|  |  |
| --- | --- |
| Name: | Ronit Gupta |
| Lab User ID: | 23SEK3324\_U12  528646462716 |
| Date: | 10/01/2024 |
| Application Name: | Juice-Shop  https://github.com/juice-shop/juice-shop.git |

**Follow the below guidelines:**

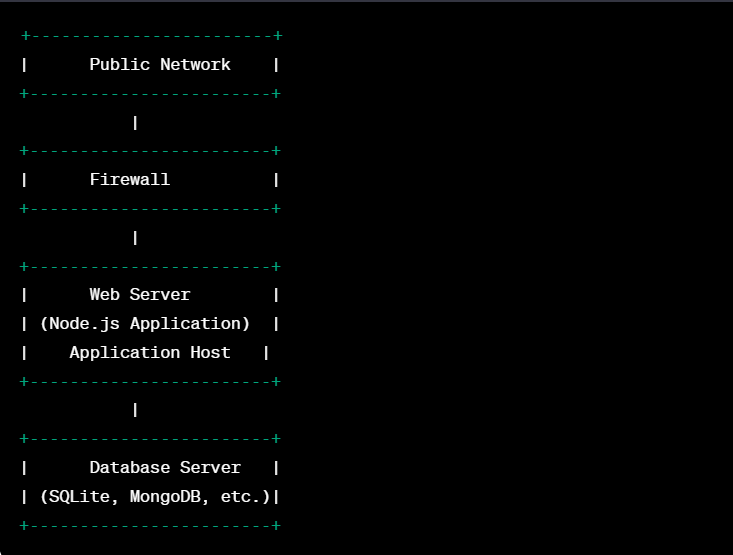




System Architecture:

(Understand the system and document the physical and logical architecture of the system, use the shapes and icons to capture the system architecture)

**Physical Architecture:**



**Logical Architecture:**



Define system’s normal behavior:

(Define the steady state of the system is defined, thereby defining some measurable outputs which can indicate the system’s normal behavior)

The Juice Shop is intentionally designed to be insecure and contains deliberate security vulnerabilities for educational and testing purposes. Below are some normal behaviors or characteristics of the Juice Shop insecure web application:

**Security Vulnerabilities:**

Juice Shop intentionally includes various security vulnerabilities such as SQL injection, cross-site scripting (XSS), insecure direct object references (IDOR), and others. These vulnerabilities are designed to help users learn about common web application security issues.

**Insecure Session Management:**Juice Shop may exhibit insecure session management, allowing users to learn about potential risks related to the mishandling of user sessions.

**Educational Resources:**Juice Shop may provide educational resources, hints, and documentation to help users understand the vulnerabilities and learn how to remediate them.

Hypothesis:

(During an experiment, we need a hypothesis for comparing to a stable control group, and the same applies here too. If there is a reasonable expectation for a particular action according to which we will change the steady state of a system, then the first thing to do is to fix the system so that we accommodate for the action that will potentially have that effect on the system. For eg: "If one of our database servers fails, our service will automatically switch to a backup server, and users will not experience any downtime or data loss.")



**Hypothesis**: The Juice Shop application may have additional security vulnerabilities not covered in widely available documentation or tutorials.

**Rationale**: Despite being well-documented, there might be lesser-known vulnerabilities or variations that users may discover during testing.

**Hypothesis**: The Juice Shop application contains a deliberate SQL injection vulnerability in the user login form.

**Rationale**: This is a well-documented vulnerability in Juice Shop, widely known among users for educational purposes.

**Hypothesis**: The application uses intentionally weak authentication credentials for administrative access.

**Rationale**: Weak credentials are part of the deliberately insecure design of Juice Shop, documented in educational materials.

**Known**

Things we are aware of but don’t understand.

Things we are aware of and understand.

**Hypothesis**: There might be unforeseen security risks or vulnerabilities in the application not known to the development team or the community.

**Rationale**: Despite thorough testing and documentation, the complexity of software systems can sometimes lead to unexpected vulnerabilities.

**Hypothesis**: Interaction between multiple vulnerabilities might result in security issues not predicted by the development team.

**Rationale**: The combination of certain vulnerabilities may lead to emergent risks that were not apparent when analyzing individual vulnerabilities in isolation.

**Hypothesis**: Users might uncover new and creative ways to exploit known vulnerabilities for unintended outcomes.

**Rationale**: As users interact with the application, they may discover innovative approaches or chained exploits not explicitly outlined in documentation.

**Unknown**

Things we are neither aware of nor understand.

Things we understand but are not aware of.

**Known**

**Unknown**

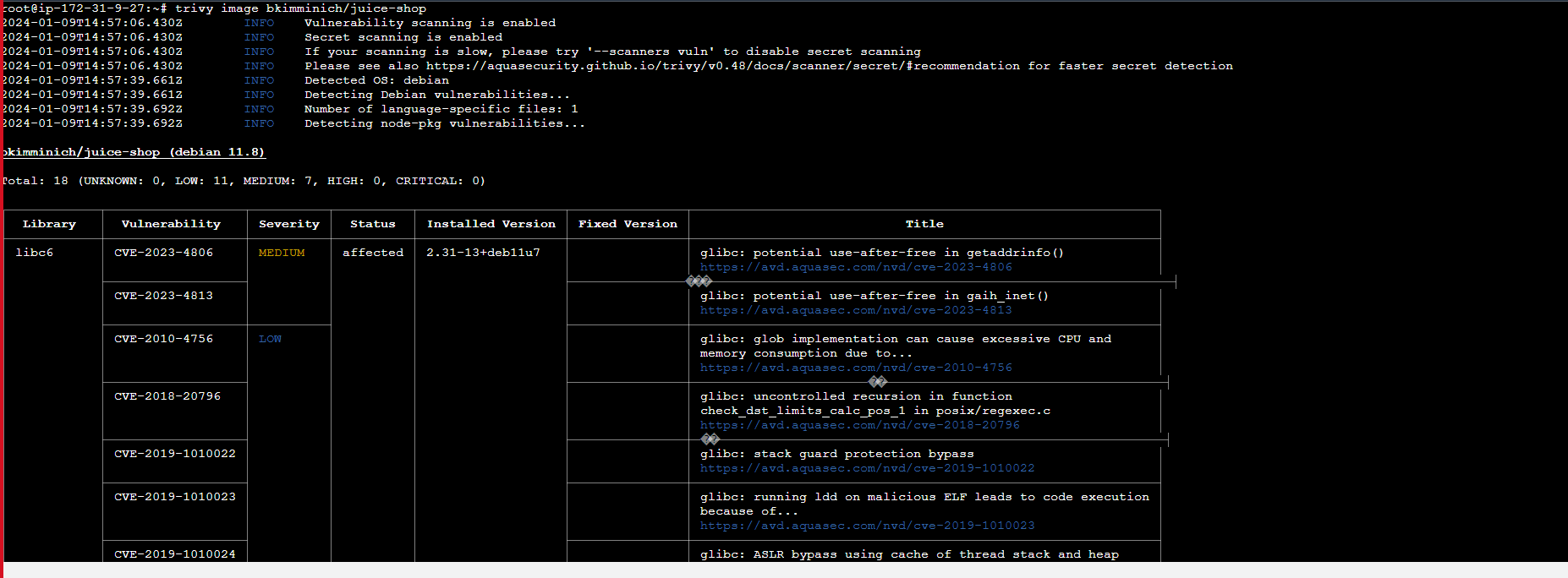
**Vulnerability analysis using trivy:**

root@ip-172-31-0-232:~# trivy repo https://github.com/juice-shop/juice-shop.git

----

bkimminich/juice-shop (debian 11.8)

Total: 18 (UNKNOWN: 0, LOW: 11, MEDIUM: 7, HIGH: 0, CRITICAL: 0)



**Vulnerability analysis using snyk:**

A screenshot of a computer

Description automatically generated

**Some critical vulnerability analysis:**

1. **Code Injection:** Code injection is a type of attack that allows an attacker to inject malicious code into an application through a user input field, which is then executed on the fly. Code injection vulnerabilities are rather rare, but when they do pop up, it is often a case where the developer has attempted to generate code dynamically.

**Mitigation Techniques:**

\* Avoid the use of dangerous functions

In JavaScript, avoid the use of eval(), setTimeout(), setInterval() and the Function constructor, especially when dealing with user input.

\* Utilize a static analysis tool

Adding a static application security testing (SAST) tool to your devops pipeline as an additional line of defence is an excellent way to catch vulnerabilities before they make it to production.

2. **Regular Expression Denial of Service (ReDoS):** Regular Expression Denial of Service (ReDoS) is a type of cyber attack that exploits the way certain regular expressions are processed to cause a denial of service. It occurs when a maliciously crafted input causes the regular expression engine to excessively backtrack while attempting to match the input against the regular expression pattern.

**Mitigation Techniques:**

\* Avoid using overly complex or nested regular expressions that can lead to exponential backtracking. Keep regular expressions simple and well-optimized.

\* Different regex engines may have varying performances and optimizations. Choose a regex engine that is well-suited for your application and requirements.

\* Implement input length limits to prevent excessively long inputs that could trigger ReDoS. This helps to mitigate the impact of the attack.

3. **Uninitialized Memory Exposure:** Uninitialized Memory Exposure is a security vulnerability that occurs when a program or application reads and exposes the contents of uninitialized memory to the user or an attacker. Uninitialized memory typically contains random or leftover data from previous operations, and exposing this information can lead to information leaks.

**Mitigation Techniques:**

\* Failure to Initialize Variables:  
Programmers might forget to initialize variables before using them, leaving them with unpredictable values.

\* Dangling Pointers:  
References or pointers that are not properly managed can lead to accessing uninitialized or freed memory.

\* Memory Corruption:  
Bugs or vulnerabilities that corrupt memory might lead to uninitialized memory exposure.

**4. Unvalidated Redirects and Forwards:** Attackers can exploit unvalidated redirects to redirect users to malicious websites or conduct phishing attacks.

**Mitigation Techniques:**

\* Avoid using user-controlled input for constructing redirects.

\* Validate and sanitize input parameters for redirects and forwards.

\* Implement proper session management and use secure, time-limited tokens.

Experiment:

(Document your Preparation, Implementation, Observation and Analysis )

Steps:

1. Install Docker

2. Run docker pull bkimminich/juice-shop

3. Run docker run --rm -p 3000:3000 bkimminich/juice-shop

4. Browse to http://localhost:3000 (on macOS and Windows browse to http://192.168.99.100:3000 if you are using docker-machine instead of the native docker installation)

**Output Snapshot**

Application was live on 65.2.127.34:3001.

