Chaos Engineering - Report

Name:	Sunidhi Agrawal
Lab User ID:	23SEK3324_U11
Date:	10.01.2024
Application Name:	Vulnerable Java application

Follow the below guidelines:

Define Hypotheses and Scenarios:

- Similar to Chaos Engineering, start by defining hypotheses and scenarios related to potential threats.

Inject Controlled Failure:

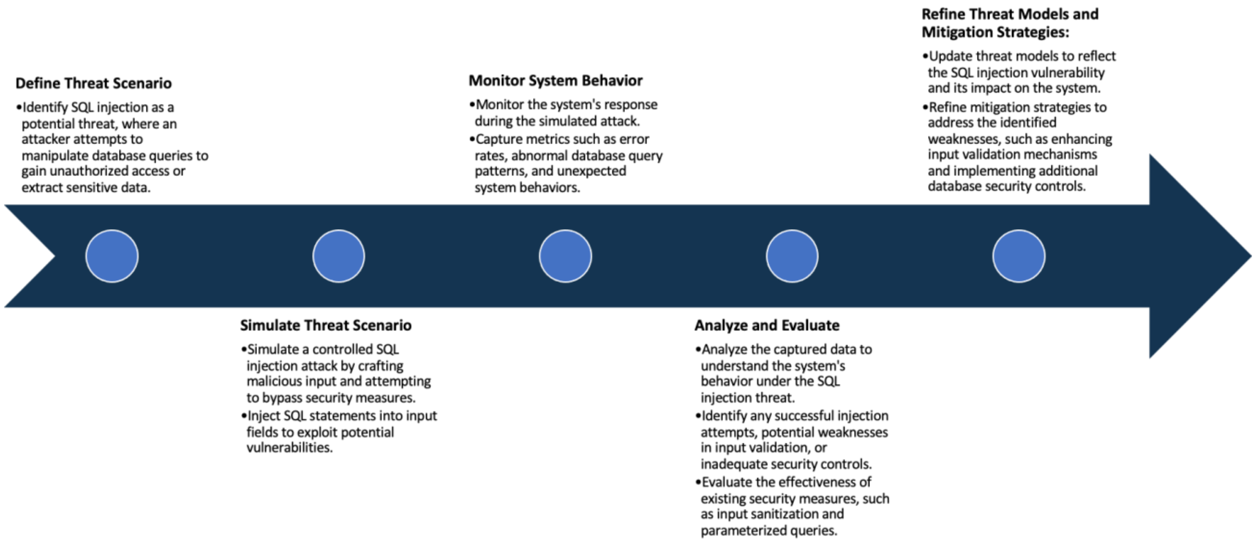
- Introduce controlled failure scenarios that mimic potential attack vectors or vulnerabilities identified in Threat Modeling.
- Simulate failure conditions, such as network disruptions, component failures, or data breaches, to observe the system's response.

Measure System Behavior:

- Capture and measure relevant system behavior metrics during the chaos experiments.
- Monitor the system's response to the injected failures, including performance metrics, error rates, and security-related indicators.
- Analyze and compare the observed behavior against the expected outcomes defined in the Threat Modeling process.

Learn and Iterate:

- Learn from the results of the chaos experiments and iterate on the Threat Modeling process.
- Analyze the observations and insights gained from the chaos experiments to refine the Threat Models. Update threat scenarios, adjust mitigation strategies, and improve security controls based on the lessons learned.



Define Threat Scenario

- Identify SQL injection as a potential threat, where an attacker attempts to manipulate database queries to gain unauthorized access or extract sensitive data.

Simulate Threat Scenario

- Simulate a controlled SQL injection attack by crafting malicious input and attempting to bypass security measures.
- Inject SQL statements into input fields to exploit potential vulnerabilities.

Monitor System Behavior

- Monitor the system's response during the simulated attack.
- Capture metrics such as error rates, abnormal database query patterns, and unexpected system behaviors.

Analyze and Evaluate

- Analyze the captured data to understand the system's behavior under the SQL injection threat.
- Identify any successful injection attempts, potential weaknesses in input validation, or inadequate security controls.
- Evaluate the effectiveness of existing security measures, such as input sanitization and parameterized queries.

Refine Threat Models and Mitigation Strategies:

- Update threat models to reflect the SQL injection vulnerability and its impact on the system.
- Refine mitigation strategies to address the identified weaknesses, such as enhancing input validation mechanisms and implementing additional database security controls.

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)

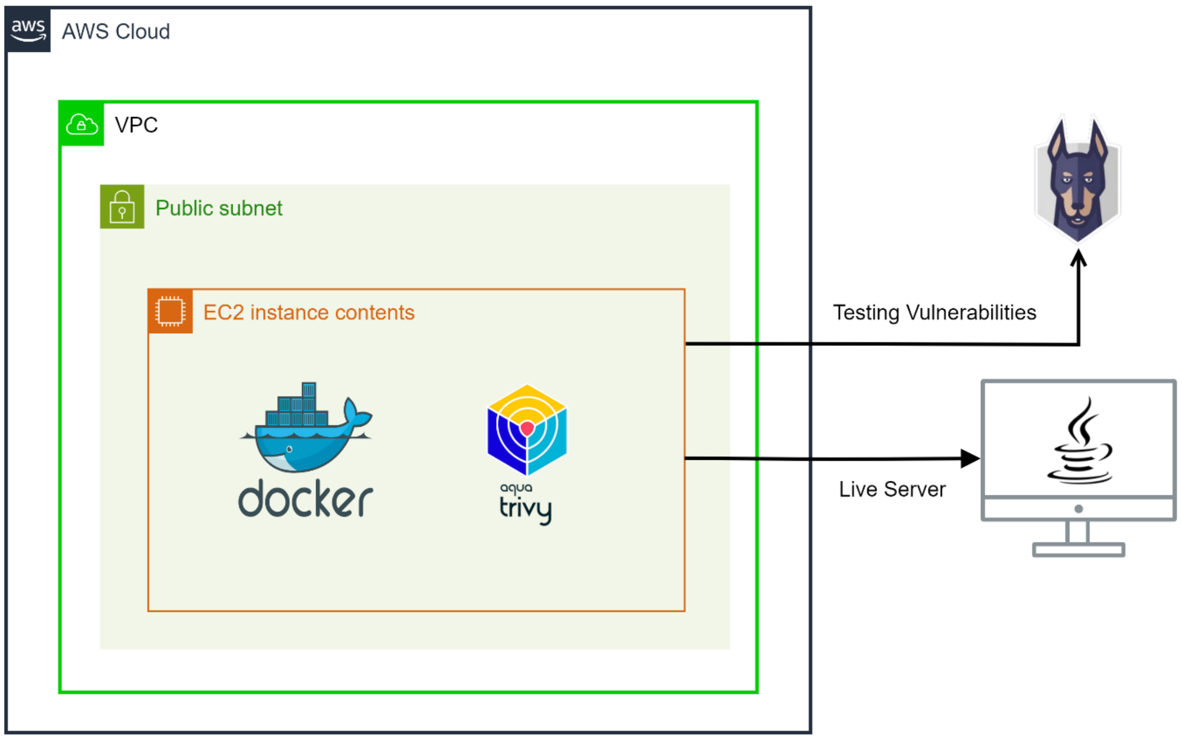
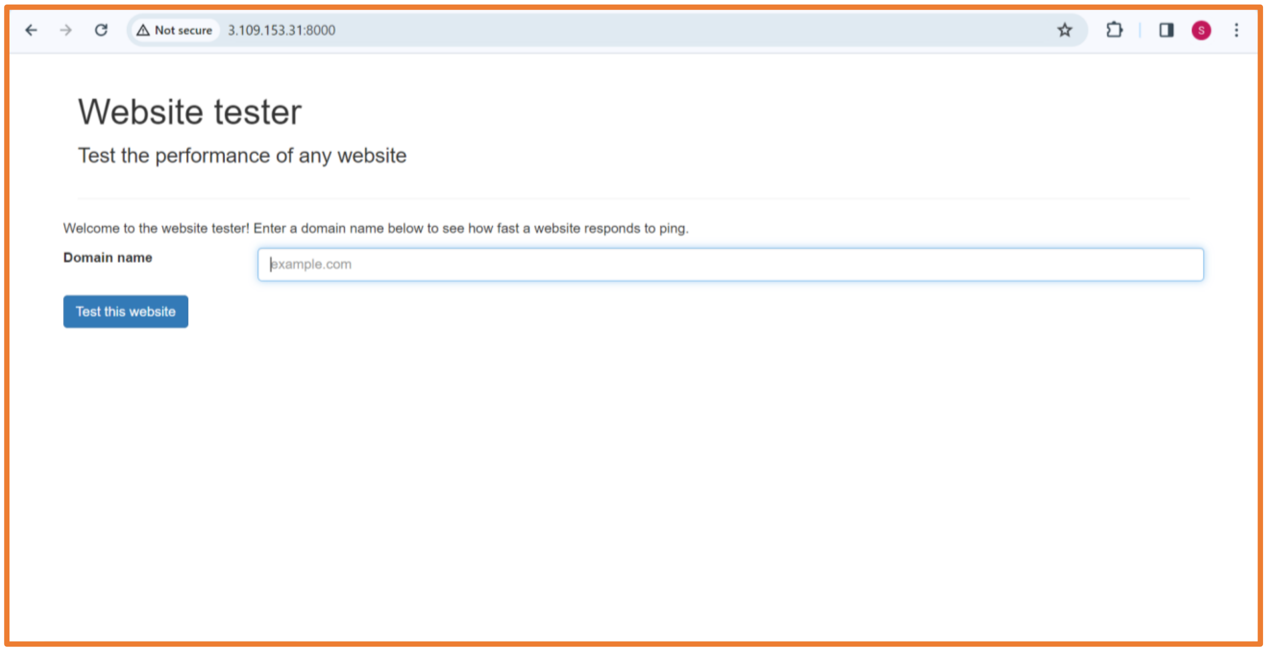


Figure 1:System Architecture



Website tester

Test the performance of any website

Welcome to the website tester! Enter a domain name below to see how fast a website responds to ping.

Domain name example.com

Test this website

Figure 2: Vulnerable Java application

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 normal behavior of a system refers to its typical or expected state when operating under standard conditions. In context of this Vulnerable Java application, the steady state can be described by several key metrics and behaviors. Here are some examples of measurable outputs that can indicate the system's normal behavior:

---

1. **Response Time:**
   - Measure the average response time for HTTP requests served by the web server.
   - Establish baseline response times under normal load conditions.

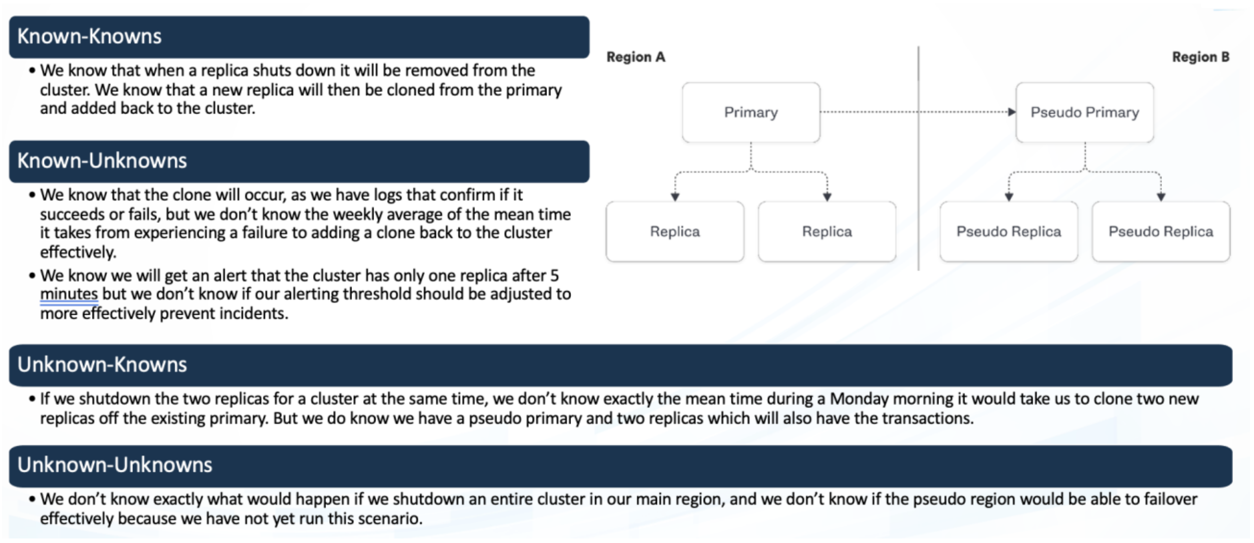
2. **Throughput:**
   - Monitor the number of successful requests processed per unit of time.
   - Define a baseline for the expected throughput under normal circumstances.

3. **Resource Utilization:**
   - Monitor CPU and memory usage of the Docker containers running the Java application.
   - Set thresholds for normal resource utilization and ensure they are not consistently exceeded.

4. **Container Health:**
   - Utilize Docker health checks or other container monitoring tools to ensure the health of the application containers.
   - Set criteria for what constitutes a healthy container.

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.")



Known-Knowns

- We know that when a replica shuts down it will be removed from the cluster. We know that a new replica will then be cloned from the primary and added back to the cluster.

Known-Unknowns

- We know that the clone will occur, as we have logs that confirm if it succeeds or fails, but we don't know the weekly average of the mean time it takes from experiencing a failure to adding a clone back to the cluster effectively.
- We know we will get an alert that the cluster has only one replica after 5 minutes but we don't know if our alerting threshold should be adjusted to more effectively prevent incidents.

Unknown-Knowns

- If we shutdown the two replicas for a cluster at the same time, we don't know exactly the mean time during a Monday morning it would take us to clone two new replicas off the existing primary. But we do know we have a pseudo primary and two replicas which will also have the transactions.

Unknown-Unknowns

- We don't know exactly what would happen if we shutdown an entire cluster in our main region, and we don't know if the pseudo region would be able to failover effectively because we have not yet run this scenario.

Known	- **Known Vulnerabilities:** The application intentionally includes specific vulnerabilities, such as Command Injection and Server-Side Request Forgery (SSRF), which are documented and understood by users for educational purposes. - **Known Attack Scenarios:** Users are aware of the predefined attack scenarios corresponding to the known vulnerabilities, providing a controlled environment for learning and experimentation.	- **User Exploration:** While users know about the existence of certain vulnerabilities, the specific ways to exploit them or uncover potential security issues may vary.
Unknown	- **Application Behavior:** Users may not be fully aware of the entire codebase, making it challenging to predict how the application responds in certain scenarios or how specific components interact.	- **Unforeseen Vulnerabilities:** Users may discover new vulnerabilities or weaknesses in the application that were not intentionally introduced, leading to unexpected security challenges. - **Future Updates:** As the vulnerable application evolves, new vulnerabilities or scenarios might be introduced, creating unknowns for users who are not familiar with the latest version.

In the context of chaos engineering, a chaos hypothesis is a statement that predicts how a system will respond to a particular event or experiment. It is a crucial part of the chaos engineering process as it provides a basis for comparison between the normal, stable state of the system (control group) and the state during the chaos experiment.

If the Website Tester Service is subjected to a simulated Command Injection attack, the system should detect and mitigate the threat, preventing any unauthorized execution of commands. We expect that by implementing security measures, such as input validation and command filtering, the application will maintain its steady state, ensuring that user data remains secure, and the system continues to operate without unexpected disruptions.

This hypothesis focuses on a specific vulnerability (Command Injection) and aims to test the effectiveness of implemented security measures. The expectation is that the system, when exposed to a known threat, will respond by mitigating the risk and maintaining its stability. The emphasis is on fortifying the system against potential security breaches and ensuring a steady state even in the face of intentional attacks.

Experiment:

(Document your Preparation, Implementation, Observation and Analysis)

Project Overview:

The primary objective of this project was to deploy and assess the security of the "Websites Tester Service," a vulnerable Java application. The focus was on identifying and documenting Command Injection and Server-Side Request Forgery (SSRF) vulnerabilities.

Tools Used:

Docker:

- Purpose: Containerization tool used to package the Vulnerable Java application and its dependencies.
- Explanation: Docker enables the creation of isolated containers, ensuring consistency across different environments.

Trivy:

- Purpose: Trivy is utilized in this project for vulnerability scanning within the Docker containers.
- Explanation: By running Trivy on your container images, it analyzes the installed packages and compares them against known vulnerability databases.

Snyk:

- Purpose: Snyk is employed in this project to address security concerns by identifying and mitigating vulnerabilities in both the Docker container images and project dependencies.
- Explanation: The tool contributes to the overall goal of creating a secure and resilient environment for the Damn Vulnerable WordPress site and its supporting infrastructure.

Chaos Engineering:

- Purpose: To introduce controlled chaos and observe system behavior.
- Explanation: Chaos experiments are designed to uncover vulnerabilities, improve system resilience, and enhance overall reliability.

Implementation:

1. **Deployment:**

- The vulnerable Java application was deployed locally using the provided Docker command:
  docker run --rm -p 8000:8000 ghcr.io/datadog/vulnerable-java-application

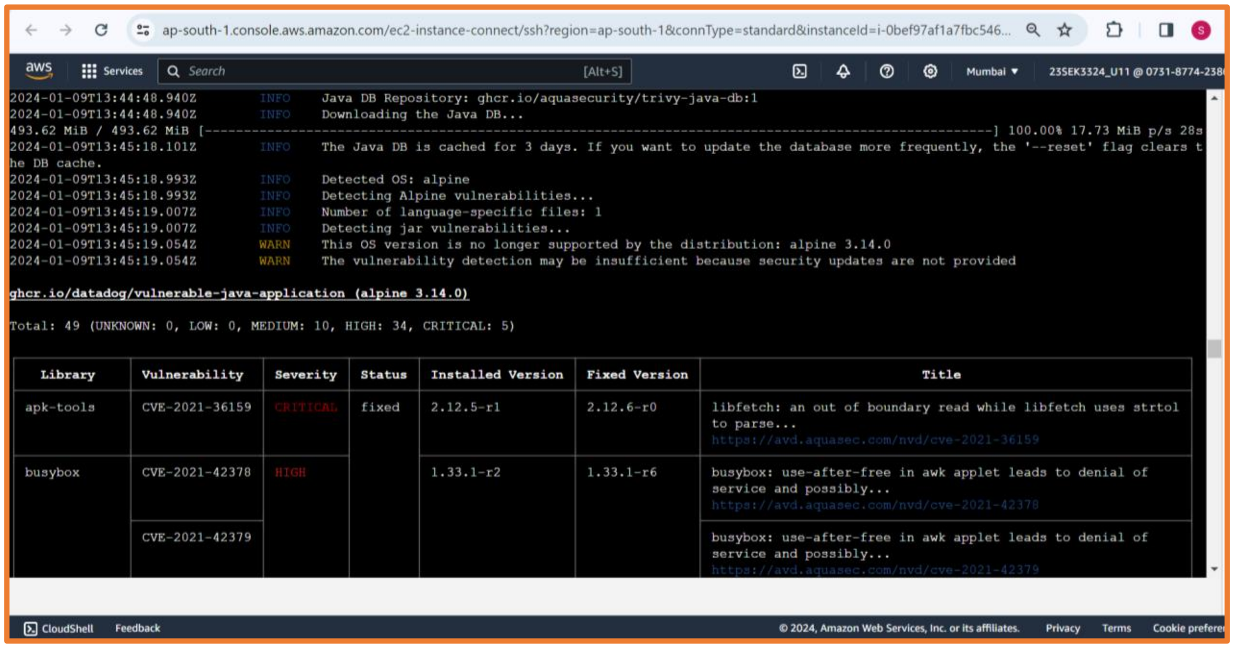
- Access to the web application was achieved at http://localhost:8000.

2. Vulnerability Scanning with Trivy:

- **Command:**
  trivy image gher.io/datadog/vulnerable-java-application

- **Explanation:** Integrate Trivy into the Docker workflow to scan container images for vulnerabilities. Identify and analyze potential security issues within the Dockerized environment.

Number of Vulnerabilities: TOTAL: 49 (UNKOWN: 0, LOW: 0, MEDIUM: 10, HIGH: 34, CRITICAL: 5)



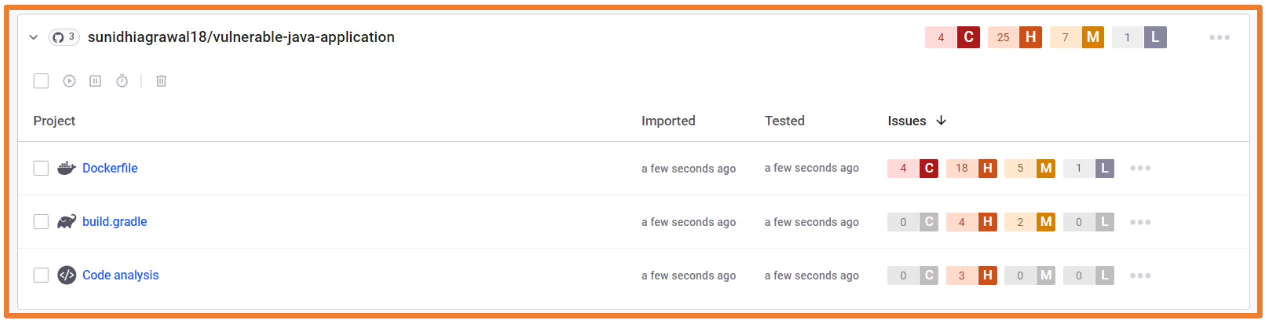
ap-south-1.console.aws.amazon.com/ec2-instance-connect/ssh?region=ap-south-1&connType=standard&instanceId=i-0bef97af1a7fbc546...
aws
Services
Search
[Alt+S]
Mumbai
23SEK3324_U11 @ 0731-8774-238

```
2024-01-09T13:44:48.940Z        INFO    Java DB Repository: ghcr.io/aquasecurity/trivy-java-db:1
2024-01-09T13:44:48.940Z        INFO    Downloading the Java DB...
493.62 MiB / 493.62 MiB [-----------------------------------------------------------------------------------] 100.00% 17.73 MiB p/s 28s
2024-01-09T13:45:18.101Z        INFO    The Java DB is cached for 3 days. If you want to update the database more frequently, the '--reset' flag clears t
he DB cache.
2024-01-09T13:45:18.993Z        INFO    Detected OS: alpine
2024-01-09T13:45:18.993Z        INFO    Detecting Alpine vulnerabilities...
2024-01-09T13:45:19.007Z        INFO    Number of language-specific files: 1
2024-01-09T13:45:19.007Z        INFO    Detecting jar vulnerabilities...
2024-01-09T13:45:19.054Z        WARN    This OS version is no longer supported by the distribution: alpine 3.14.0
2024-01-09T13:45:19.054Z        WARN    The vulnerability detection may be insufficient because security updates are not provided

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

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

| Library   | Vulnerability  | Severity | Status | Installed Version | Fixed Version | Title                                                  |
|-----------|----------------|----------|--------|-------------------|---------------|--------------------------------------------------------|
| apk-tools | CVE-2021-36159 | CRITICAL | fixed  | 2.12.5-r1         | 2.12.6-r0     | libfetch: an out of boundary read while libfetch uses strtol |
|           |                |          |        |                   |               | to parse...                                            |
|           |                |          |        |                   |               | https://avd.aquasec.com/nvd/cve-2021-36159             |
|-----------|----------------|----------|        |-------------------|---------------|--------------------------------------------------------|
| busybox   | CVE-2021-42378 | HIGH     |        | 1.33.1-r2         | 1.33.1-r6     | busybox: use-after-free in awk applet leads to denial of |
|           |                |          |        |                   |               | service and possibly...                                |
|           |                |          |        |                   |               | https://avd.aquasec.com/nvd/cve-2021-42378             |
|           |----------------|          |        |                   |               |--------------------------------------------------------|
|           | CVE-2021-42379 |          |        |                   |               | busybox: use-after-free in awk applet leads to denial of |
|           |                |          |        |                   |               | service and possibly...                                |
|           |                |          |        |                   |               | https://avd.aquasec.com/nvd/cve-2021-42379             |
```

Figure 3: trivy image gher.io/datadog/vulnerable-java-application

3. **Security Analysis with Snyk:**

- **Explanation:** Utilize Snyk to scan project dependencies and Docker container images for known vulnerabilities. Receive continuous monitoring alerts and remediation guidance to enhance security.



sunidhiagrawal18/vulnerable-java-application 4 C 25 H 7 M 1 L

Project	Imported	Tested	Issues
Dockerfile	a few seconds ago	a few seconds ago	4 C 18 H 5 M 1 L
build.gradle	a few seconds ago	a few seconds ago	0 C 4 H 2 M 0 L
Code analysis	a few seconds ago	a few seconds ago	0 C 3 H 0 M 0 L

Figure 4: Snyk Vulnerability scan of Vulnerable Java application

Observation and Analysis:

Vulnerabilities Identified in the Project:

- openssl/libcrypto1.1 Buffer Overflow - CVE-2021-3711

- This vulnerability is associated with the EVP_PKEY_decrypt() function used for decrypting SM2 encrypted data.
- An application calling this function is expected to do so twice, and a buffer overflow vulnerability in this process can lead to security issues.

- zlib/zlib Out-of-bounds Write - CVE-2022-37434:

- The zlib library version 1.2.12 and earlier has a vulnerability in the inflate function.
- The issue arises from a heap-based buffer over-read or buffer overflow when dealing with a large gzip header extra field.

- apk-tools/apk-tools Out-of-bounds Read - CVE-2021-36159:

- The vulnerability is in libfetch, used in apk-tools, xbps, and other products.
- It involves mishandling numeric strings for the FTP and HTTP protocols, leading to an out-of-bounds read in the FTP passive mode implementation.

- libretls/libretls Loop with Unreachable Exit Condition ('Infinite Loop') - CVE-2022-0778:

- The vulnerability is related to the BN_mod_sqrt() function in libretls, which computes a modular square root.
- The bug causes the function to loop forever for non-prime moduli due to an issue with the exit condition.

- openssl/libcrypto1.1 Double Free - CVE-2022-4450:

- This vulnerability is associated with the PEM_read_bio_ex() function in OpenSSL.
- The function reads a PEM file from a BIO and parses and decodes data, and a double-free vulnerability can occur during this process.

In response to these vulnerabilities:

- It is crucial to keep software libraries and dependencies up to date to apply patches and security fixes.
- Developers should be aware of the specific security recommendations provided by the maintainers of the affected libraries.
- Affected systems should be patched as soon as possible to mitigate the risk of exploitation.

Conclusion:

In conclusion, this project provided a comprehensive exploration of the Vulnerable Java Application - Website Tester Service, focusing on deploying, identifying, and addressing security vulnerabilities. Through intentional chaos engineering experiments and user interactions, we gained valuable insights into the application's resilience, user experience, and potential risks. The implemented security measures showcased their effectiveness in mitigating known vulnerabilities. Moving forward, continuous updates and user feedback will be integral for refining the educational experience and enhancing the application's security posture. Overall, this project contributes to fostering a deeper understanding of web application security, resilience, and the importance of proactive measures in securing online services.