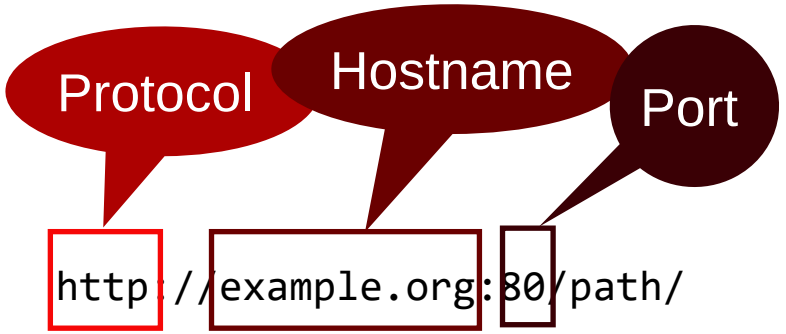
Stony Brook University

CSE 361: Web Security

Attacking the Same-Origin Policy

Nick Nikiforakis

Same-Origin Policy in Action



Attacking the Same-Origin Policy: DNS Rebinding

- Same-Origin Policy is based on the hostname
  - Hostname is not permanently bound to an IP address
- Attacker wants to gain access to network behind a firewall
- Idea: abuse Time-To-Live of DNS

DNS Rebinding - Concept

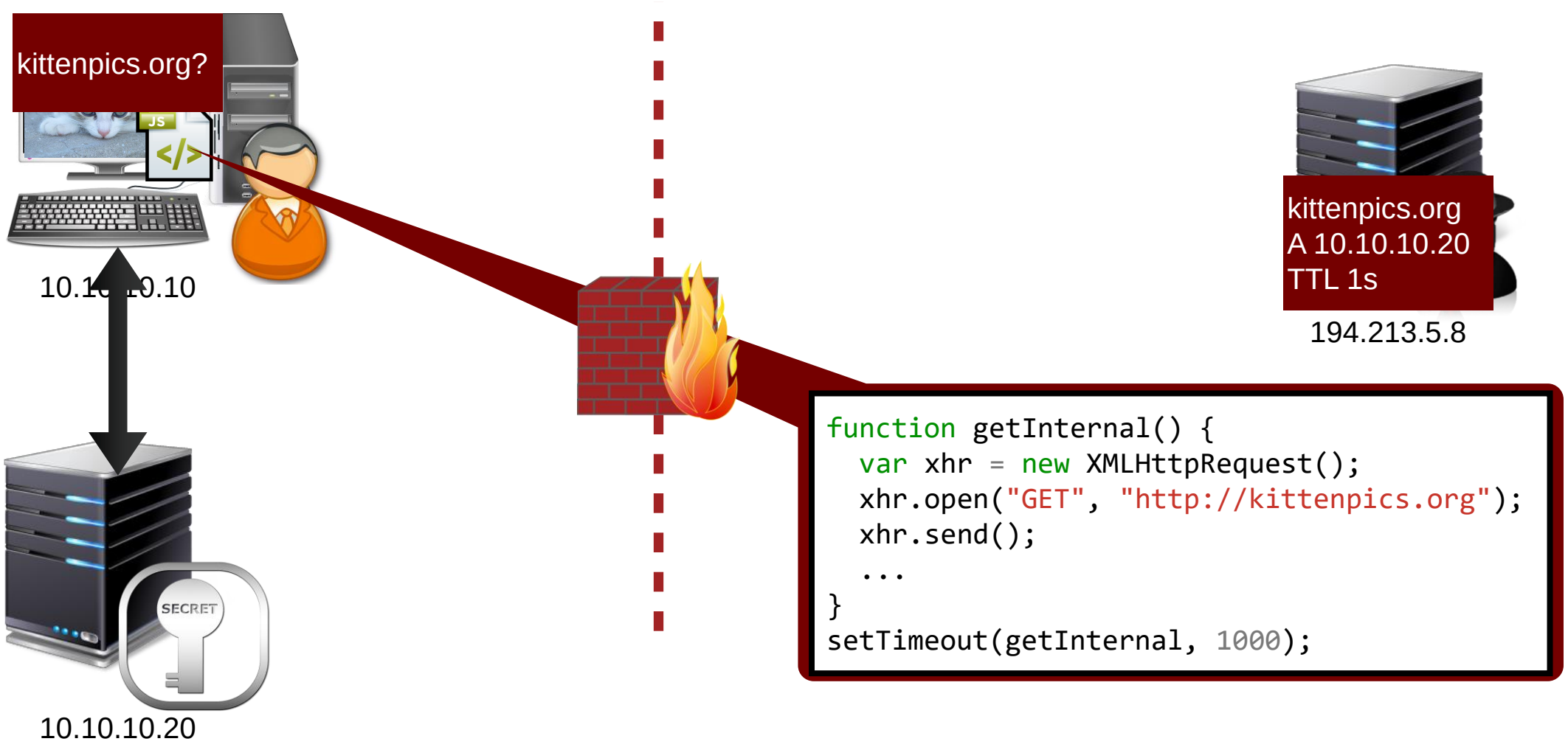
10.10.10.10

10.10.10.20

194.213.5.8

```
function getInternal() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://kittenpics.org");
  xhr.send();
  ...
}
setTimeout(getInternal, 1000);
```

DNS Rebinding - Concept



DNS Rebinding - A Brief History

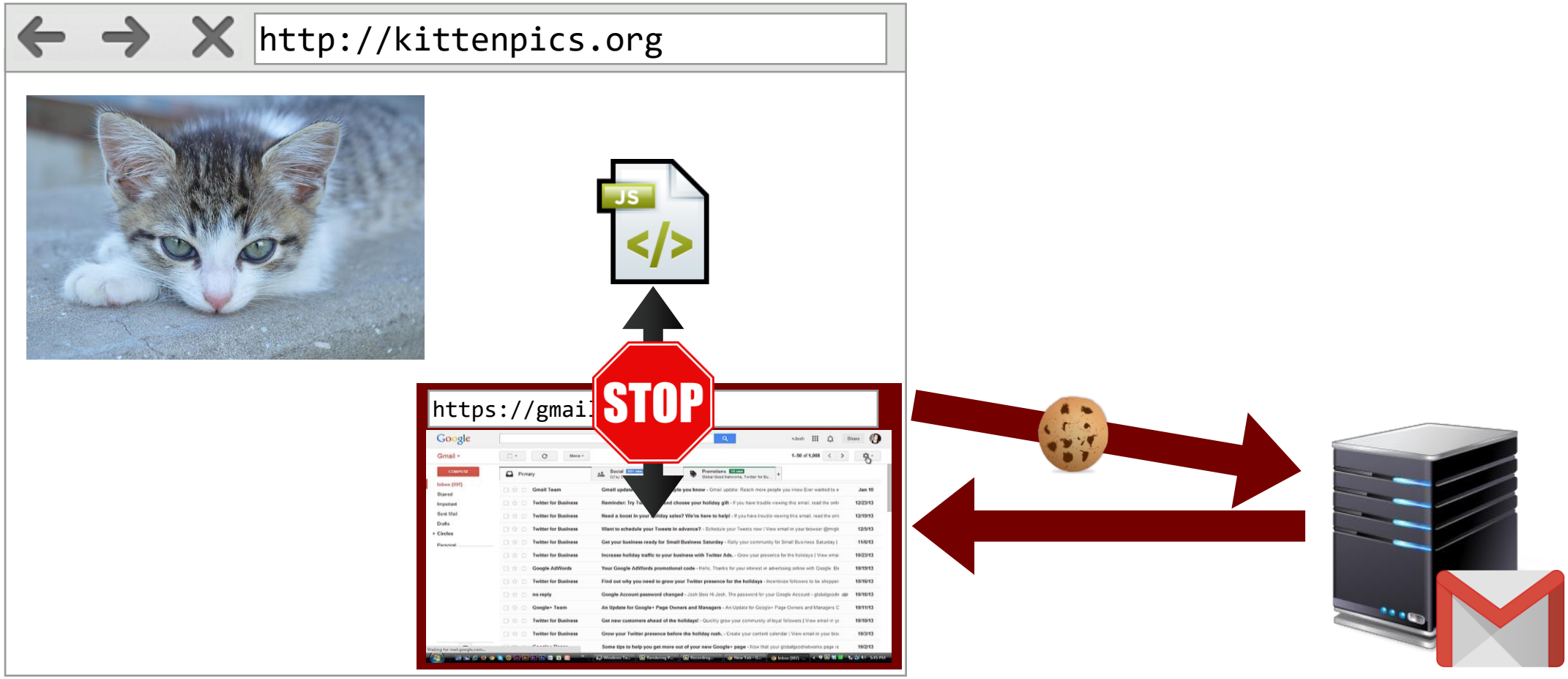
- 1996 Princeton Attack
  - Not real DNS Rebinding, rather two response (attacker and target), specifically targeted a bug in Java's VM
  - Mitigation: Java "pins" IP address used first
- 2002 Adam Megacz
  - Domain Relaxation, bind `attacker.org` to target, `sub.attacker.org` to own site (Recall the new domain relaxation rules?)
  - Mitigation: IE pins for 30 minutes, other browser do similar things
- 2006 Martin Johns
  - IE and Firefox dropped pin whenever a connection to the IP failed
- 2006 Kanatoko
  - same for Flash, but even with sockets
- 2013 Johns et al.
  - Using the HTML5 AppCache

Modern DNS rebinding

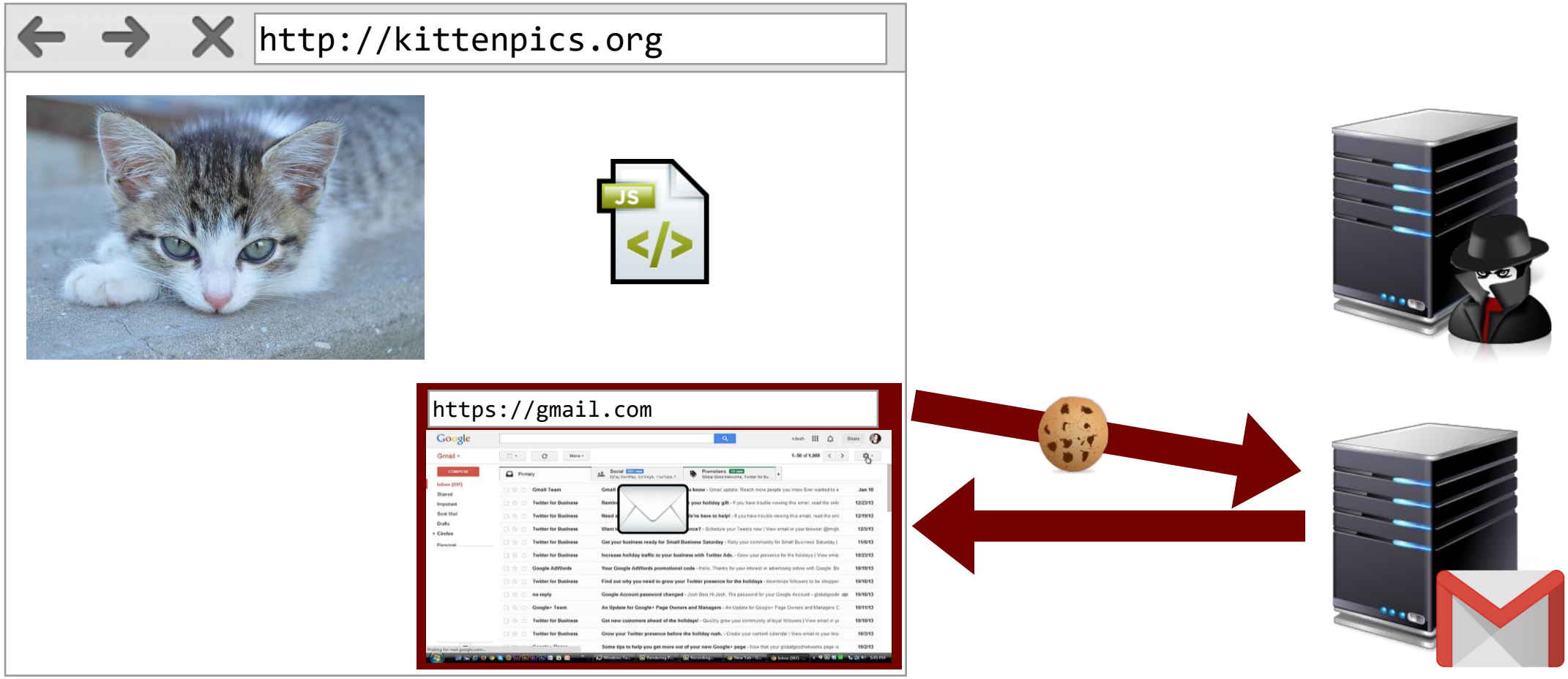
- Browsers only have a finite DNS cache size
  - Chrome 25 had 100, Chrome 26 1000, nowadays 1600
- Idea: evict existing entry by flooding the DNS cache
  - after that, have fun with the rebound IP

```
for(var i = 0; i < 1600; i++)
{
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://" + i + ".attacker.com");
}
```

Same-Origin Policy in Action



Bypassing the SOP with Code Injection



Cross-Site Scripting

- Attacker can inject his own **script** into another site (**cross-site**)
  - actually, might have to inject HTML markup
  - ... which contains JavaScript code
- Injected code runs in origin of vulnerable page
  - can do whatever legitimate code can do
  - can modify page to attacker's liking
- Has roughly two orthogonal dimensions
  - Location of vulnerable code (server or client)
  - Persistence of attack payload (reflected or persistent)

A short history of Cross-Site Scripting

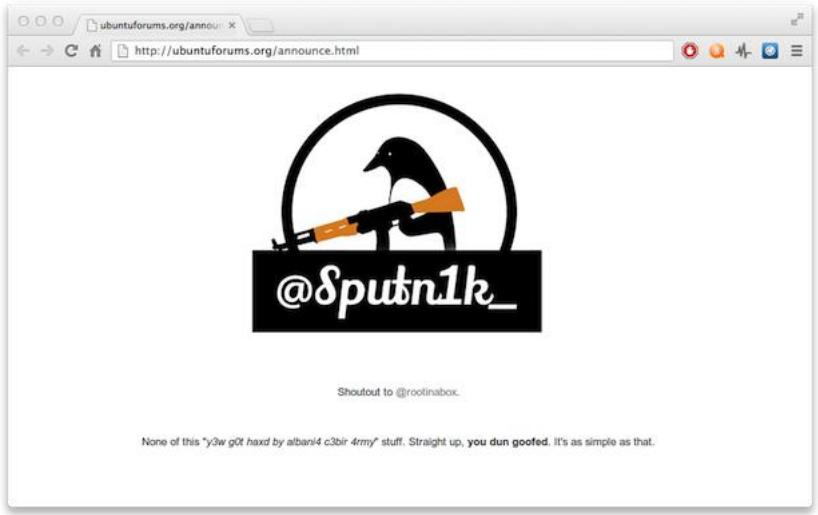
- First discovered in 1999
  - December 1999 by Microsoft
  - (allegedly) November 1999 by people at American Express
  - **Reflected** Server-Side Cross-Site Scripting in several 404 pages
- Amit Klein coined the term "DOM-based Cross-Site Scripting" in 2005
  - referring to the DOM as the part which would be abused to inject code
  - .. does not really cover the eval case
  - we refer to this as **Client-Side Cross-Site Scripting**

Impact of Cross-Site Scripting vulnerabilities

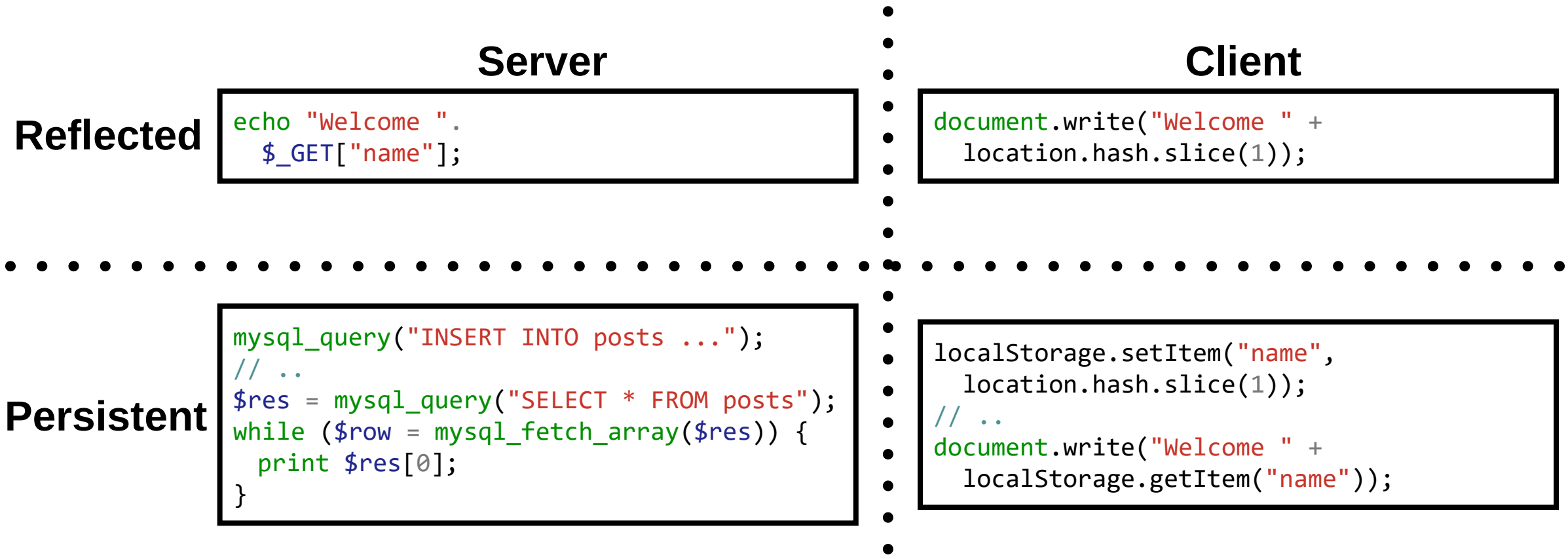
- JavaScript execution allows attacker to pretend to be
  - ... user towards the server (e.g., posting content in social network)
  - ... server towards the user (e.g., by modifying the look of a page)
- Obvious first target: reading cookies (**session hijacking**)
  - somewhat mitigated by HTTPOnly cookies
- Other "use cases" include
  - attacking browser-based password managers
  - setting cookies

Real-World XSS: 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 (yes, cookies....)
- Given elevated privileges, the attacker could upload PHP shell
  - eventually dumped the users database and left defacement on main page



Dimensions of Cross-Site Scripting



Reflected Server-Side Cross-Site Scripting

1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)

Reflected Server-Side Cross-Site Scripting

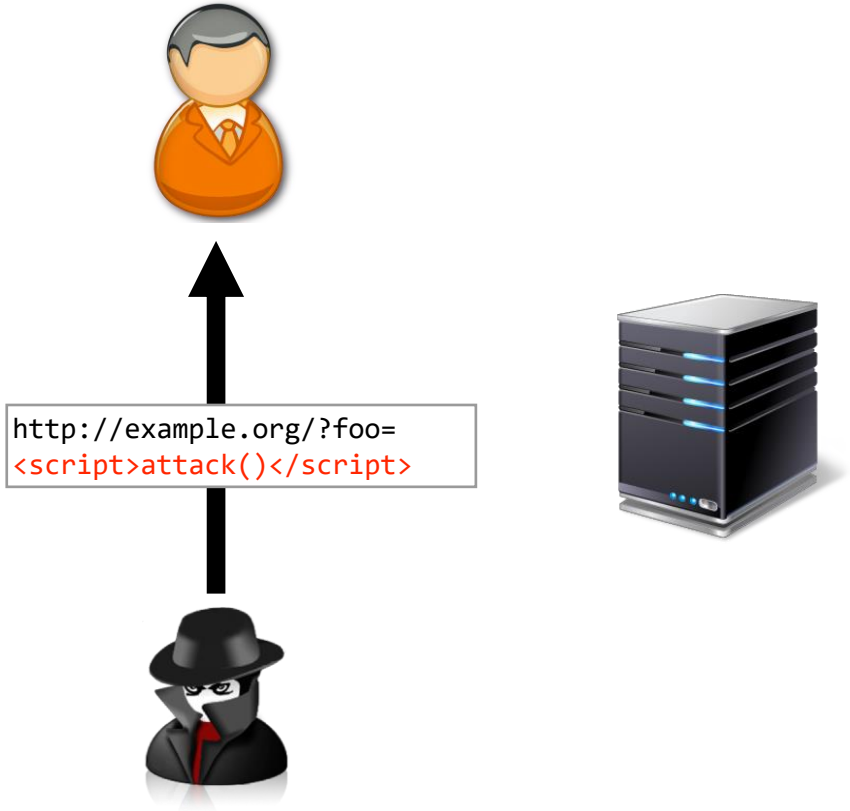
1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)
2. Once reflected potentially dangerous content is found, injects complete script

Reflected Server-Side Cross-Site Scripting

1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)
2. Once reflected potentially dangerous content is found, injects complete script
3. Crafts specific attack payload, e.g., to steal cookie

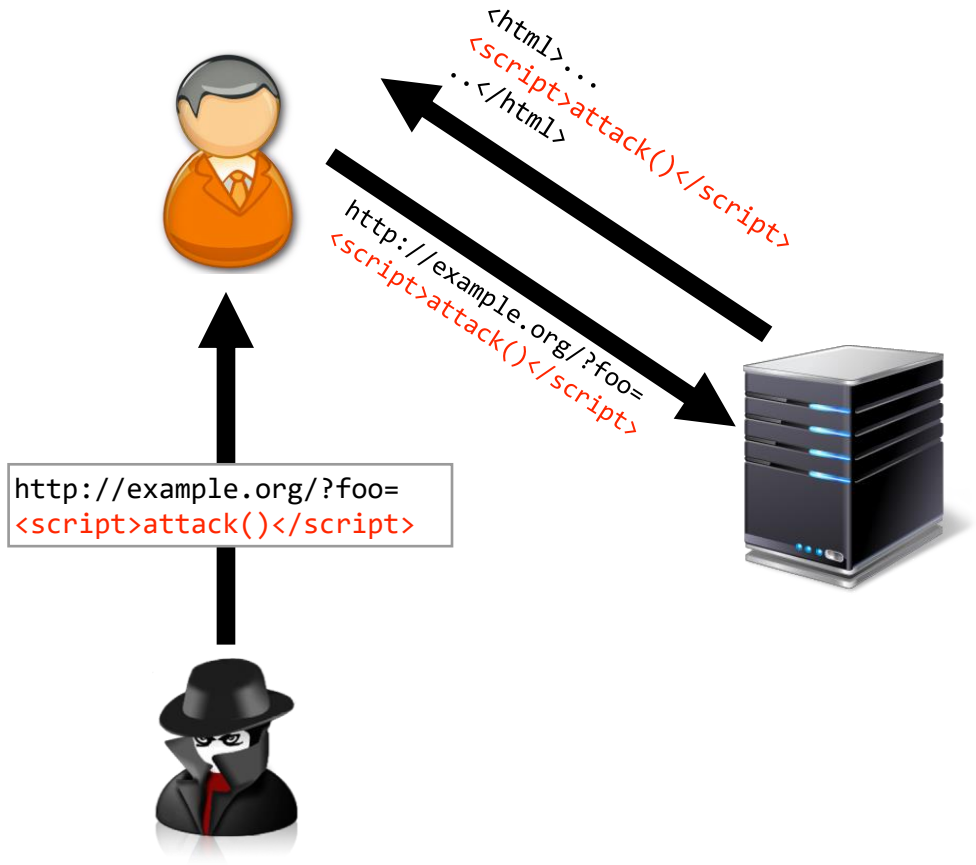
Reflected Server-Side Cross-Site Scripting

1. Attacker tricks victim into visiting link
   - Sends email with link
   - Embeds iframe to vulnerable site on his own domain



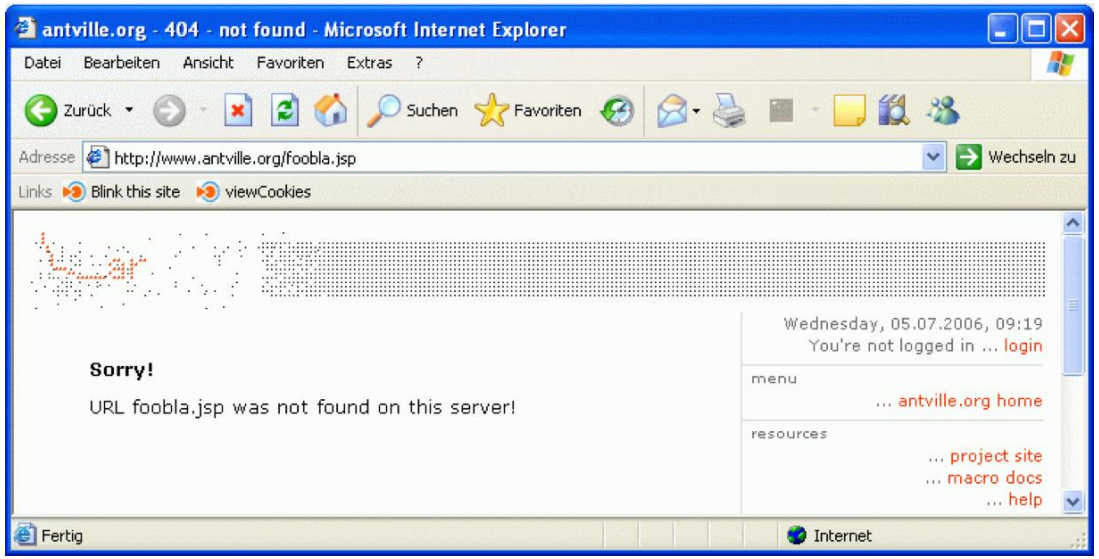
Reflected Server-Side Cross-Site Scripting

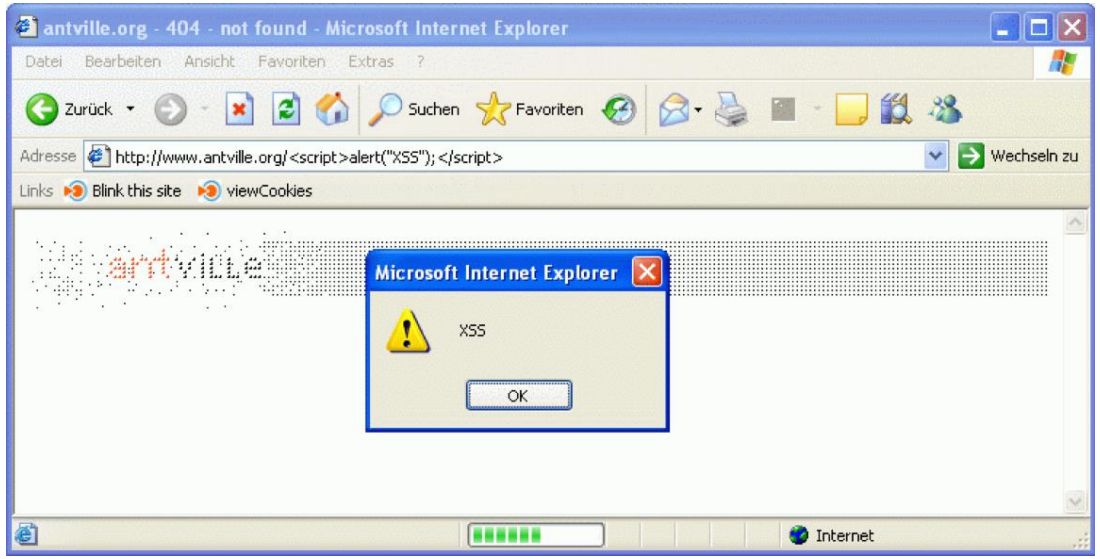
1. Attacker tricks victim into visiting link
   - Sends email with link
   - Embeds iframe to vulnerable site on his own domain
- Malicious payload is reflected from server
   - May interact with server as the user
   - May leak sensitive information (e.g., cookie) to the attacker



Reflected Server-Side Cross-Site Scripting: Examples

- 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';





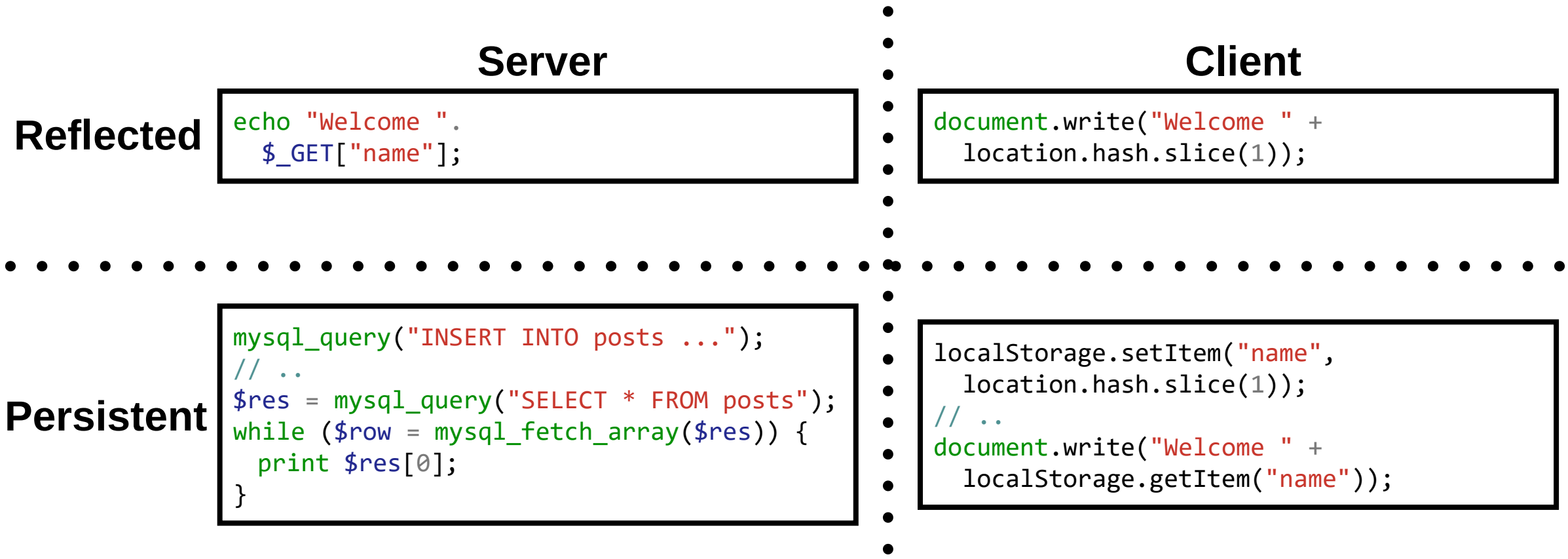
Example: exploiting reflected server-side XSS

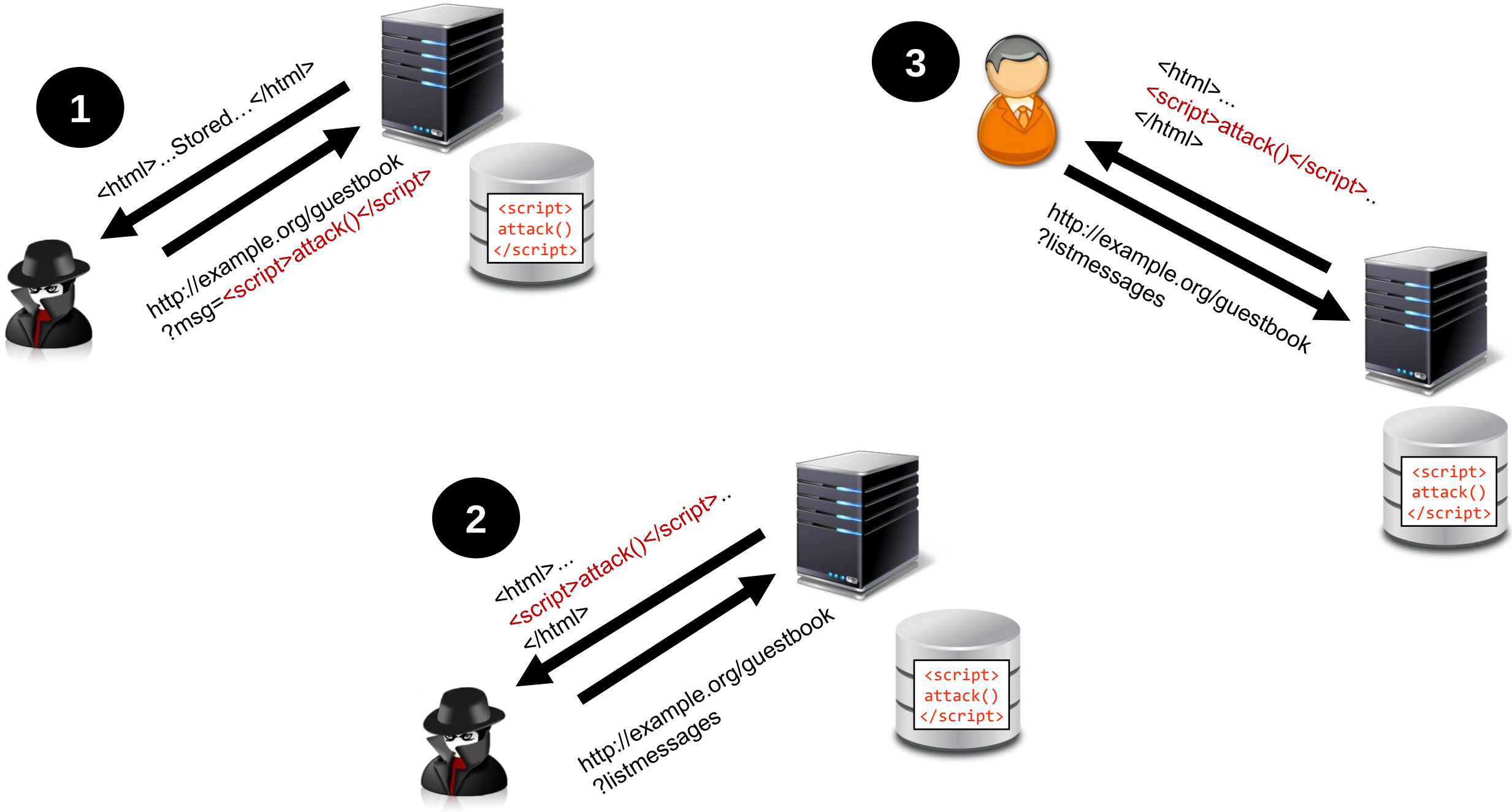
```php
<?php
// load avatar
echo "<img src='//avatar.com/img.php?user=" . $_GET["user"] . "'>";
?>
```

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

- <img src='//avatar.com/img.php?user='><script>alert(1)</script>'>

Dimensions of Cross-Site Scripting



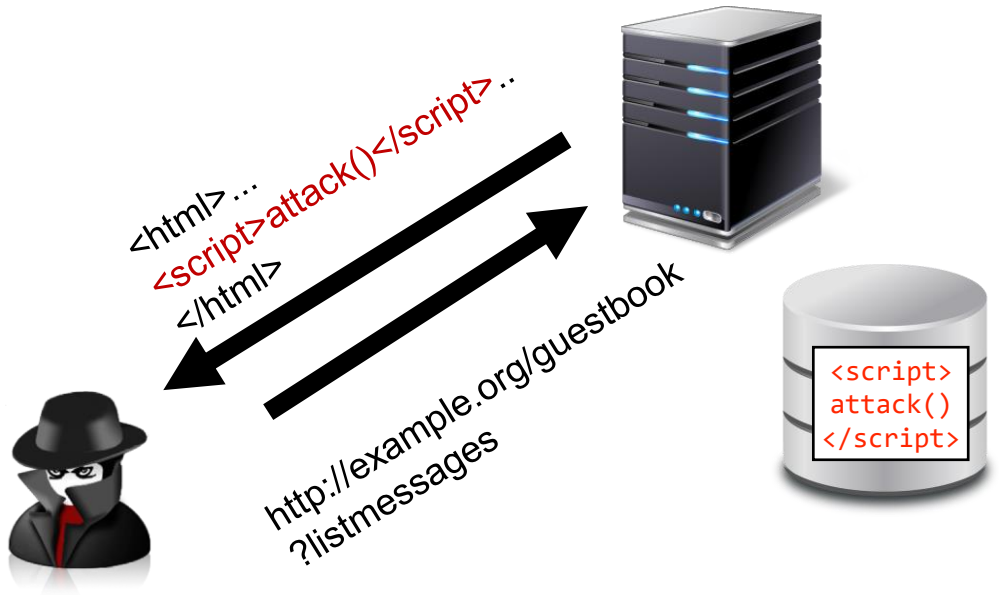

1
<html>…Stored…</html>
http://example.org/guestbook
?msg=<script>attack()</script>
<script>
attack()
</script>
2
<html>…
<script>attack()</script>..
</html>
http://example.org/guestbook
?listmessages
<script>
attack()
</script>
3
<html>…
<script>attack()</script>..
</html>
http://example.org/guestbook
?listmessages
<script>
attack()
</script>

Persistent Server-Side Cross-Site Scripting

1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)
2. Data is not immediately reflected, rather stored in database

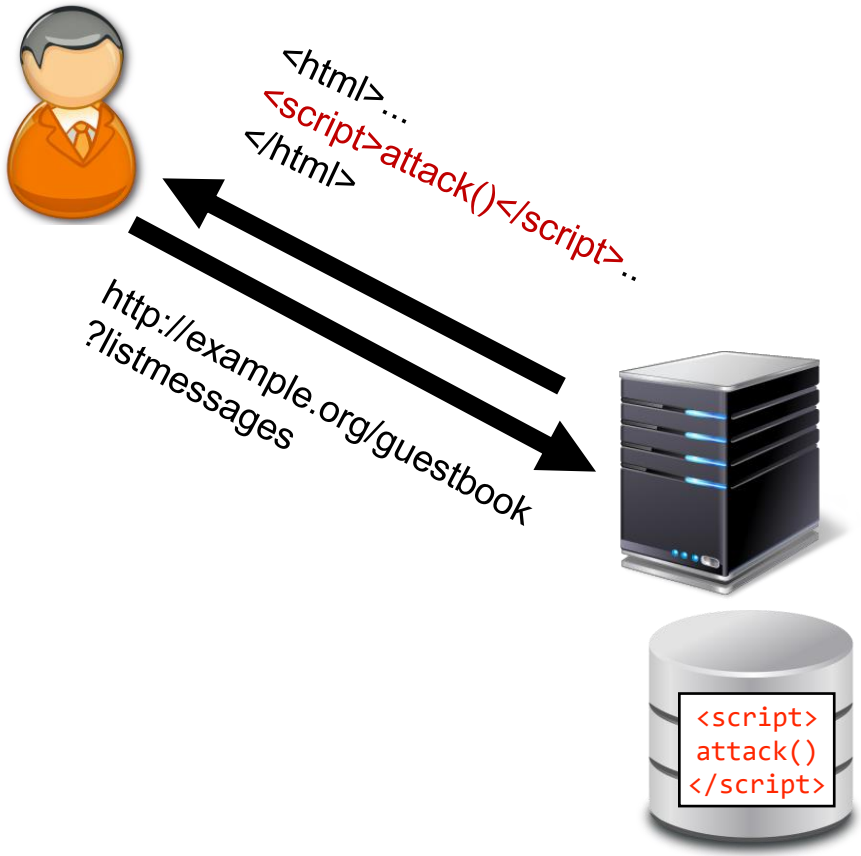
Persistent Server-Side Cross-Site Scripting

1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)
2. Data is not immediately reflected, rather stored in database
3. Attacker checks the stored entry



Persistent Server-Side Cross-Site Scripting

1. Attacker probes server for vulnerabilities
   - Injecting markup into request parameters (in case data is used within HTML)
   - Injecting JavaScript in request parameters (in case data is used within script)
2. Data is not immediately reflected, rather stored in database
3. Attacker checks the stored entry
4. **Every user** of the site is attacked

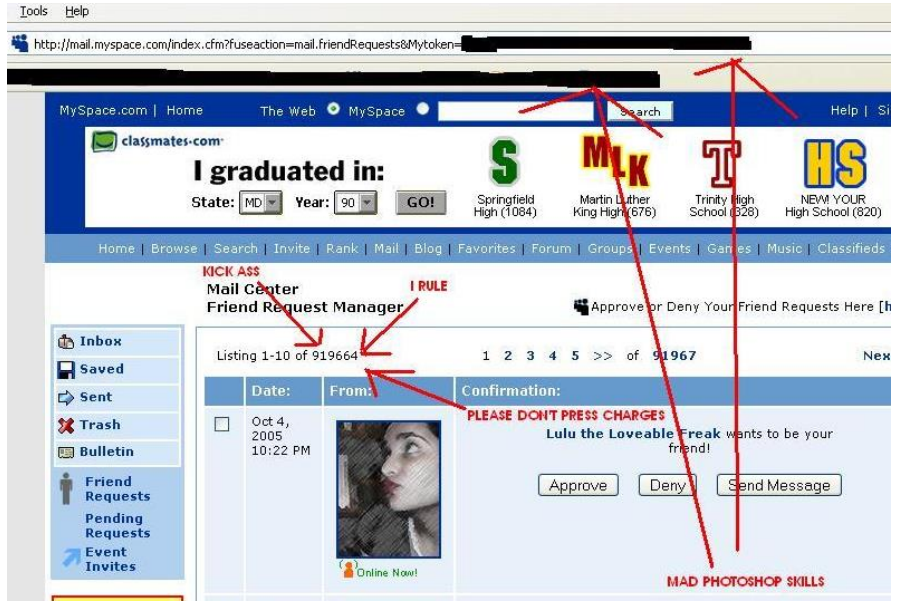


Persistent Server-Side Cross-Site Scripting: Examples

- Anything that stores data
  - Guestbooks
  - Forums
  - Profile pages on social media
- More interesting vectors
  - Description of books on Amazon
  - Abstract of a book on Amazon
    - scanned the XSS payload with OCR
- ....

Persistent Server-Side Cross-Site Scripting: MySpace worm

- MySpace allowed certain HTML tags in profiles
  - tried to block others
- Samy Kamkar (April 2005) found bypass
  - `<div id="mycode" expr="alert('hah!')" style="background:url('java script:eval(document.all.mycode.expr)')">`
- Attack payload added Samy as a friend
  - According to Samy, goal was to "befriend girls"
- and updated the profile of the infected victim
  - in turn, all friends could be infected
- over 1,000,000 friends (over 3% of MySpace) within 20 hours



Preventing Server-Side Cross-Site Scripting

- Option 1: Input Validation/Sanitization
- Check input against list of allowed/expected characters
  - Is this a number? Is this an email?
- Can only be considered first line of defense
  - Usage of data might not be known at that point
  - Hard to get right, for the general case
- (bad) alternative: removing unwanted elements
  - Known as blacklisting/blocklisting
  - e.g., all script tags
  - simple replace does not suffice: `<scr<script>ipt>`

```php
foreach ($_REQUEST as $key => value) {
  $_REQUEST[$key] = preg_replace("[^0-9a-zA-Z]",
                                 "", $value);
}
// ....
$username = base64_decode($_REQUEST["user"]);
```

Preventing Server-Side Cross-Site Scripting

- Option 2: Output Encoding
- When using the data, encode it
  - depending on context, different encoders might be necessary

HTML Encoding **PHP**

```php
<?php
 function noHTML($input, $encoding = 'UTF-8'){
  return htmlentities($input, ENT_QUOTES | ENT_HTML5, $encoding)
 }
 ...
 echo '<div> You searched for ' . noHTML($_GET['q']) . ' </div>';
?>
```

Preventing Server-Side Cross-Site Scripting

- Option 2: Output Encoding
- When using the data, encode it
  - depending on context, different encoders might be necessary

URL Encoding

PHP

```php
<?php

function sanitizeParam(){
 return urlencode($param);
}

echo '<a href="https://example.com/article?input="' . sanitizeParam($_GET['q']) . '">...</a>';

?>
```

Preventing Server-Side Cross-Site Scripting: Best Practices

- Avoid creating your own filters
  - frameworks typically have (hopefully) context-aware filters
  - read the exact manual of functions if you use them (e.g., htmlentities)
- Do not allow user-provided markup
  - recall MySpace?
  - if need be, use well-defined alternative mark-up languages
    - BBCode, Markdown, ...
- Disable error reporting to the Web frontend
  - among other reasons: stack trace might contain unencoded parameters...

Dimensions of Cross-Site Scripting

	Server	Client
Reflected	`echo "Welcome ".` `  $_GET["name"];`	`document.write("Welcome " +` `  location.hash.slice(1));`

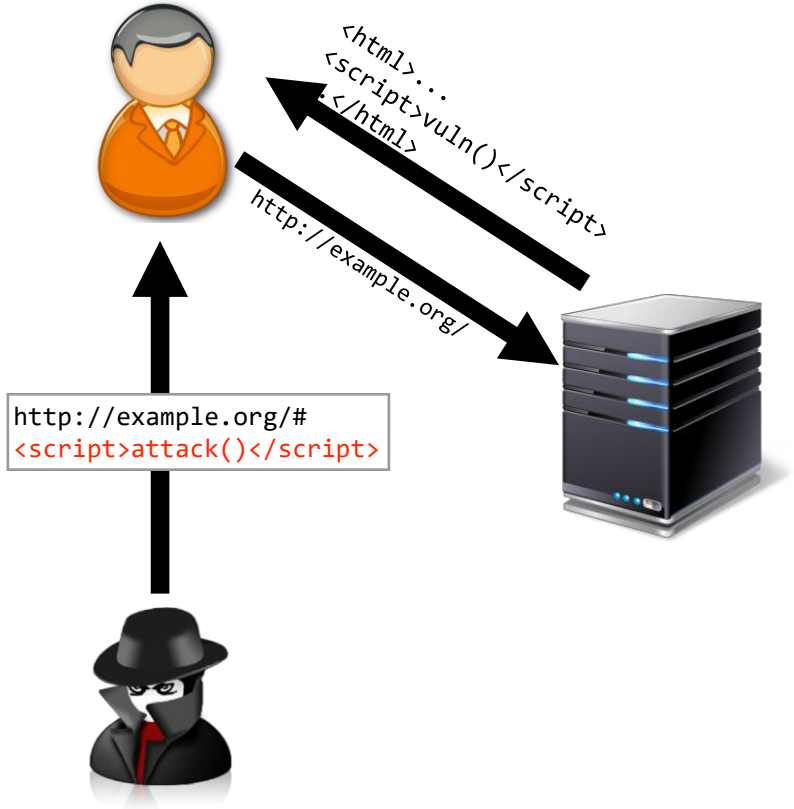
Persistent	`mysql_query("INSERT INTO posts ...");`
	`// ..`
	`$res = mysql_query("SELECT * FROM posts");`
	`while ($row = mysql_fetch_array($res)) {`
	`  print $res[0];`
	`}`
		`localStorage.setItem("name",`
		`  location.hash.slice(1));`
		`// ..`
		`document.write("Welcome " +`
		`  localStorage.getItem("name"));`

Reflected Client-Side Cross-Site Scripting

1. Attacker analyzes client-side JavaScript code for vulnerabilities
   - searches for unfiltered usage of attacker-controllable data (e.g., the URL)
   - such data may be contained in URL fragment
     - Important: not sent to the server

Reflected Client-Side Cross-Site Scripting

1. Attacker analyzes client-side JavaScript code for vulnerabilities
   - searches for unfiltered usage of attacker-controllable data (e.g., the URL)
   - such data may be contained in URL fragment
     - Important: not sent to the server
2. Attacker tricks victim into visiting URL with payload, e.g., in fragment
   - vulnerable JavaScript is delivered to client
   - exploit triggered without payload being sent to server (if in fragment)



Relevant APIs for Client-Side Cross-Site Scripting

- document.write, document.writeln
  - Can write new script tags which will be executed
- eval, setTimeout, setInterval
  - Directly executes JavaScript code
- innerHTML, outerHTML
  - will not execute script elements, but event handlers work
  -

Example: exploiting reflected client-side XSS

```
// ensure that things are always unencoded, as browsers differ in their behaviour
var hash = unescape(location.hash);

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

- **Important:** iframe is one of very few elements that needs to be closed
  - anything between iframe tags is shown only if browser does not support framing
- Exploit payload:
  - Close opening iframe tag: `'>`
  - Close iframe: `</iframe>`
  - Add payload: `<script>alert(1)</script>`
- Exploit URL:

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

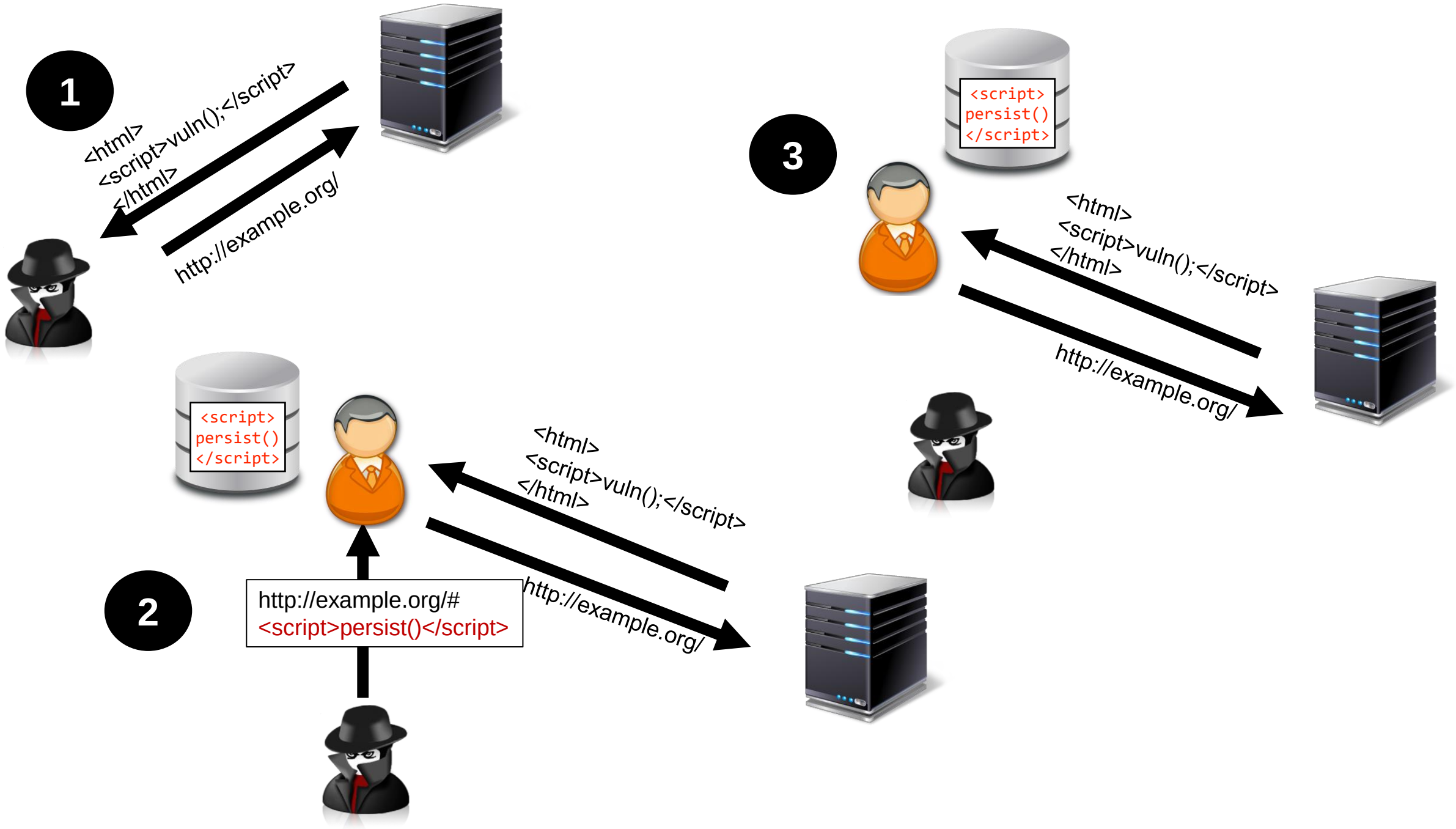
Dimensions of Cross-Site Scripting

	Server	Client
Reflected	`echo "Welcome ".` `  $_GET["name"];`	`document.write("Welcome " +` `  location.hash.slice(1));`

Persistent	`mysql_query("INSERT INTO posts ...");`
	`// ..`
	`$res = mysql_query("SELECT * FROM posts");`
	`while ($row = mysql_fetch_array($res)) {`
	`  print $res[0];`
	`}`
		`localStorage.setItem("name",`
		`  location.hash.slice(1));`
		`// ..`
		`document.write("Welcome " +`
		`  localStorage.getItem("name"));`

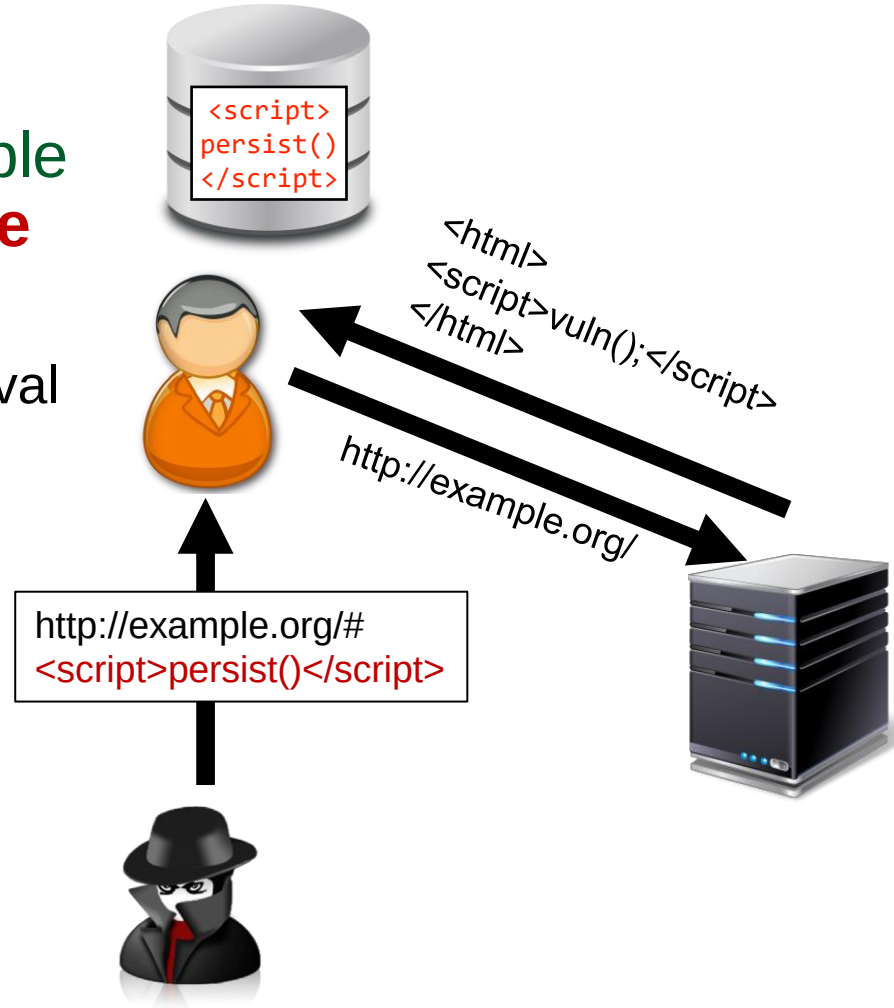
Persistent Client-Side Cross-Site Scripting

1. Attacker analyzes client-side JavaScript code for vulnerabilities
   - searches for unfiltered usage of attacker-controllable data (e.g., the URL) **flowing to persistent storage**
   - Searches for execution of persistent storage
     - Example: cookie stores first visited URL, is used in later eval statement


1
<html>
<script>vuln();</script>
</html>
http://example.org/
2
http://example.org/#
<script>persist()</script>
<script>
persist()
</script>
<html>
<script>vuln();</script>
</html>
http://example.org/
3
<script>
persist()
</script>
<html>
<script>vuln();</script>
</html>
http://example.org/

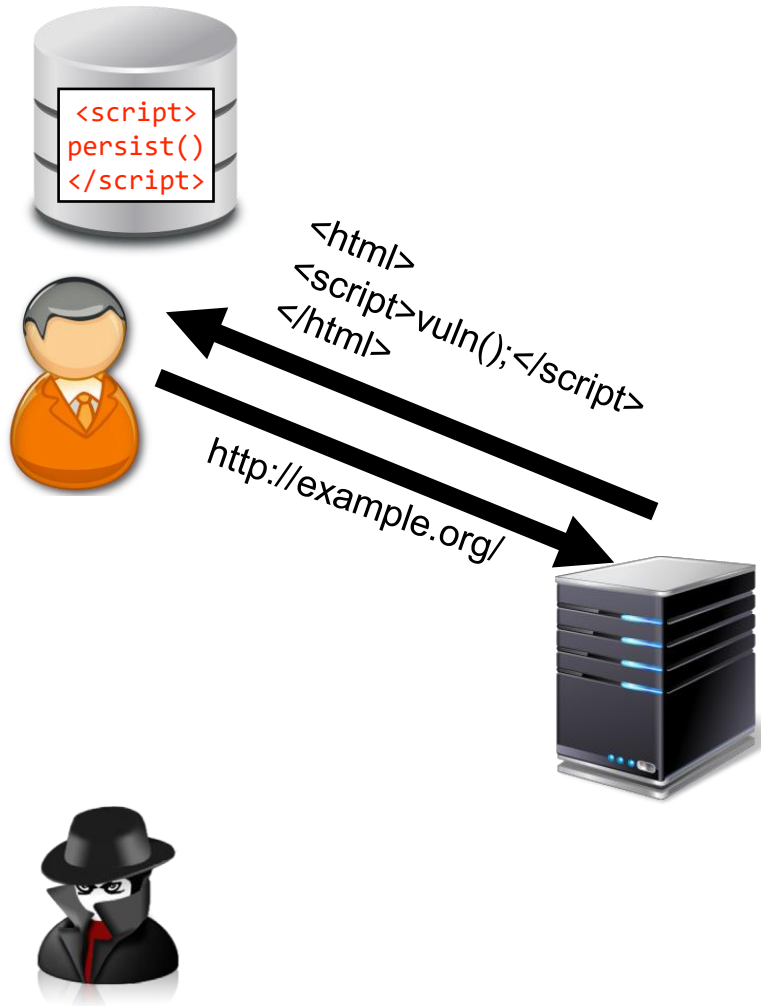
Persistent Client-Side Cross-Site Scripting

1. Attacker analyzes client-side JavaScript code for vulnerabilities
   - searches for unfiltered usage of attacker-controllable data (e.g., the URL) **flowing to persistent storage**
   - Searches for execution of persistent storage
     - Example: cookie stores first visited URL, is used in later eval statement
2. Attacker tricks victim into visiting URL with payload, e.g., in fragment
   - data-persisting JavaScript is delivered to client
   - exploit payload is stored in persistent storage
   - Alternatively: exploit other type of XSS to gain permanent foothold in the client's browser



Persistent Client-Side Cross-Site Scripting

1. Attacker analyzes client-side JavaScript code for vulnerabilities
2. Attacker tricks victim into visiting URL with payload, e.g., in fragment
3. On every page visit, payload is extracted from persistent storage
   - flow from storage to execution sink
   - malicious payload is executed



Sources for Persistent Client-Side Cross-Site Scripting

- **Cookies**
  - bound to eTLD+1 or hostname
  - limited character set
    - e.g., no semicolon
    - 4,096 chars at most
- **Web Storage**
  - bound to an origin
  - Local Storage
  - Session Storage
- **IndexedDB**
  - bound to origin

- HTML Markup

```
element.innerHTML = "foobar";
```

- JavaScript

```
eval("x = 'foobar'");
```

- Script source

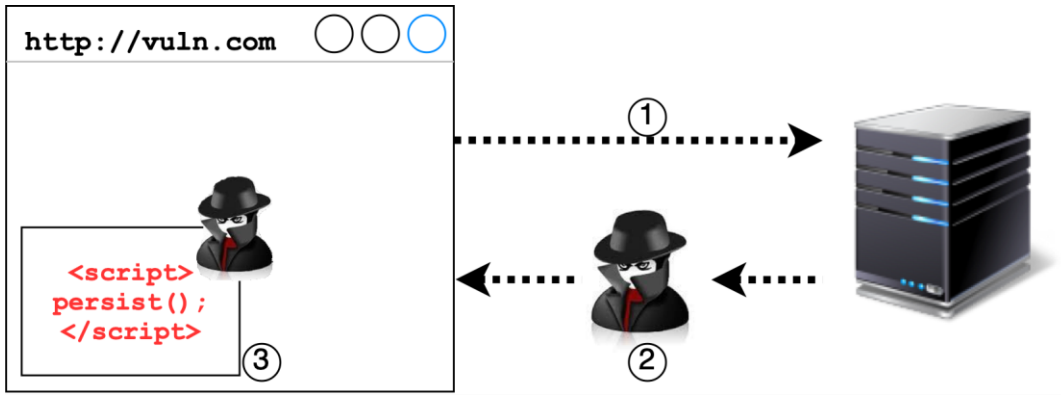
```
var script =
document.createElement("script");
script.src="//foobar.script.com";
document.body.appendChild(script)
```

Interlude: HTTP Strict Transport Security

- HTTP header (Strict-Transport-Security) sent by server
  - only valid if sent via HTTPS
  - `Strict-Transport-Security: max-age=<expiry in seconds>`
    - `includeSubDomains`: header is valid for all subdomains
    - `preload`: allows for inclusion in preload list
  - ensures that site cannot be loaded via HTTP until expiry is reached
- Domains can be preloaded in browsers
  - HSTS preload list (https://hstspreload.org/)
  - only possible with at least 18 weeks max-age, includeSubDomains and automatic redirect from HTTP

Persistent Client-Side Cross-Site Scripting: Attacker Models

- Requirement for successful attack: persisted malicious payload
  - extracted on every page load; single "infection" is sufficient
- Attacker Model #1: **Network Attacker**
  - can modify unencrypted connections
  - cannot get arbitrary TLS certificates
- Capabilities
  - Cookies: set cookies for any domain without HSTS
    - HSTS must use includeSubDomains
  - Local Storage: inject items on HTTP sites only



Persistent Client-Side Cross-Site Scripting: Attacker Models

- Attacker Model #2: **Web Attacker**
  - can force victim's browser to visit any URL
- Attack Vector #1: Abuse existing XSS flaw
  - allows to inject data into origin (Storage) or domain (cookies)
  - HTTPS does not help at all
- Attack Vector #2: Abuse flows into storage
  - requires a flow into storage item
  - important: same storage item must be later used
  - hard to find in practice

Preventing Client-Side Cross-Site Scripting

- Problems originate from use of insecure APIs
  - **eval, document.write, innerHTML**
  - and the use of user-provided input in them
- Depending on the context, functionally equivalent APIs exist
  - document.createElement, element.innerText
  - JSON.parse

```
function writeURLInsecure() {
  document.write("<p>The current URL is: "
    + location.href + "</p>");
}
```

```
function writeURLSecure() {
  var p = document.createElement("p");
  p.innerText = "The current URL is: " + location.href;
  document.write(p.outerHTML);
}
```

Preventing Client-Side Cross-Site Scripting

```
function loadAdvertisementInsecure() {
  document.write("<script src='http://ad.com/?referrer=" + location.href + "'></script>");
}
```

```
function loadAdvertisementSecure() {
  var script = document.createElement("script");
  script.src = 'http://ad.com/?referrer=' + location.href;
  document.body.appendChild(script);
}
```

- element.src ensures that attacker-controllable data can only be in src attribute

Preventing Client-Side Cross-Site Scripting

```
function parseJSONInsecure(json) {
  var object = eval(json);
}
```

```
function parseJSONSecure(json) {
  var object = JSON.parse(json);
}
```

```
function registerGlobalInsecure(key, value) {
  eval(key + " = '" + value + "'");
}
```

```
function registerGlobalSecure(key, value) {
  // check if key is something you want to
  // have overwritten in the first place…
  window[key] = value;
}
```

- Depending on the desired use, either
  - use JSON.parse
  - use object[key] = value notion

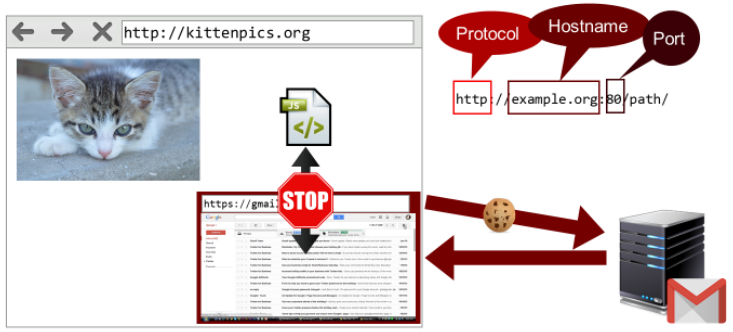
Dimensions of Cross-Site Scripting

	Server	Client
Reflected	`echo "Welcome ".` `  $_GET["name"];`	`document.write("Welcome " +` `  location.hash.slice(1));`

Persistent	`mysql_query("INSERT INTO posts ...");`
	`// ..`
	`$res = mysql_query("SELECT * FROM posts");`
	`while ($row = mysql_fetch_array($res)) {`
	`  print $res[0];`
	`}`
		`localStorage.setItem("name",`
		`  location.hash.slice(1));`
		`// ..`
		`document.write("Welcome " +`
		`  localStorage.getItem("name"));`

Summary

Same-Origin Policy in Action

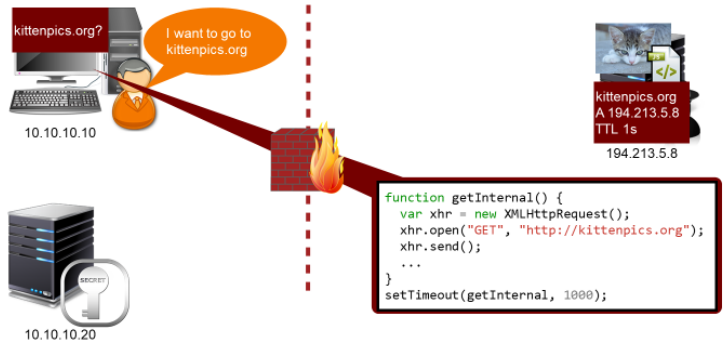


Dimensions of Cross-Site Scripting

	Server	Client
Reflected	`echo "Welcome ". $_GET["name"];`	`document.write("Welcome " + location.hash.slice(1));`

Persistent	`mysql_query("INSERT INTO posts ...");`
	`// ..`
	`$res = mysql_query("SELECT * FROM posts");`
	`while ($row = mysql_fetch_array($res)) {`
	`  print $res[0];`
	`}`
		`localStorage.setItem("name", location.hash.slice(1));`
		`// ..`
		`document.write("Welcome " + localStorage.getItem("name"));`

DNS Rebinding - Concept



Preventing Server-Side Cross-Site Scripting

- Option 2: Output Encoding
- When using the data, encode it
  - depending on context, different encoders might be necessary

HTML Encoding **PHP**

```php
<?php
 function noHTML($input, $encoding = 'UTF-8'){
  return htmlentities($input, ENT_QUOTES | ENT_HTML5, $encoding)
 }
 ...
 echo '<div> You searched for ' . noHTML($_GET['q']) . ' </div>';
?>
```

Credits

- Original slide deck by Ben Stock
- Modified by Nick Nikiforakis