**VULNERABILITIES HADNDLED**

**IN**

**CLOUD SECUIRTY POC**

**A1:2017-Injection**

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

Primary Defenses:

* Option 1: Use of Prepared Statements (with Parameterized Queries)
* Option 2: Use of Stored Procedures
* Option 3: Whitelist Input Validation
* Option 4: Escaping All User Supplied Input

Additional Defenses:

* Also: Enforcing Least Privilege
* Also: Performing Whitelist Input Validation as a Secondary Defense

Implementation in POC:

1. Input Validation for all input fields on all forms of web application.

# A2:2017-Broken Authentication

# Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.

Primary Defenses:

1. Authentication Solution and Sensitive Accounts
   * Do NOT allow login with sensitive accounts that can be used internally within the solution to any front-end user-interface
   * Do NOT use the same authentication solution (e.g. IDP / AD) used internally for unsecured access (e.g. public access / DMZ)
2. Implement Proper Password Strength Controls
   * Password Length
     + Minimum length - Passwords shorter than 8 characters are considered to be weak (NIST SP800-63B).
     + Maximum password length should not be set too low, as it will prevent users from creating passphrases. It is important to set a maximum password length to prevent long password Denial of Service attacks.
   * Allow usage of all characters including Unicode and whitespace. There should be no password composition rules limiting the type of characters permitted.
   * In almost all circumstances, passwords should be hashed rather than encrypted, as this makes it difficult or impossible for an attacker to obtain the original passwords from the hashes.
   * Ensure credential rotation when a password leak, or at the time of compromise identification.
3. Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
4. Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

Implementation in POC:

1. Input Validation for email & password fields on all forms of web application.
2. Checked password for good strength (minimum 8 & maximum 64 characters), requires at least one Uppercase, lowercase, digit and special characters (#?!@$%^&\*-)

# A3:2017-Sensitive Data Exposure

# Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

Primary Defenses:

# Classify data processed, stored or transmitted by an application.

# Apply controls as per the classification.

# Don’t store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.

# Make sure to encrypt all sensitive data at rest.

# Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.

# Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).

# Disable caching for response that contain sensitive data.

# Store passwords using strong adaptive and salted hashing functions.

# Implementation in POC:

# A4:2017-XML External Entities (XXE)

# This attack occurs when untrusted XML input containing a reference to an external entity is processed by a weakly configured XML parser.

# This attack may lead to the disclosure of confidential data, denial of service, Server Side Request Forgery (SSRF), port scanning from the perspective of the machine where the parser is located, and other system impacts.

# Primary Defenses:

# Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.

# Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system.

# Implement positive (“whitelisting”) server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.

# The safest way to prevent XXE is always to disable DTDs (External Entities) completely.

# Implementation in POC:

# A5:2017-Broken Access Control

# Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.

# Primary Defenses

# Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

# With the exception of public resources, deny by default.

# Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.

# Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.

# Unique application business limit requirements should be enforced by domain models.

# Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.

# Log access control failures, alert admins when appropriate (e.g. repeated failures).

# JWT tokens should be invalidated on the server after logout.

# Implementation in POC:

# Added two roles for the user – normal user & admin user, maintained in DB

# Logged-in user details are stored in authReducer using Redux, also data to identify admin user is also stored in authReducer.

# All URLs are redirected to login when user is not logged in. Also VerifyToken is called on loading login page.

# UserList on Dashboard is only visible to admin user.

# A6:2017-Security Misconfiguration

# Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc to gain unauthorized access or knowledge of the system. Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.

# Primary Defenses:

# A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment

# A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process.

# A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).

# Implementation in POC:

# A7:2017-Cross-Site Scripting (XSS)

# Reflected XSS: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim’s browser. Typically the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

# Stored XSS: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

# DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

# Primary Defenses:

# Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS.

# Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities.

# Never Insert Untrusted Data except in Allowed Locations

# HTML Encode Before Inserting Untrusted Data into HTML Element Content

# Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes

# JavaScript Encode Before Inserting Untrusted Data into JavaScript Data Values

# HTML Encode JSON values in an HTML context and read the data with JSON.parse

# CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

# URL Encode Before Inserting Untrusted Data into HTML URL Parameter Values

# Sanitize HTML Markup with a Library Designed for the Job

# Avoid JavaScript URLs

# Prevent DOM-based XSS

# Similar context sensitive escaping techniques can be applied to browser APIs in case of DOM XSS.

# Implementation in POC:

# A8:2017-Insecure Deserialization

# Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

# Primary Defenses:

# Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.

# Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.

# Isolating and running code that deserializes in low privilege environments when possible.

# Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.

# Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.

# Monitoring deserialization, alerting if a user deserializes constantly.

# Implementation in POC:

|  |
| --- |
| **A9:2017-Using Components with Known Vulnerabilities** |

# Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

# Primary Defenses:

# Implementation in POC:

# A10:2017-Insufficient Logging & Monitoring

# Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data.

# Primary Defenses:

# Implementation in POC: