2024

CCSEP Assignment Report GROUP 11

ISEC3004 – CYBER CRIME AND SECURITY ENHANCED PROGRAMMING

GITHUB REPOSITORY LINK:
HTTPS://GITHUB.COM/SALEEMO97/CCSEP

- Enter your group number from BB (e.g. 1)
  -
- Student IDs and Full names of people in your group
  - Group size must be between 3 to 5 members. Comma separate each name.
  -

	Student Full Name	Student ID
1	Osama Saleem	19926975
2	Raahim Haider	19972570
3	Chong Shi An	20681799
4	Ze Hao Ting	20466213
5	Mike Alan Khor	21186936

- Choose your research topics [Select any Two]

- [ ] HTTP Request Smuggling (HTTP Desync)
- [ ] Unicode Normalisation Vulnerability
- [ ] Unicode Visual Spoofing
- [ ] Web Cache Poisoning
- [ ] Regular Expression Denial of Service (ReDoS)
- [ ] Time-based Side Channel Vulnerability
- [ ] Off-by-one
- [ ] Double Free
- [ ] Use After Free
- [ ] Buffer over-read
- [ ] Server-Side Request Forgery (SSRF)
- [ ] Second order (blind) XSS
- [x] DOM-based XSS
- [ ] Log injection
- [ ] Time Of Check Time of Use
- [x] NoSQL injection
- [ ] Path Traversal (Path Resolution Vulnerability)
- [ ] Cross Site Request Forgery (CSRF)
- [ ] OAUTH Session Fixation
- [ ] JavaScript Module Hijacking
- [ ] CRLF Injection
- [ ] Insecure Deserialization
- [ ] Meltdown
- [ ] Spectre attack
- [ ] Heartbleed

Contents

NoSQL Injection and DOM-Based XSS Vulnerability Analysis

1.0 Discussion on Chosen Vulnerabilities

1.1 Identification of the Vulnerabilities

This project demonstarates the NoSQL injection and DOM-Based XXS injection vulnerabilities. In the following project compraises of thr exploiting of the client side JavaScript and MongoDB to successfully identify and reslove the vulnerabilities.

1.2 Why These Vulnerabilities are Common

These vulnerabilities are common as the data entered through the client side is often passed on directly without any proper validation, causing the database or server to commit unauthorised actions. These vulnerabilities might cause an unauthorised manipulation of data or actions once injected. It can directly affect the authenticity of the data.

1.3 How the Vulnerabilities are Exploited

It is possible to bypass the authentication process by using specially crafted JSON scripts and gain unauthorised access to the server using a NoSQL injection attack. This process would be demonstrated later on in the report. A DOM-Based XXS attack can exploit DOM by utilising malicious JavaScript in DOM by an unsuspecting user.

2.0 NoSQL Injection Vulnerable Code Design

2.1 Vulnerable Code Example

Below is an example of the vulnerable code demonstrating NoSQL injection in a MongoDB query.

```php
// Taking user input directly from POST request
$username = $_POST['username'];

// Decode password input as JSON if possible; if not, keep it as a string
$password = json_decode($_POST['password'], true);
if (is_null($password)) {
    $password = $_POST['password']; // If JSON decode fails, use the original string
}

// Vulnerable query - directly using user input without sanitization or validation
$query = [
    'username' => $username,
    'password' => $password
];
```

The code snippet is vulnerable to NoSQL Injection because it uses the user input directly from $_POST request without any other checks. More precisely, the username is used directly from the input and can be easily manipulated to carry NoSQL operators or commands that interfere with query logic. The password input is decoded as JSON, which, when crafted maliciously, would cause unexpected behaviour. For example, if an attacker submits some JSON string that includes NoSQL operators, the attacker can tamper with the query to trigger an action that would bypass the authentication.

This vulnerability is alarming since it is directly passed without proper validation when the user inputs a query. An attacker can pass an admin Username and a password with the value {"$ne": ''}, resulting in a query where the condition "password is not equal to an empty string" is checked. This is always true for any existing user, meaning this attacker can bypass authentication and obtain unprivileged access. In order to protect against these vulnerabilities, it is a good practice to always thoroughly validate and sanitize all user inputs for the correct type of expected input and use parameterized queries or strict input checks.

3.0 Exploitation Script (NoSQL Injection)

3.1 Exploitation Script Example

```
{ "$ne": "" }
```

An attacker mostly registers himself as a new user to execute a NoSql vulnerability where he gives any username (e.g: testuser) and a password. Password can an any simple text like, “password123”. The username and password will be saved directly to the DB without being hashed or any verification. It would create a valid account to be used later in exploitation. It is especially important since this would imply the existence of a real user in the database that an attacker could use to try to bypass the authentication process.

Once the user account is created, an attacker can attempt to log in using the registered username ("testuser") along with a specially crafted input shown in the snippet above in the password field. The attacker might enter {"$ne": ""} as the password. In MongoDB, the $ne operator means "not equal". When the query runs, MongoDB looks for a record where the username is "testuser" and the password is not an empty string. Since any valid password (including the ones stored in the database) is not empty, this condition will always be true for any user with a non-empty password. As a result, the query will return the user document for "testuser," regardless of the actual password. This allows the attacker to bypass the password check, giving them unauthorized access to the account.

When the query is executed, MongoDB searches for a record of the specified username and an empty string as a password. However, any valid password (including the ones stored in the database) is not empty. This condition will always be true for any user with a non-empty password. This would result in returning a user document for a user "testuser" without a valid password. The practice would allow an attacker to successfully bypass the authentication process and have access to the webpage.

3.2 Potential Consequences of a Successful Exploit

NoSQL injection can result in the manipulation and theft of important data. The information stored in the database can be publicised, breaching the confidentiality of the data. This can damage the reputation of businesses and can lead to a series of critical cyber attacks with the gathered information.

4.0 Mitigation Technique

4.1 Mitigation Code Example

nosql_injection_fixed.php

```php
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    $username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
    $password = $_POST['password'];
    $collection = $db->users;

    // Secure login (protected against NoSQL injection)
    $result = $collection->findOne([
        'username' => $username,
        'password' => $password
    ]);
```

register_fixed.php

```php
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    $username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
    $password = $_POST['password'];
    $collection = $db->users;

    // Check if username already exists
    $existing_user = $collection->findOne(['username' => $username]);
    if ($existing_user) {
        echo '<div class="alert alert-danger" role="alert">Username already exists!</div>';
    } else {
        // Secure registration (protected against NoSQL injection)
        $hashed_password = password_hash($password, PASSWORD_DEFAULT);
        $result = $collection->insertOne([
            'username' => $username,
            'password' => $hashed_password
        ]);
```

4.2 Explanation of Mitigation

The updated code aims to strengthen security against NoSQL injection attacks. A good improvement was securing the username input using the filter_input() function with

FILTER_SANITIZE_STRING. This ensures that no unauthorised or special scripts are passed forward. Cleaning up the user input can reduce exposure to processing malicious data and is necessary to protect against those attacks.

Furthermore, the password is now being stored more securely. The password is hashed using the password_hash() function prior to storing it in the database. The hash password would protect against the leakage of the original password. This process is an essential security practice for the secure development of applications and would protect sensitive data in case of a breach.

The username is run against the stored records in the database to enhance the application's security. There are multiple other ways to discourage duplicate records in the database. The use of the ORM library is another approach towards safely handling the data input and defending against NoSql injections.

Sanitization is a fundamental security measure. This ensures data integrity and defense against exploiting vulnerabilities such as NoSQL in sanitization, and sanitization ensures the application runs smoothly without risking the data stored in the database.

5.0 DOM-Based XSS Vulnerable Code Design

5.1 Vulnerable Code Example

Below is a snippet of vulnerable code that shows a DOM-Based XXS malicious JavaScript

```
<script>
  function showMessage() {
    var userInput = document.getElementById('userInput').value;
    document.getElementById('output').innerHTML = userInput; // Vulnerable line
  }
</script>
```

5.2 Description of the Vulnerability

The code shown above takes unitized input from a user and inserts it to the HTML. The code "document.getElementById('output').innerHTML = userInput;" allows user to input any text as HTML.

5.3 Code Comments and Risks

The input is directly taken and inserted to DOM without any checks. This can allow the user to inject malicious scripts resulting in data theft (cookies, session information). This can also trick the users visiting the website to be redirected to a malicious website.

6.0 Exploitation Script (DOM-Based XSS)

6.1 Exploitation Script Example

```
<img src=x onerror="alert('XSS!')">
```

6.2 Breakdown of the Exploitation Script

This malicious script injects a payload directly to the innerHTML when the showMessage() function is executed. The <img> is used it embed an image but its set to x which is invalid, and wont laid any image. The oneerror is event handler and when it's triggered when the image tries to load throws an error. When triggered, the oneerror processes the code inside the quotes. In this case, it would show "XSS!".

6.3 Potential Consequences of a Successful Exploit

The DOM-Based XXS attack can execute arbitrary JavaScript's. An attacker can use the scripts to hijack sessions and steal cookies and token and cause phishing attacks by redirecting user visiting the website to a malicious website. This can also let attackers execute malicious actions on behalf of someone else by stealing users data without their knowledge.

7.0 Mitigation Technique

7.1 Mitigation Code Example

```
<script>

  function showMessage() {
    var userInput = document.getElementById('userInput').value;
    // Sanitize the input to prevent XSS
    var sanitizedInput = DOMPurify.sanitize(userInput);
    document.getElementById('output').textContent = sanitizedInput;
  }
</script>
```

7.2 Explanation of Mitigation

To mitigate the vulnerability, DOMPurify.sanitize() is introduced. This would handle and sanitize the user input and make sure that the user input does not have any malicious scripts or event handlers before passing the input to the DOM. This is demonstrated in the dom_fixed.php file. By properly sanitizing the input, we can make sure that the sensitive data is safe adds another level of protection prohibiting the entry of script tags, inline event handlers to the webpage.

Furthermore, the innerHTML property is replaced with textContent property. This is safer as if a malicious JavaScript is entered, it treated as plain text rather than a script. This ensures that the client-side browser is not prone to executing scripts. The input entered is not treated as executable code mitigating the risk script-based attacks. Combing these techniques we can ensure that the web application displays secure content towards the users and is safeguarded from external threats and vulnerabilities such as DOM based XXS attacks.

8.0 Conclusion

To summarize the discussion about NoSQL Injection and DOM-Based XSS vulnerabilities, we can say that these are kinds of security risks modern web applications are exposed to exploitation if user inputs have not been validated or sanitized properly. The two vulnerabilities prove that an attacker can control different kinds of inputs to bypass the authentication mechanisms or give them the possibility of running malicious scripts within a browser context. NoSQL Injection: A vulnerability that allows malicious attackers to access databases by manipulating query logic; DOM-Based XSS, this exploit allows attackers to Execute dangerous scripts on the client browser allows attackers to access the sensitive data and compromised accounts.

Mitigation strategies help to defend against these vulnerabilities. Input sanitization, writing some safer code like parameterized queries, or using a library such as DOMPurify will reduce the attack surface. Moreover, replacing vulnerable operations like innerHTML with textContent ensures user inputs are processed as plain text and will not execute an injected code. These adjustments enhance the best practices for secure app development that help in progress aims to ensure data integrity and also keep users away from malicious attackers.

In summary, is taking a proactive security posture important when creating web applications? Validation and sanitization, along with secure password management/service query handling, strengthen the defense against injection vulnerability so these can be used to implement proper security. To stay safe and keep the code secure from server-side and client-side attacks, the latest security practices should help protect sensitive data and follow the clients' trust towards a service/business.