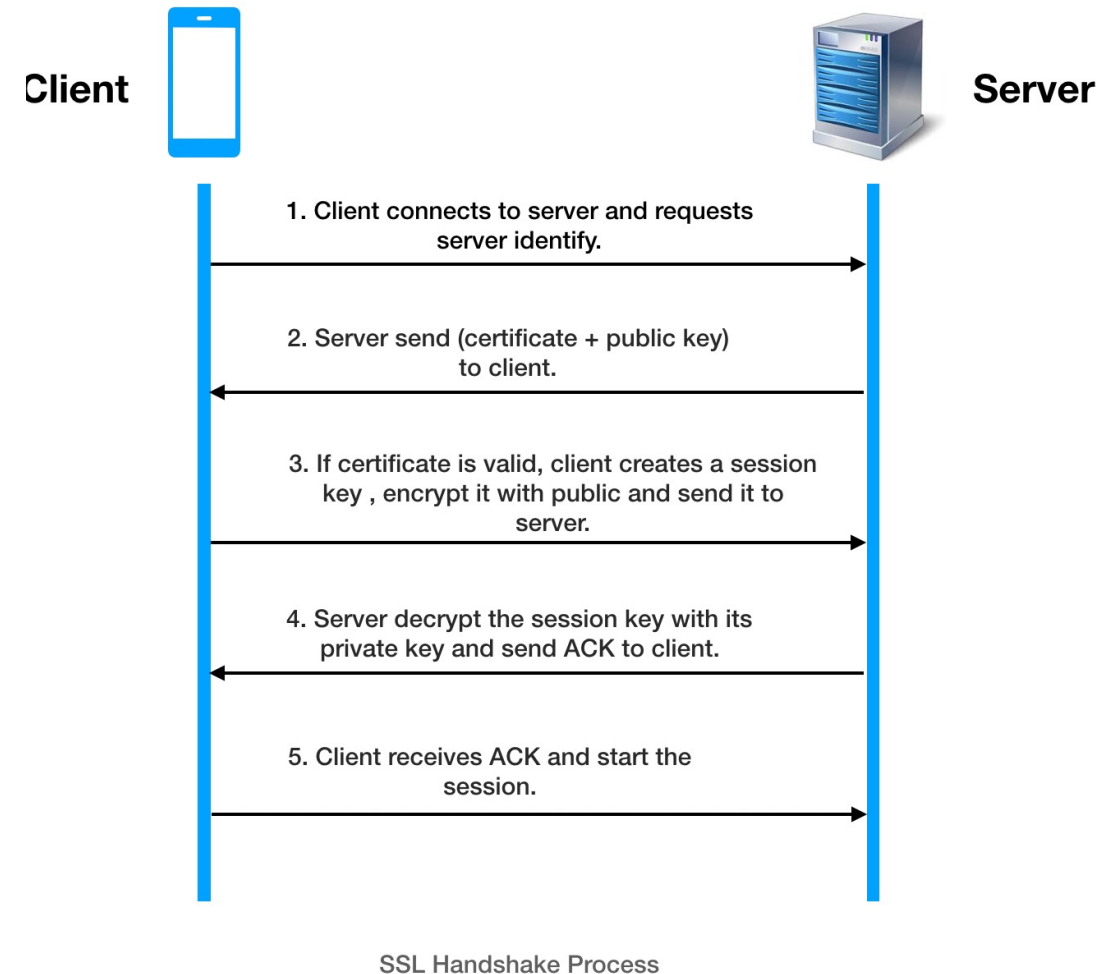
💡 There have been several iterations of SSL, each more secure than the last. In 1999 SSL was updated to become TLS, so they usually called SSL/TLS.

2. Secure connections with HTTPS

- SSL certificates are what enable websites to move from HTTP to HTTPS, which is more secure. An SSL certificate is a data file hosted in a website's origin server.
- SSL certificates make SSL/TLS encryption possible, and they contain the website's public key and the website's identity, along with related information. Devices attempting to communicate with the origin server will reference this file to obtain the public key and verify the server's identity. The private key is kept secret and secure.
- Each browser has its own CA Certificate List. If a certificate is issued by one of CA in the list, it is trusted.

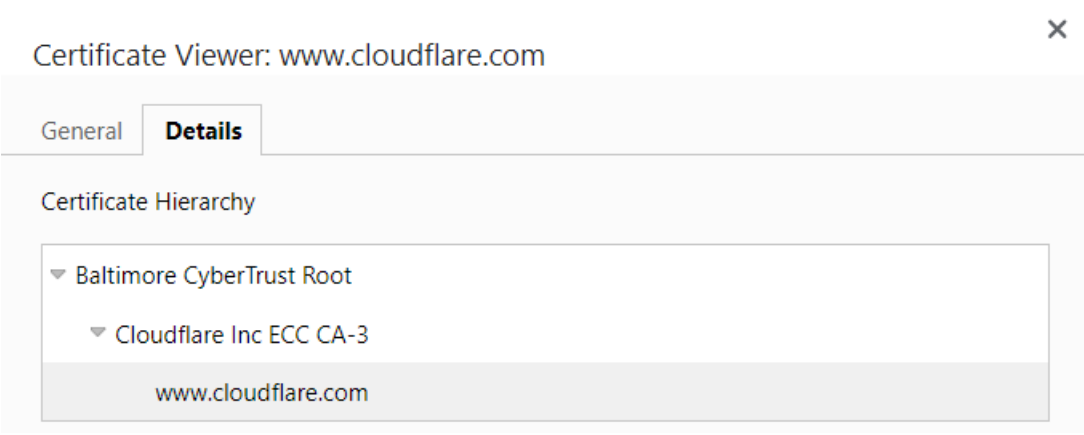
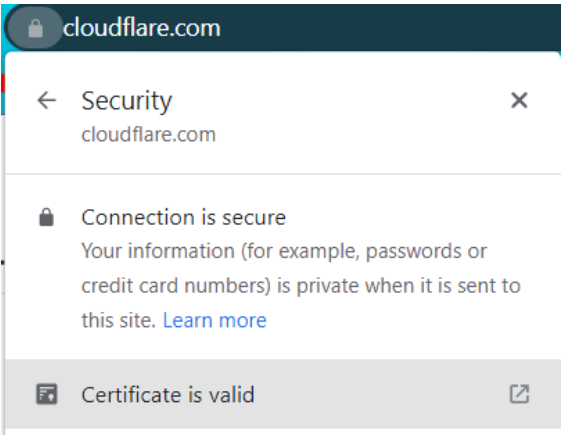
Eg: Firefox's CA Certificate List:

<https://ccadb-public.secure.force.com/mozilla/IncludedCACertificateReport>



2. Secure connections with HTTPS

- You can check certificate of a website by browser
- Eg: certificate of Cloudflare is issued by Baltimore CyberTrust Root, which in trusted list of browser



ccadb-public.secure.force.com/mozilla/IncludedCACertificateReport					
DigiCert	Baltimore	CyberTrust	Baltimore CyberTrust Root	020000B9	16AF57A9F676B0AB126095AA5E BADEF22AB31119D644AC95CD4 B93DBF3F26AEB

2. Secure connections with HTTPS

- Transport Layer Security, or TLS, is a widely adopted security protocol that provides privacy and data integrity for Internet communications. Implementing TLS is a standard practice for building secure web apps.
- HTTPS is HTTP with SSL/TLS encryption.
- HTTPS use TLS to encrypt normal HTTP requests and response, making it safer and more secure. A website that uses HTTPS has https:// in the beginning of its URL instead of http://
- Almost browsers as Chrome, Firefox,... mark all HTTP websites as “Not secure”



Secure

https://example.com



Not secure

http://example.com

2. Secure connections with HTTPS

- When attacker try to intercept a request:

👉 HTTP:

```
POST /login HTTP/1.1
User-Agent: curl/7.63.0 libcurl/7.63.0
OpenSSL/1.1.1 zlib/1.2.11
Host: www.example.com
Accept-Language: en
{
  "username": "admin",
  "password": "1234"
}
```

HTTP vs HTTPS



👉 HTTPS:

```
t8Fw6T8UV81pQfyhDkhebbz7+oiwldr1j2gHBB3L3RFTRs
QCpaSnSBZ78Vme+DpDVJPvZdZUZHpbzbbcmSW1+3xX
GsERHg9YDmpYk0VVDiRvw1H5miNieJ/FNUjgH0BmVR
WII6+T4MnDwmCMZUI/orxP3HGwYCSlvyzS3MpmSe4i
aWKCOHQ==
```

Content

1. Security basics
2. Secure connections with HTTPS
3. **Prevent info leaks**
4. Popular types of attack

3. Prevent info leaks

- "Origin" is a combination of a scheme (also known as the protocol, for example HTTP or HTTPS), hostname, and port (if specified). For example, given a URL of `https://www.example.com:443/foo`, the "origin" is `https://www.example.com:443`.

Origin

https://www.example.com:443
scheme host name port

- The same-origin policy is a browser security feature that restricts how documents and scripts on one origin can interact with resources on another origin.
- A browser can load and display resources from multiple sites at once. You might have multiple tabs open at the same time, or a site could embed multiple iframes from different sites. If there is no restriction on interactions between these resources, and a script is compromised by an attacker, the script could expose everything in a user's browser.

3. Prevent info leaks

Origin A	Origin B	Explanation of whether Origin A and B are "same-origin" or "cross-origin"
https://www.example.com:443	https:// www.evil.com :443	cross-origin: different domains
	https:// example.com :443	cross-origin: different subdomains
	https:// login .example.com:443	cross-origin: different subdomains
	http ://www.example.com:443	cross-origin: different schemes
	https://www.example.com: 80	cross-origin: different ports
	https://www.example.com:443	same-origin: exact match
	https://www.example.com	same-origin: implicit port number (443) matches

3. Prevent info leaks

- Generally, embedding a cross-origin resource is permitted, while reading a cross-origin resource is blocked.

iframes	Cross-origin embedding is usually permitted (depending on the X-Frame-Options directive), but cross-origin reading (such as using JavaScript to access a document in an iframe) isn't.
CSS	Cross-origin CSS can be embedded using a <code><link></code> element or an <code>@import</code> in a CSS file. The correct <code>Content-Type</code> header may be required.
forms	Cross-origin URLs can be used as the <code>action</code> attribute value of form elements. A web application can write form data to a cross-origin destination.
images	Embedding cross-origin images is permitted. However, reading cross-origin image data (such as retrieving binary data from a cross-origin image using JavaScript) is blocked.
multimedia	Cross-origin video and audio can be embedded using <code><video></code> and <code><audio></code> elements.
script	Cross-origin scripts can be embedded; however, access to certain APIs (such as cross-origin fetch requests) might be blocked.

3. Prevent info leaks

❑ Example 1: A webpage on the web.dev domain includes this iframe:



```
1 <iframe id="iframe" src="https://example.com/some-page.html" alt="Sample iframe">
  </iframe>
```

The webpage's JavaScript includes this code to get the text content from an element in the embedded page. Is this JavaScript allowed?




```
1 const iframe = document.getElementById('iframe');
2 const message = iframe.contentDocument.getElementById('message').innerText;
```

➤ No. Since the iframe is not on the same origin as the host webpage, the browser doesn't allow reading of the embedded page.

3. Prevent info leaks

❑ Example 2: A webpage on the web.dev domain includes this form. Can this form be submitted?



```
1 <form action="https://example.com/results.json">
2   <label for="email">Enter your email: </label>
3   <input type="email" name="email" id="email" required>
4   <button type="submit">Subscribe</button>
5 </form>
```

➤ Yes. Form data can be written to a cross-origin URL specified in the action attribute of the `<form>` element.

3. Prevent info leaks

❑ Example 3: A webpage on the web.dev domain includes this iframe. Is this iframe embed allowed?



```
1 <iframe id="iframe" src="https://example.com/some-page.html" alt="Sample iframe">
  </iframe>
```

➤ Usually. Cross-origin iframe embeds are allowed as long as the origin owner hasn't set the X-Frame-Options HTTP header to deny or sameorigin.

3. Prevent info leaks

❑ Example 4: A webpage on the web.dev domain includes this canvas:

```
1 <canvas id="bargraph"></canvas>
```

The webpage's JavaScript includes this code to draw an image on the canvas. Can this image be drawn on the canvas?

```
1 const context = document.getElementById('bargraph').getContext('2d')
2 const img = new Image()
3 img.onload = function () {
4   context.drawImage(img, 0, 0)
5 }
6 img.src = 'https://example.com/graph-axes.svg'
7
```

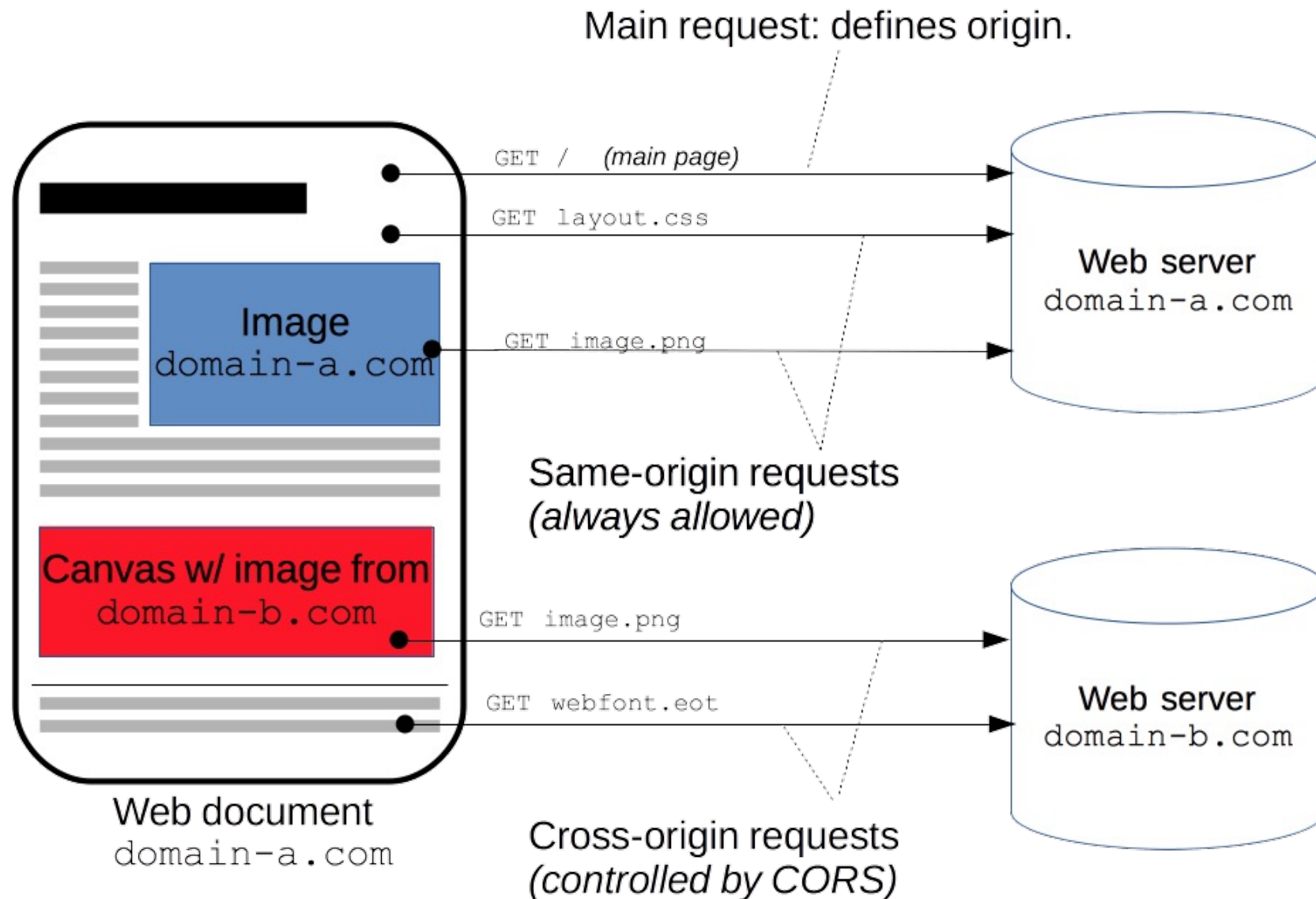
➤ Yes. Although the image is on a different origin, loading it as an img source does not require CORS. However, accessing the binary of the image using JavaScript such as `getImageData`, `toBlob` or `toDataURL` requires an explicit permission by CORS.

3. Prevent info leaks

- In a modern web application, an application often wants to get resources from a different origin. In other words, there are public resources that should be available for anyone to read, but the same-origin policy blocks that.
- **Cross-Origin Resource Sharing (CORS)** fixes this in a standard way. Enabling CORS lets the server tell the browser it's permitted to use an additional origin.
- When you want to get a public resource from a different origin, the resource-providing server needs to tell the browser "This origin where the request is coming from can access my resource". The browser remembers that and allows cross-origin resource sharing.

💡 Same-origin policy is features of **browsers** to prevent a malicious site from reading another site's data. You can still access the resource without browser, for example using curl, postman,... or write your own code to send HTTP request

3. Prevent info leaks



3. Prevent info leaks

```
> const xhr = new XMLHttpRequest()
  xhr.onreadystatechange = () => {
    if (xhr.readyState === 4) {
      xhr.status === 200 ? console.log(xhr.responseText) : console.error('error')
    }
  }
  xhr.open('GET', 'https://google.com')
  xhr.send()
< undefined
```

✖ Failed to load <https://google.com/>: Redirect from '<https://google.com/>' to 'https://www.google.it/?gfe_rd=cr&dcr=0&ei=3yDHWvnPN9LCXsWxp9AN' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin '<http://localhost:1313>' is therefore not allowed access.

✖ ▶ error VM387:4

▶ XHR finished loading: GET "<https://google.com/>". VM387:8

> |

3. Prevent info leaks

How does CORS work?

- **Step 1 - client (browser) request:** When the browser is making a cross-origin request, the browser adds an Origin header with the current origin (scheme, host, and port).
- **Step 2 - server response:** On the server side, when a server sees this header, and wants to allow access, it needs to add an Access-Control-Allow-Origin header to the response specifying the requesting origin (or * to allow any origin.)
- **Step 3 - browser receives response:** When the browser sees this response with an appropriate Access-Control-Allow-Origin header, the browser allows the response data to be shared with the client site.

Content

1. Security basics
2. Secure connections with HTTPS
3. Prevent info leaks
4. **Popular types of attack**

4. Popular types of attack

4.1. Clickjacking:

- An attack called "clickjacking" embeds a site in an iframe and overlays transparent buttons which link to a different destination. Users are tricked into thinking they are accessing your application while sending data to attackers. To block other sites from embedding your site in an iframe, add a content security policy with frame-ancestors directive to the HTTP headers.

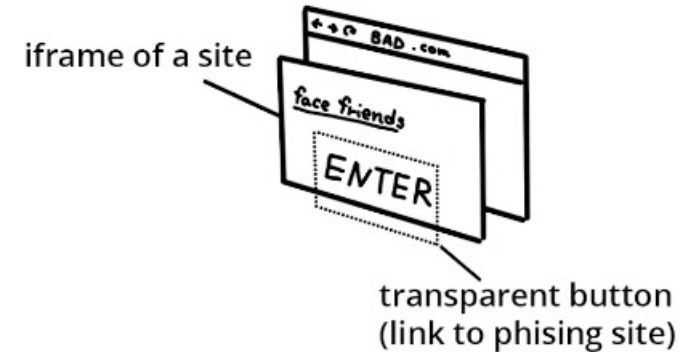


Figure: Clickjacking mechanism illustrated in 3 separate layers (base site, iframed site, transparent button).

4. Popular types of attack

4.1. Clickjacking:

- Eg: The target website iframe is positioned within the browser so that there is a precise overlap of the target action with the decoy website using appropriate width and height position values. Absolute and relative position values are used to ensure that the target website accurately overlaps the decoy regardless of screen size, browser type and platform. The z-index determines the stacking order of the iframe and website layers. The opacity value is defined as 0.0 (or close to 0.0) so that the iframe content is transparent to the user. Browser clickjacking protection might apply threshold-based iframe transparency detection (for example, Chrome version 76 includes this behavior but Firefox does not). The attacker selects opacity values so that the desired effect is achieved without triggering protection behaviors.

```
1 <head>
2   <style>
3     #target_website {
4       position:relative;
5       width:128px;
6       height:128px;
7       opacity:0.00001;
8       z-index:2;
9     }
10    #decoy_website {
11      position:absolute;
12      width:300px;
13      height:400px;
14      z-index:1;
15    }
16  </style>
17 </head>
18 ...
19 <body>
20   <div id="decoy_website">
21     ... decoy web content here ...
22   </div>
23   <iframe id="target_website" src="https://vulnerable-website.com">
24   </iframe>
25 </body>
```

4. Popular types of attack

4.1. Clickjacking:

- Clickjacking can be prevented by implementing a Content Security Policy (frame-ancestors).

The following CSP whitelists frames to the same domain only:

 Content-Security-Policy: frame-ancestors 'self';

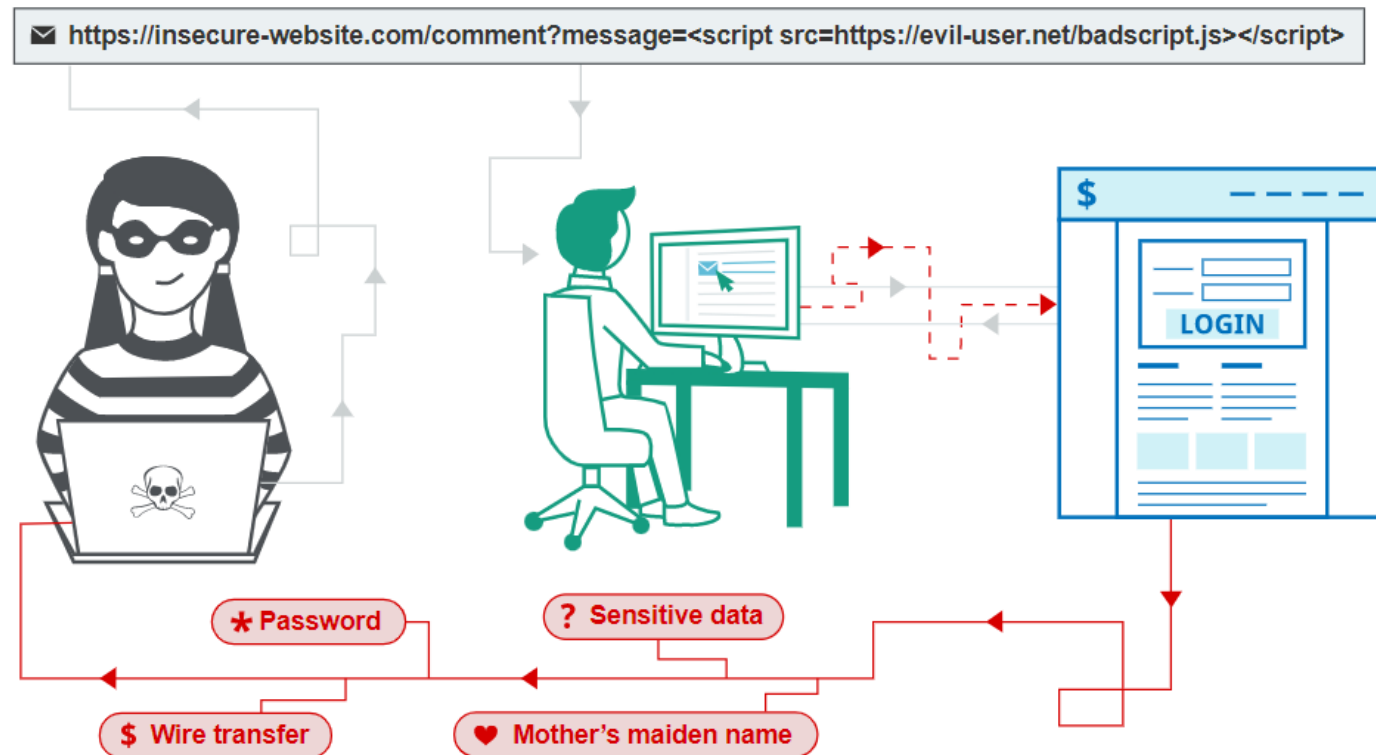
Alternatively, framing can be restricted to named sites:

 Content-Security-Policy: frame-ancestors normal-website.com;

4. Popular types of attack

4.2. XSS:

- Cross-site scripting (XSS) is a security exploit which allows an attacker to inject into a website malicious client-side code. This code is executed by the victims and lets the attackers bypass access controls and impersonate users.



4. Popular types of attack

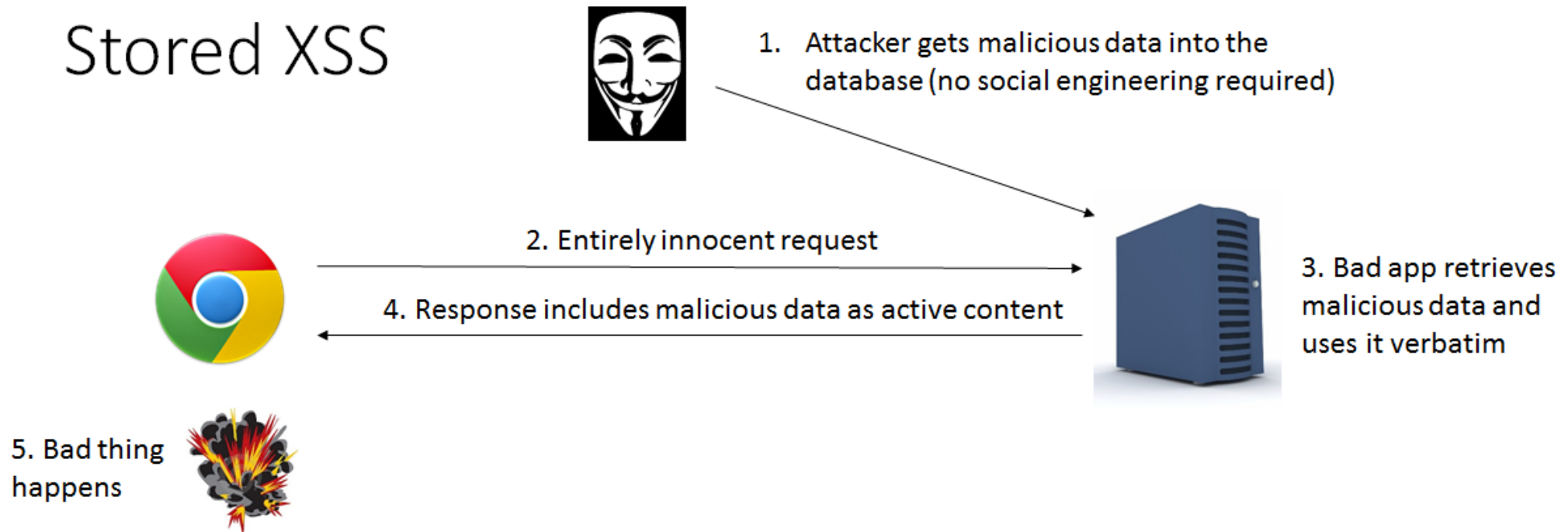
4.2. XSS:

- XSS usually occur when data enters a Web app through an untrusted source (most often a Web request) or dynamic content is sent to a Web user without being validated for malicious content.
- The malicious content often includes Javascript or any other code the browser can execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to a webpage controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

4. Popular types of attack

4.2. XSS:

- XSS attacks can be put into 3 categories: stored (also called persistent), reflected (also called non-persistent), or DOM-based.
- ❑ Stored XSS Attacks: The injected script is stored permanently on the target servers. The victim then retrieves this malicious script from the server when the browser sends a request for data.



4. Popular types of attack

4.2. XSS:

- XSS attacks can be put into 3 categories:

❑ Reflected XSS Attacks: A user is tricked into clicking a malicious link, submitting a specially crafted form, or browsing to a malicious site.



How Does Reflected XSS Work?

1. Attacker sends evil email



5. Attacker has full access to victim's account

4. Victim's browser now trusts the attacker's script is from bank.com

3. Vulnerable bank website takes data from request and includes in valid webpage



`http://bank.com?p1=">`



2. Victim clicks on link, sends request to vulnerable bank.com website

4. Popular types of attack

4.2. XSS:

- ❑ DOM-based XSS Attacks: The payload is executed as a result of modifying the DOM environment (in the victim's browser) used by the original client-side script. That is, the page itself does not change, but the client side code contained in the page runs in an unexpected manner because of the malicious modifications to the DOM environment.

Eg:

```
1 // An application uses some JavaScript to read the value from an input field and write that
  value to an element within the HTML:
2 const search = document.getElementById('search').value;
3 const results = document.getElementById('results');
4 results.innerHTML = 'You searched for: ' + search;
5
6 // 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:
7 You searched for: <img src=1 onerror='/* Bad stuff here ... */'>
```

4. Popular types of attack

4.2. XSS

- Preventing cross-site scripting is trivial in some cases but can be much harder depending on the complexity of the application and the ways it handles user-controllable data.
- In general, effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:

- ❑ 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.

4. Popular types of attack

4.3. SQL Injection

- SQL injection (SQLi) is a web security 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.
- 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.

4. Popular types of attack

4.3. SQL Injection

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



```
1 -- with x is value passed
2 SELECT * FROM products WHERE category = 'x' AND released = 1
3 -- x=Gifts
4 SELECT * FROM products WHERE category = 'Gifts' AND released = 1
5 -- x=Gifts'--
6 SELECT * FROM products WHERE category = 'Gifts'-- ' AND released = 1
7 -- x=Gifts' OR 1=1--
8 SELECT * FROM products WHERE category = 'Gifts' OR 1=1-- ' AND released = 1
```

4. Popular types of attack

4.3. SQL Injection

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



```
1 -- x and y are values passed
2 SELECT * FROM users WHERE username = 'x' AND password = 'y'
3 -- x=wiener, y=bluecheese
4 SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
5 -- x=administrator'--
6 SELECT * FROM users WHERE username = 'administrator' -- ' AND password = ''
```

4. Popular types of attack

4.3. SQL Injection

❑ UNION attacks: where you can retrieve data from different database tables.



```
1 -- x is value passed
2 SELECT name, description FROM products WHERE category = 'x'
3 -- x=Gifts
4 SELECT name, description FROM products WHERE category = 'Gifts'
5 -- x=Gifts' UNION SELECT username, password FROM users--
6 SELECT name, description FROM products WHERE category = 'Gifts'
   UNION SELECT username, password FROM users--
```

4. Popular types of attack

4.3. SQL Injection

- SQL injection can be detected 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.

