|  |  |
| --- | --- |
| Name: | Ronit Gupta |
| Lab User ID: | 23SEK3324\_U12  528646462716 |
| Date: | 10/01/2024 |
| Application Name: | Vulnerable-Java-Application  https://github.com/DataDog/vulnerable-java-application.git |

**Follow the below guidelines:**





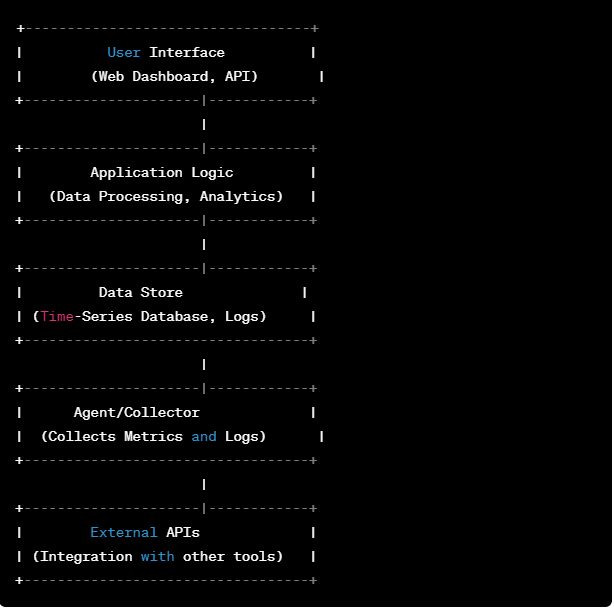
System Architecture:

(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)

Features of system’s normal behavior are:

**Data Collection:**

The Java application incorporates the DataDog agent or library to collect metrics, traces, and logs.

**Instrumentation:**   
The application is instrumented with DataDog APIs to capture and send relevant data points. This includes metrics such as response times, error rates, and throughput.

**Application Metrics:**  
Custom application metrics, specific to the Java application's business logic, are defined and collected. These metrics could include user registrations, order processing times, or any other key performance indicators.

**Error Tracking:**  
DataDog captures and reports errors and exceptions, providing insights into the types and frequency of errors occurring within the application.

**Alerting:**Thresholds and alerting policies are configured in DataDog to notify relevant personnel when certain metrics or events surpass predefined thresholds. This enables proactive issue resolution.

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 website tester application correctly identifies and reports well-known vulnerabilities such as SQL injection, cross-site scripting (XSS), and security misconfigurations.

**Rationale**: These are common vulnerabilities that the application is expected to recognize and report.

**Hypothesis**: The website tester application may have limitations in detecting certain variations or new techniques of known vulnerabilities.

**Rationale**: Despite being aware of known vulnerabilities, the application might not cover all possible evasion techniques or variations.

**Known**

Things we are aware of but don’t understand.

Things we are aware of and understand.

**Hypothesis**: During testing, users may discover undocumented features or functionalities in the website tester application that could impact the accuracy of vulnerability detection.

**Rationale**: The documentation may not cover every aspect of the application's behavior or configuration.

**Hypothesis**: The website tester application might have undiscovered vulnerabilities that are not part of the standard testing scenarios.

**Rationale**: There could be unforeseen interactions or code paths that lead to security vulnerabilities not explicitly tested for.

**Unknown**

**Unknown**

**Known**

Things we are neither aware of nor understand.

Things we understand but are not aware of.

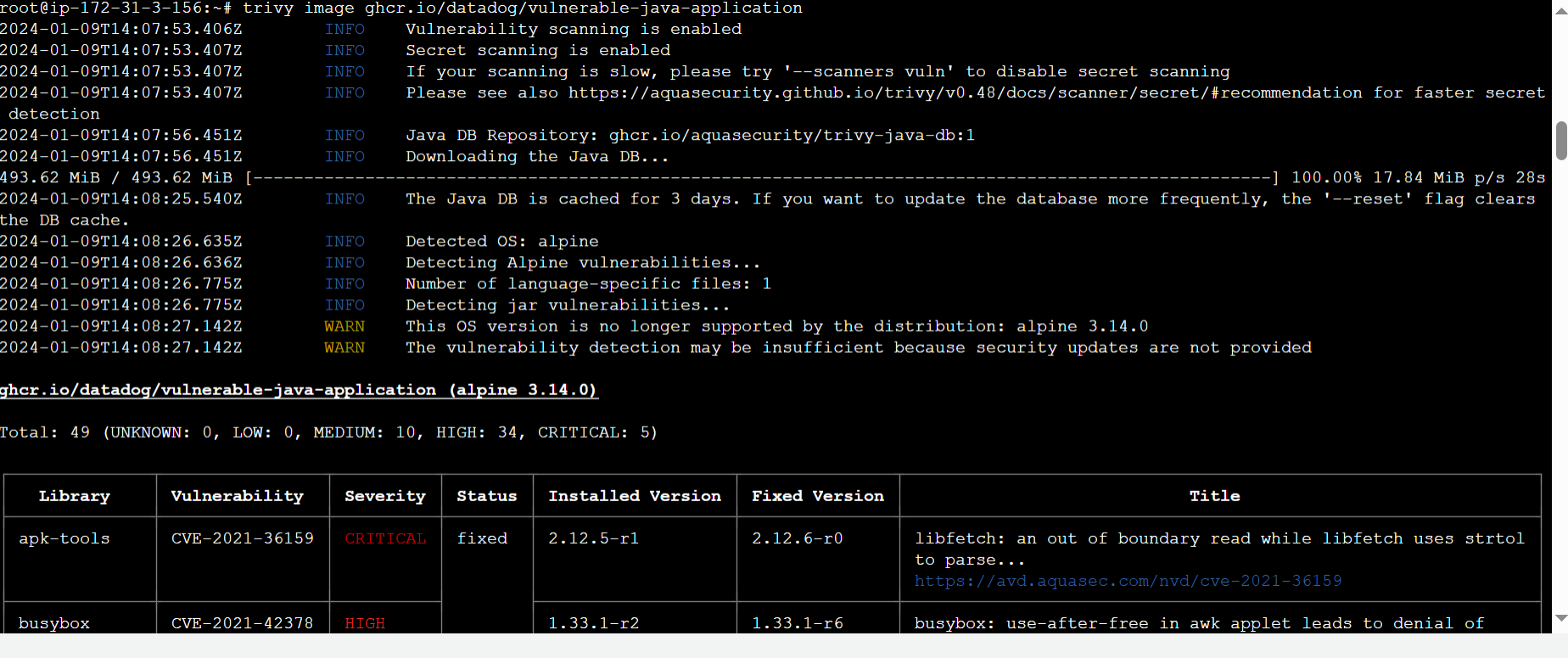
**Vulnerability analysis using trivy:**

root@ip-172-31-3-156:~# trivy image ghcr.io/datadog/vulnerable-java-application

------

ghcr.io/datadog/vulnerable-java-application (alpine 3.14.0)

Total: 49 (UNKNOWN: 0, LOW: 0, MEDIUM: 10, HIGH: 34, CRITICAL: 5)



**Vulnerability analysis using snyk:**

A screenshot of a computer

Description automatically generated

**Some critical vulnerability analysis:**

**1. Out-of-bounds Write:**An "Out-of-bounds Write" vulnerability occurs when a program writes data beyond the boundaries of a buffer, array, or allocated memory. This can lead to unintended consequences, including overwriting critical data structures, causing crashes, or potentially enabling attackers to execute arbitrary code.

**Mitigation Techniques:**

\* Boundary Checking:  
Implement strict boundary checking to ensure that all write operations stay within the allocated memory bounds.

\* Use Safe String and Memory Functions:  
Use safer string and memory manipulation functions (e.g., strncpy, snprintf, memcpy\_s in C) that automatically handle bounds checking.

\* Static Code Analysis:  
Employ static code analysis tools to identify potential out-of-bounds write vulnerabilities during development.

**2. Loop with Unreachable Exit Condition ('Infinite Loop'):** A "Loop with Unreachable Exit Condition," commonly known as an "Infinite Loop," occurs when the exit condition of a loop is never satisfied, leading to the loop executing indefinitely. This situation can result in a program becoming unresponsive, consuming excessive resources, or causing a system to hang.

**Mitigation Techniques:**

\* Proper Exit Conditions:   
Ensure that the exit condition of the loop is correctly defined and will eventually evaluate to false.

\* Counter Increment/Decrement:   
If the loop relies on a counter, verify that the counter is being incremented or decremented correctly inside the loop.

\* Use Break Statements:  
Introduce break statements or return statements inside the loop to force an exit when a certain condition is met.

**3. Improper Certificate Validation:** Improper Certificate Validation is a security vulnerability that occurs when a system or application fails to adequately verify the authenticity and validity of certificates during the SSL/TLS handshake process. This vulnerability can be exploited by attackers to perform various types of man-in-the-middle attacks, such as intercepting or altering encrypted communication between parties.

**Mitigation Techniques:**

\* Always Validate Certificates:  
Ensure that certificates are always validated during the SSL/TLS handshake process. Reject connections with untrusted or improperly validated certificates.

\* Use Trusted Certificate Authorities (CAs):  
Only accept certificates issued by trusted and reputable Certificate Authorities. Configure the application or system to recognize and trust these CAs.

\* Check Certificate Revocation:  
Implement checks for certificate revocation status using mechanisms like Certificate Revocation Lists (CRLs) or Online Certificate Status Protocol (OCSP).

**4. Cross-Site Request Forgery (CSRF):** CSRF occurs when an attacker tricks a user's browser into making unintended requests on a site where the user is authenticated.

**Mitigation Techniques:**

\* Use anti-CSRF tokens in forms to validate the authenticity of requests.

\* Ensure that state-changing requests require user authentication and authorization.

\* Implement SameSite cookie attributes to prevent cross-site request forgery.

Experiment:

(Document your Preparation, Implementation, Observation and Analysis )

Steps:

1. Build the image locally, or use ghcr.io/datadog/vulnerable-java-application:

2. Run:  
docker run --rm -p 8000:8000 ghcr.io/datadog/vulnerable-java-application

3. You can then access the web application at http://127.0.0.1:8000

**Output snapshot**

Application was live on 43.204.108.120:8000

