Navigating CSRF Vulnerability

Developer Insights & Solutions

Agenda

- Understanding CSRF
- Impact of CSRF Attacks
- Remediation Strategies
- Demo
- Q&A

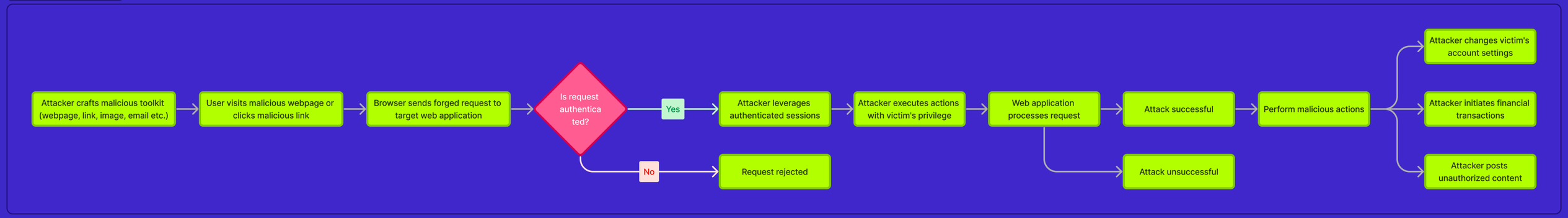
Understanding CSRF

“Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they’re currently authenticated.”

– The OWASP Foundation

CSRF Attack Workflow

How CSRF Attack works



Impact of CSRF attacks

Risks Posed

- Unauthorized Actions
- Data Breaches
- Session Hijacking
- Malicious Activities

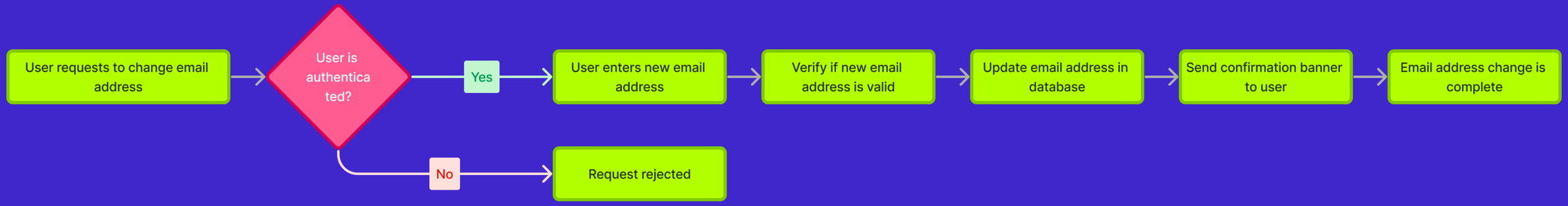
Potential Consequences

- Reputation Damage
- User Disengagement
- Legal Ramifications
- Long-term Damage

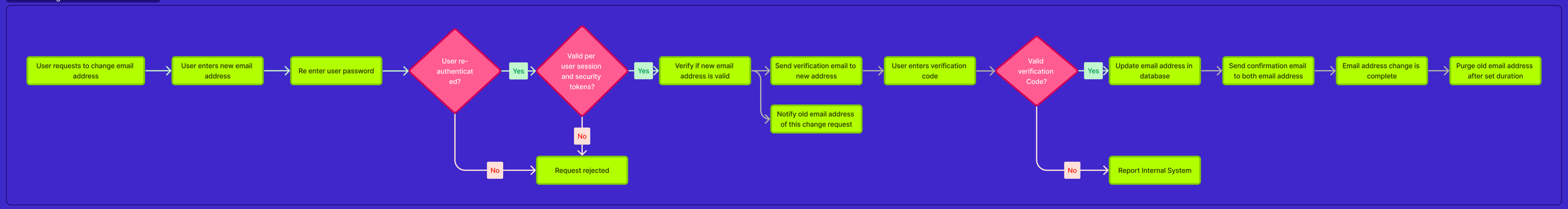
Remediation Strategies

Extend Business Logic + Implement Security Features

Insecure Change Email Address Workflow



Secure Change Email Address Workflow



Security Features

CSRF Tokens

Same-Site Cookies

Origin Checks

Security Tools [2FA, MFA, WAF etc.]

Demo - PortSwigger Web Security Academy

Q&A

Subject: Regarding SQL Injection Identified by Checkmarx Platform

Hi John,

Thank you for reaching out to us regarding the SQL injection identified by our product in your code scan.

After reviewing your code snippet, it appears that the vulnerability arises from directly incorporating user input ($_GET['id']) into the SQL query without adequate sanitization or parameterization. This practice can expose your application to SQL injection attacks, where malicious actors manipulate input to execute unauthorized SQL commands.

To address the SQL injection vulnerability, we recommend revising your code to utilize parameterized queries or prepared statements. This approach separates user input from the SQL query, significantly reducing the risk of SQL injection attacks.

Please find below a modified version of your code snippet incorporating parameterized queries:

```php
<?php
include_once "lib/db.php";
include "lib/sql_form.php";

$id = isset($_GET['id']) ? $_GET['id'] : '';
$id = str_replace("/", "", $id);

$query = "SELECT * FROM listings WHERE id = ?";
$stmt = $pdo->prepare($query);
$stmt->execute([$id]);
$result = $stmt->fetch(PDO::FETCH_ASSOC);

if ($result['status'] == "DELETED") {
    print("This file is deleted.");
    exit;
}

$_POST['form_id'] = $result['form_id'];
$_POST['fields'] = explode(",", $result['fields']);

excel_report_results();
?>
```

Please note that the modified version of the code provided is for explanatory purposes only and is intended to demonstrate a potential solution for addressing SQL injection vulnerabilities. It is not intended to be used as-is in production environments, and its effectiveness may vary depending on the specific context and requirements of your application.

We understand that you may have implemented infrastructure and environment for your workloads, which likely provides security features to mitigate SQL injection risks. However, it's crucial to note that relying solely on these security measures may not be sufficient to safeguard against all potential threats. While these features are valuable, they should be complemented with secure coding practices on the application side for comprehensive security.

Implementing this change will enhance the security of your application by mitigating the risk of SQL injection attacks.

You can also explore our glossary page dedicated to SQL injection vulnerabilities and best practices by visiting https://checkmarx.com/glossary/sql-injection.

If you have any further questions or require additional assistance, please feel free to reach out.

Best regards,
Pranam