

IERG 4210

Web Programming & Security

Tutorial 6

CHEN Lin

(Part of slides are modified from the former TA Fan YANG)

Outline

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

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

- A SQL injection attack happens when a user injects malicious bits of SQL into your database queries.

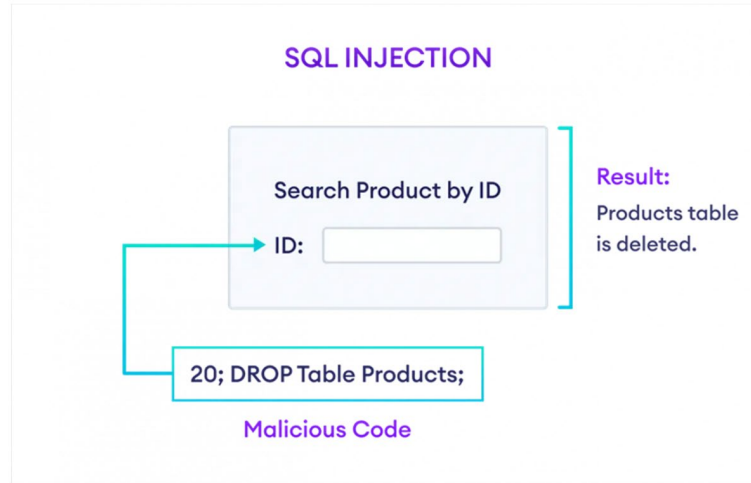


Illustration of a successful SQL Injection attack through the user-facing input

SQL injection

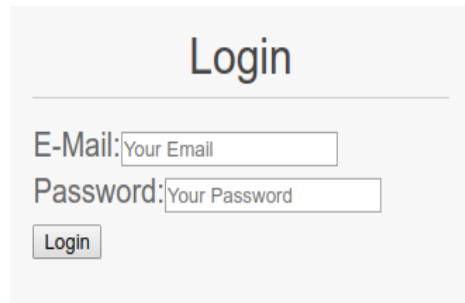
Normal SQL query:

```
SELECT * FROM users WHERE username='alice' and password='secret'
```

SQL injection:

```
SELECT * FROM users WHERE username='alice' and password=''; drop table  
user;
```

Delete sensitive data!



The image shows a login form with the title "Login". Below the title is a horizontal line. There are two input fields: "E-Mail:" followed by a text box containing "Your Email", and "Password:" followed by a text box containing "Your Password". Below these fields is a button labeled "Login".

SQL injection

Normal SQL query:

```
SELECT * FROM users WHERE username='alice' and password='secret'
```

SQL injection:

```
SELECT * FROM users WHERE username='alice'--" and password='any'
```

```
SELECT * FROM users WHERE username=' alice' and password='any' or '1'='1'
```

The above two statements are equivalent to

```
SELECT * FROM users WHERE username='alice'
```

Log in without password!

SQL injection - Defense

Use prepared statements and parameterized queries.

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.
- Use Placeholder or parametrized queries: the malicious SQL will be escaped and treated as a raw string, not as actual SQL code.

SQL injection - Defense



The query only needs to be parsed (or prepared) once, but can be executed multiple times with the same or different parameters.

The prepared statements use fewer resources and thus run faster.

```
String username = request.getParameter("j_username");
String passwd = request.getParameter("j_password");
String query = "select * from t_sysuser where username=? and passwd=?";
PreparedStatement stmt = con.prepareStatement(query);
stmt.setString(1, username);
stmt.setString(1, passwd);
ResultSet rs=stmt.executeQuery();
```

All contents entered in the input field is considered as raw string.

SQL injection - Defense

Example: `username` alice `password` any' or 1=1

SQL injection

```
SELECT * FROM users WHERE username='alice' and password='any' or 1=1;
```

Statement object uses part of the password as a query condition when executing the sql statement

Use prepared statements and parameterized queries.

```
SELECT * FROM users WHERE username= 'alice' and passwd='any' 'or 1=1'
```

All contents entered in the password field is treated as a raw string, not as actual SQL code.

SQL injection - Defense

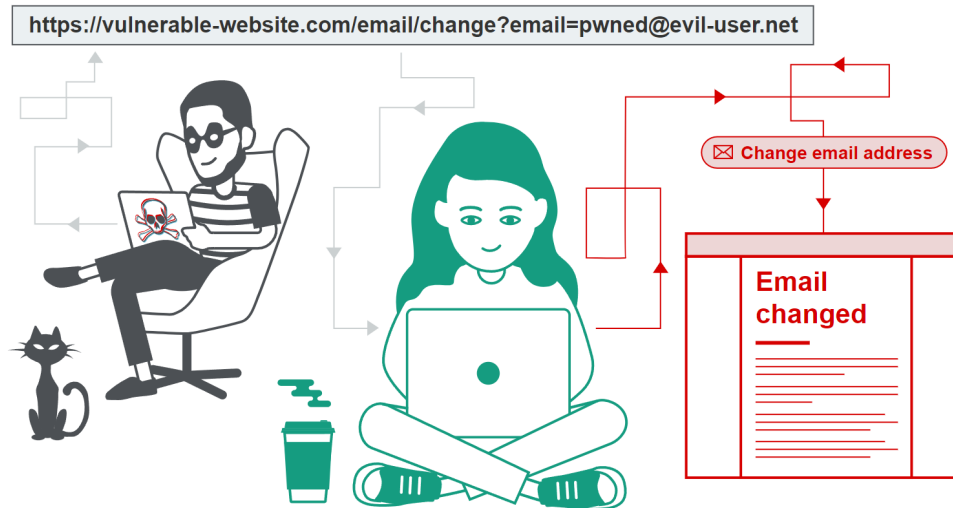
- Avoid the usage of dynamic SQL query
- Use strict input sanitization
 - e.g., replace or filter single quotes ('), double dashes (--), SELECT, UNION and other query keywords
- Check input data type
 - e.g., only integer allowed, regular expression.
- Use security control interfaces.
 - Reference: <https://owasp.org/www-project-enterprise-security-api/>

Client Side Security

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

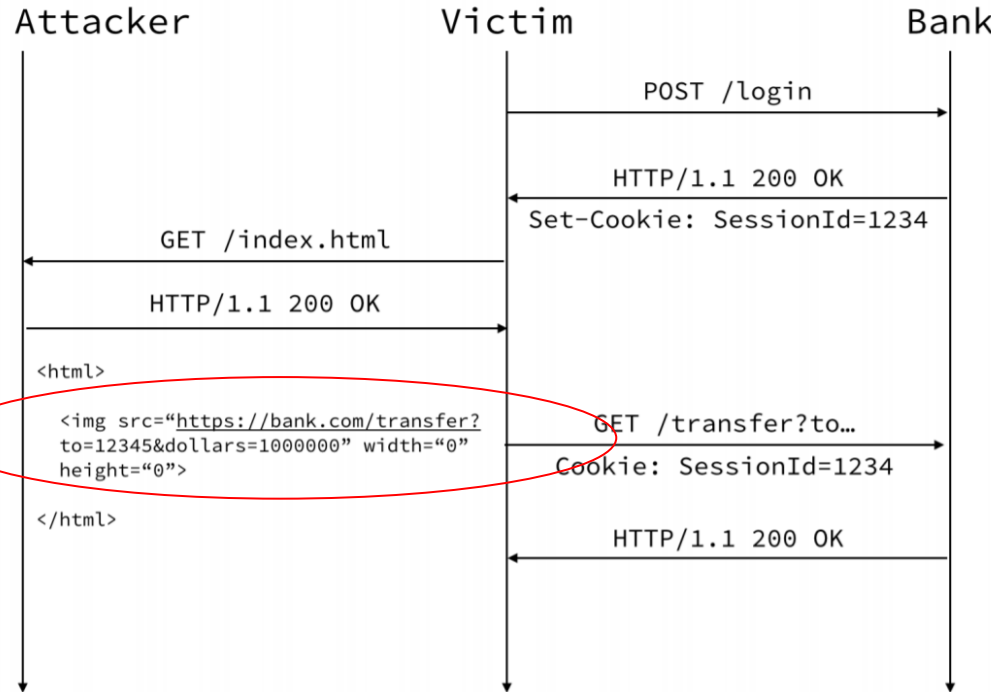
Cross-Site Request Forgery (CSRF)

CSRF is an attack that forces a user to execute unwanted actions on a web application in which they're currently authenticated. It allows an attacker to partly circumvent the same origin policy (SOP), which is designed to prevent different websites from interfering with each other.



CSRF Example

- If the user is logged in to the vulnerable website, their browser will automatically include their session cookie in the request .
- The attacker's page will **trigger an HTTP request** to the vulnerable website.
- The vulnerable website will process the request in the normal way, treat it as having been made by the victim user



CSRF example

The attacker's page will trigger an HTTP request to the vulnerable website.

- Using GET request:

```

```

- 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

- Referer-based validation - verify that the request originated from the application's own domain. e.g., when a user sends a request to the bank through the hacker's website, the Referer of the request points to the hacker's own website.

The image shows a screenshot of an HTTP request. The 'Request Line' is highlighted with a box and labeled 'Request Line'. The 'HTTP Request Headers' section is also highlighted with a box and labeled 'HTTP Request Headers'. The 'Referer' header is highlighted with a red box, showing a URL from a third-party domain: `Referer: http://plus.url.google.com/url?sa=z&n=1400603604168&url=http%3`. The rest of the headers, including Host, Proxy-Connection, Cache-Control, Accept, User-Agent, and Cookie, are visible but not highlighted.

Not safe:

- The method of verifying the Referer value relies on the third party (i.e. the browser) to ensure security.
- Referer field is optional. When the client sends a request, it decides whether to add this field.

CSRF - hidden nonce

- Why CSRF attacks are successful?

All user verification information in the request exists in cookies. Hackers can completely forge the user's request.

- The key to resisting CSRF is to put information in the request that hackers cannot forge, and that information does not exist in cookies.
- **Submit a hidden nonce(i.e. number used only once) with every form**

Nonce (token) is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client.

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)

- Unauthorized cross-origin script access

An attacker “injects” a malicious script into an otherwise trusted website.

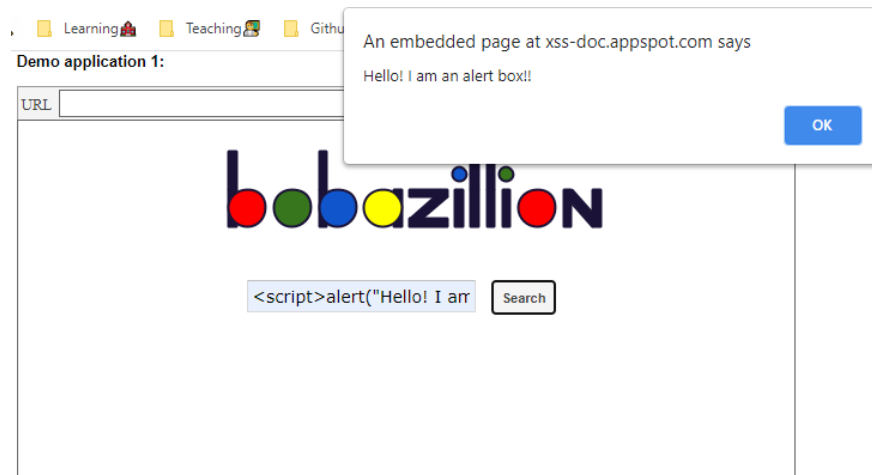
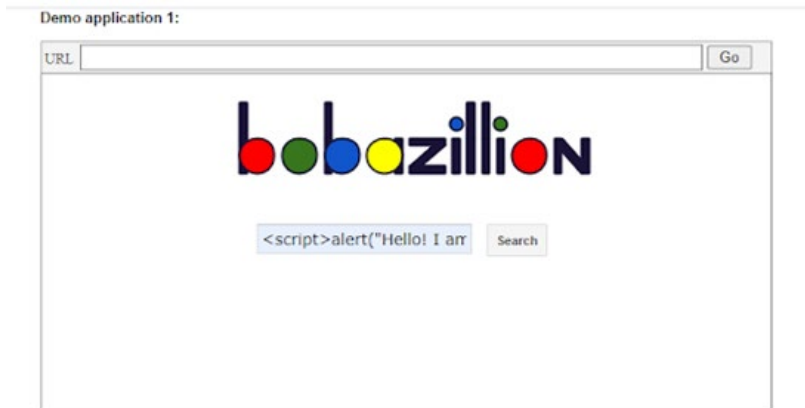
- Consequences: **executing script in a victim’s origin**
 - The injected script gets downloaded and executed by the end user’s browser when the user interacts with the compromised website.
 - Since the script came from a trusted website, it cannot be distinguished from a legitimate script.

Cross-Site Scripting (XSS)

- 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

XSS-- Example

- Reflected XSS attack
- The malicious input is used in the response HTML page.
- <https://www.google.com/about/appsecurity/learning/xss/>
- `<script>alert("Hello\nHow are you?");</script>`



XSS - Defense

- Input Validation and sanitization

- PHP filters

- Reference:

<https://www.php.net/manual/en/filter.filters.sanitize.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>

Thank you!