|  |  |
| --- | --- |
| Name: | BHUMIKA GUPTA |
| Lab User ID: | 23SEK3324\_U13 |
| Date: | January 10, 2024 |
| Application Name: | Vulnerable Java application |

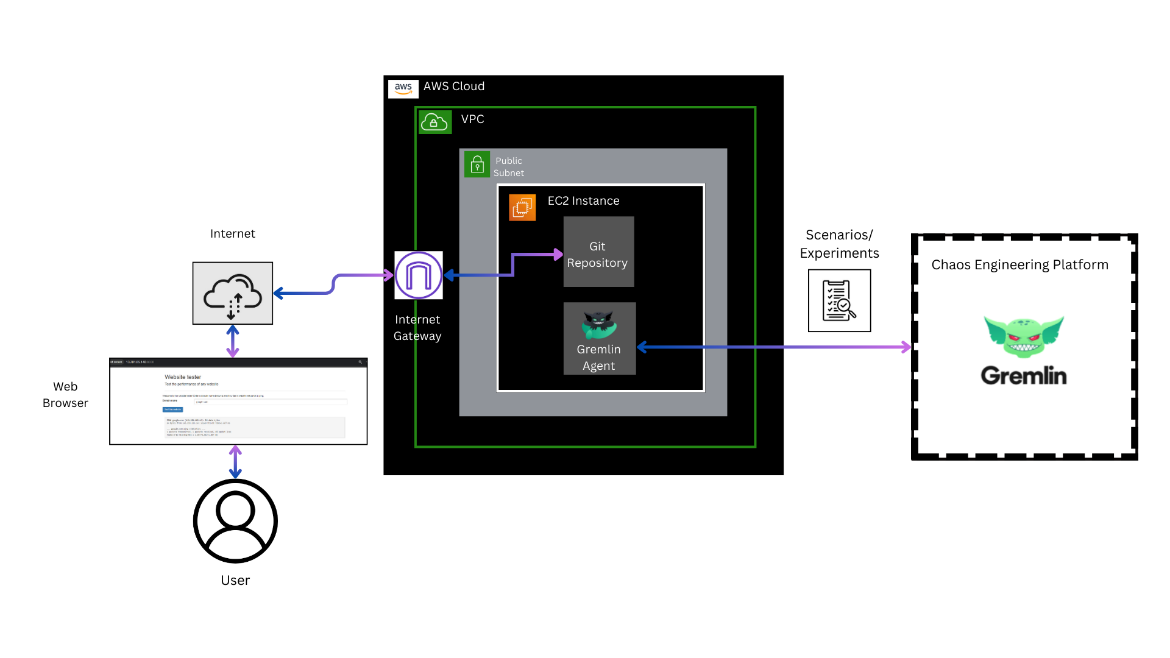
**Follow the below guidelines:**





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)



**Fig: System Architecture**

**System architecture** is a conceptual model that describes the structure and behavior of multiple components and subsystems like multiple software applications, network devices, hardware, and even other machinery of a system.

**Explanation of the above System Architecture-**

1. AWS Cloud: The entire infrastructure is hosted on AWS.
2. VPC (Virtual Private Cloud): The VPC serves as an isolated network within the AWS Cloud. It allows you to logically isolate resources and control network settings.
3. Public Subnet: Within the VPC, there is a public subnet. Public subnets are accessible from the internet and typically host resources like web servers.
4. EC2 Instance: An EC2 instance is launched in the public subnet.
   * + **Scenario 1:** *Web Application Deployment*
       - This instance hosts the web application.
       - Git Repository: The Git repository holds the source code and configurations for the "Vulnerable Java application" application.
       - Internet Gateway: An Internet Gateway allows communication between the VPC and the internet. In this case, it enables users to access the Vulnerable Java application from a web browser.
       - Web Browser (User): Users interact with the Vulnerable Java application through a web browser, connecting to the public IP address or domain associated with the EC2 instance.
     + **Scenario 2:** *Gremlin Chaos Engineering Experiment*
       - This instance hosts the Gremlin Agent.
       - Gremlin Agent: The Gremlin Agent is a software component installed on the EC2 instance. It facilitates chaos engineering experiments by injecting faults into the system.
       - Gremlin Chaos Engineering Platform: The Gremlin Agent connects to the Gremlin Chaos Engineering Platform. This platform allows you to perform controlled experiments, such as shutting down the EC2 instance.

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)

**Steady State** – The steady state is the stable and expected state where the system functions smoothly without disruptions.

**System’s Normal Behavior -** It involves the consistent interactions between various components, resulting in reliable performance and desired outcomes. Measurable outputs and observed behaviors during steady state operations define the system's normal behavior

So the steady state and measurable outcome of our system is as follows:-

1. **Web Application Deployment:** Users can successfully access the Vulnerable Java web application hosted on the EC2 instance.
   * Measurable Outputs:

* Consistent availability of the Vulnerable Java application web interface.
* Expected and reliable responses to user interactions within the application.
* Stable and predictable resource utilization on the EC2 instance.

1. **Gremlin Chaos Engineering Experiment:** The Gremlin Agent operates without disruptions. Controlled chaos experiments, such as shutting down the EC2 instance, do not occur under normal circumstances.
   * Measurable Outputs:
     + Stable connectivity between the Gremlin Agent and the Gremlin Chaos Engineering Platform.
     + No unexpected interruptions or failures during Gremlin experiments.

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**

Specific command injection is simulated

Predictable scenarios where the system's response is well-understood and expected

**Unknown**

**Unknown**

**Known**

Sudden increase in user traffic

Unanticipated system failure occurs

These hypothesis provide a structured approach to defining and understanding the expected and unexpected behaviors of the Vulnerable Java application.

By categorizing scenarios into knowns and unknowns, and actions into intentional and unintentional, the hypothesis aid in identifying, responding to, and improving the security and robustness of the system.

**Knowns-Knowns Hypothesis:**

* Predictable scenarios where the system's response is well-understood and expected.

**Known-Unknown Hypothesis:**

* Controlled experiments with a known action (injection) but an uncertain outcome, revealing potential vulnerabilities.

**Unknown-Known Hypothesis:**

* Situations with known triggers (increased traffic) but uncertain consequences, requiring monitoring for correlations.

**Unknown-Unknown Hypothesis:**

* In Unforeseen events with both triggers and outcomes not known in advance, emphasizing the need for robust monitoring and incident response strategies.

Experiment:

(Document your Preparation, Implementation, Observation and Analysis )

**Objective:**

The objective is to assess and enhance the security and resilience of the deployed web application ("Vulnerable Java application") within the AWS environment. This involves identifying vulnerabilities, conducting controlled chaos experiments using Gremlin, and implementing security measures to address the detected issues.

**Context:**

Deploying in AWS Cloud with a VPC, public subnet, EC2 instance hosting the vulnerable web application, Git repository for version control, and Internet Gateway for external access.

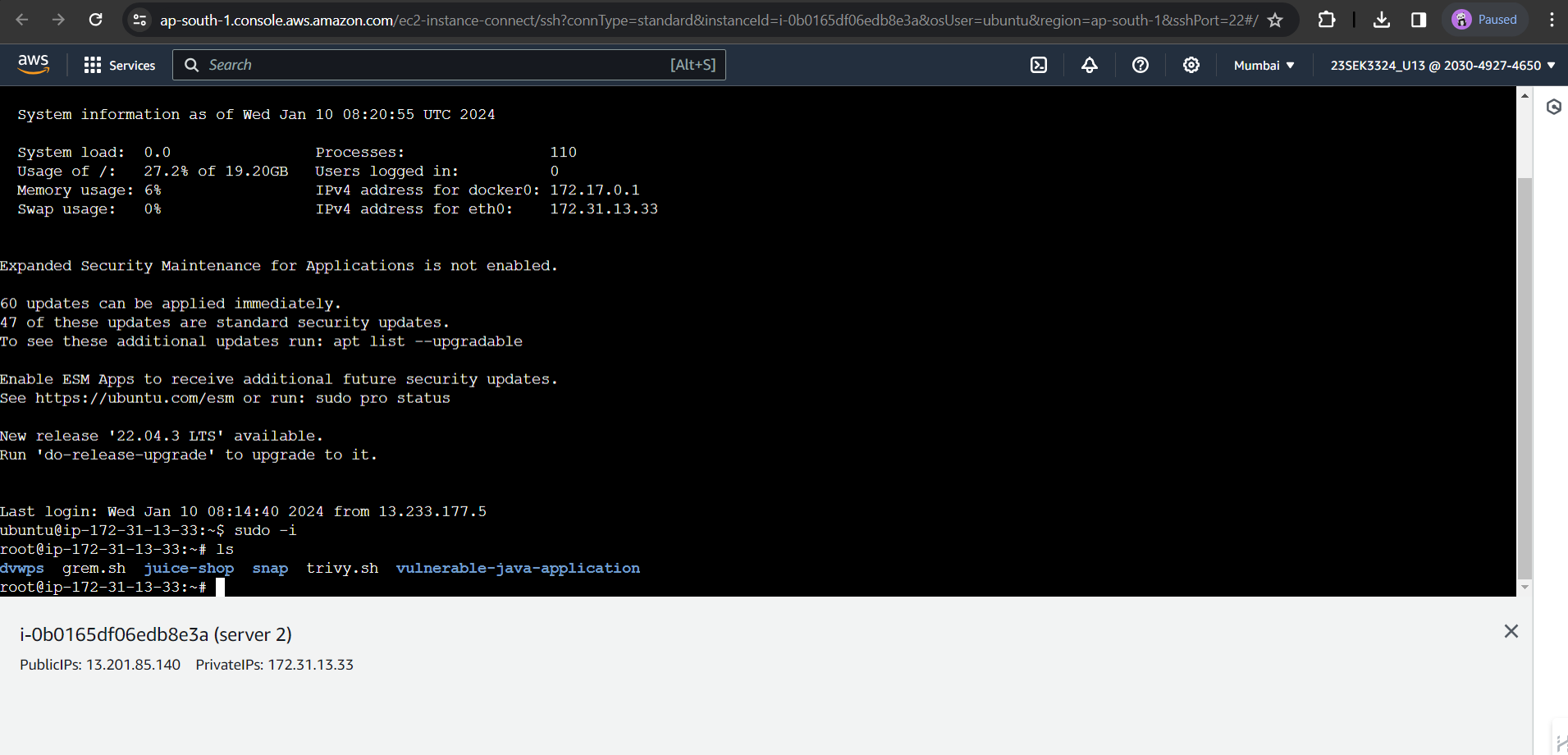
Utilizing Gremlin for controlled chaos engineering experiments to simulate disruptions and test system resilience.

Identified vulnerabilities in the web application using Trivy, specifically focusing on Command Injection and Server-Side Request Forgery (SSRF).

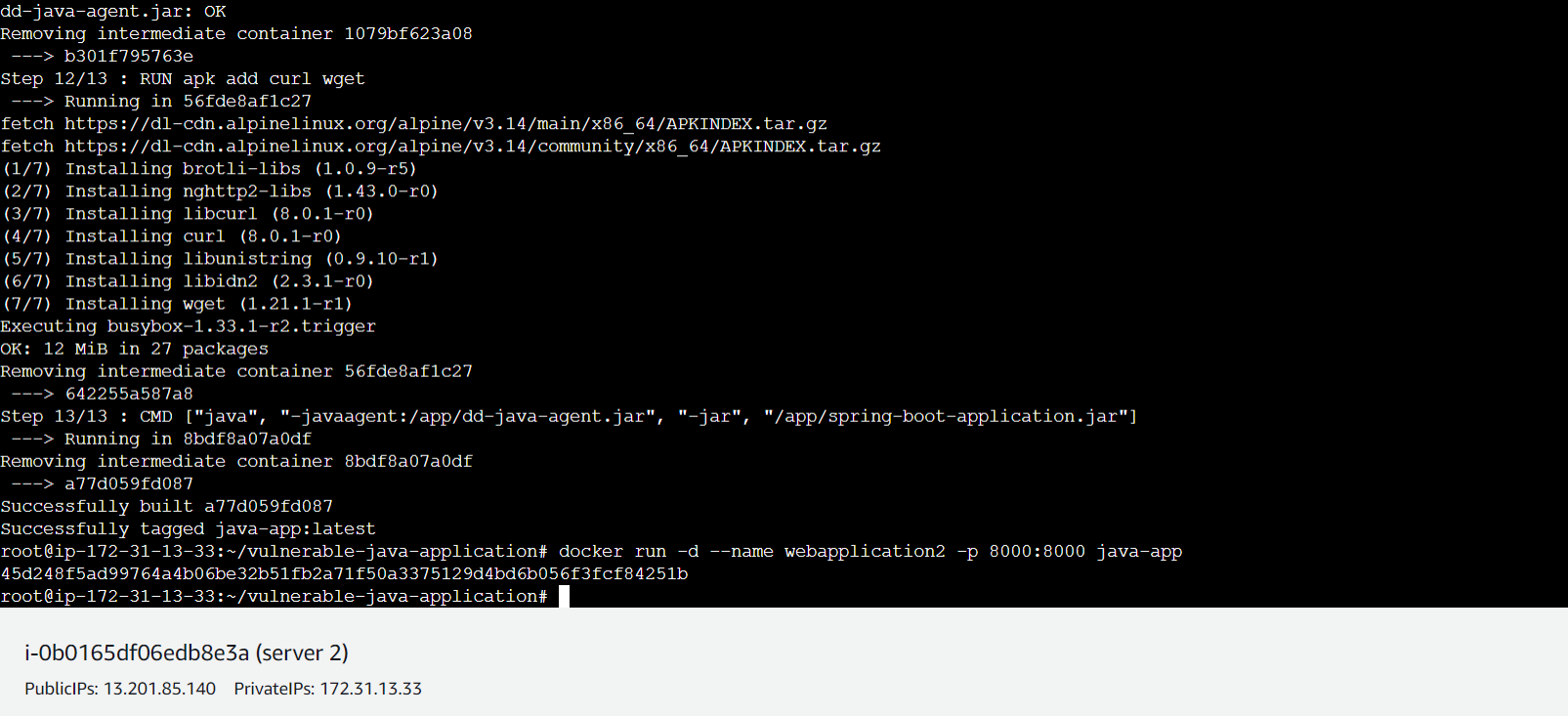
This experiment is organized into four main sections:

**Preparation:**

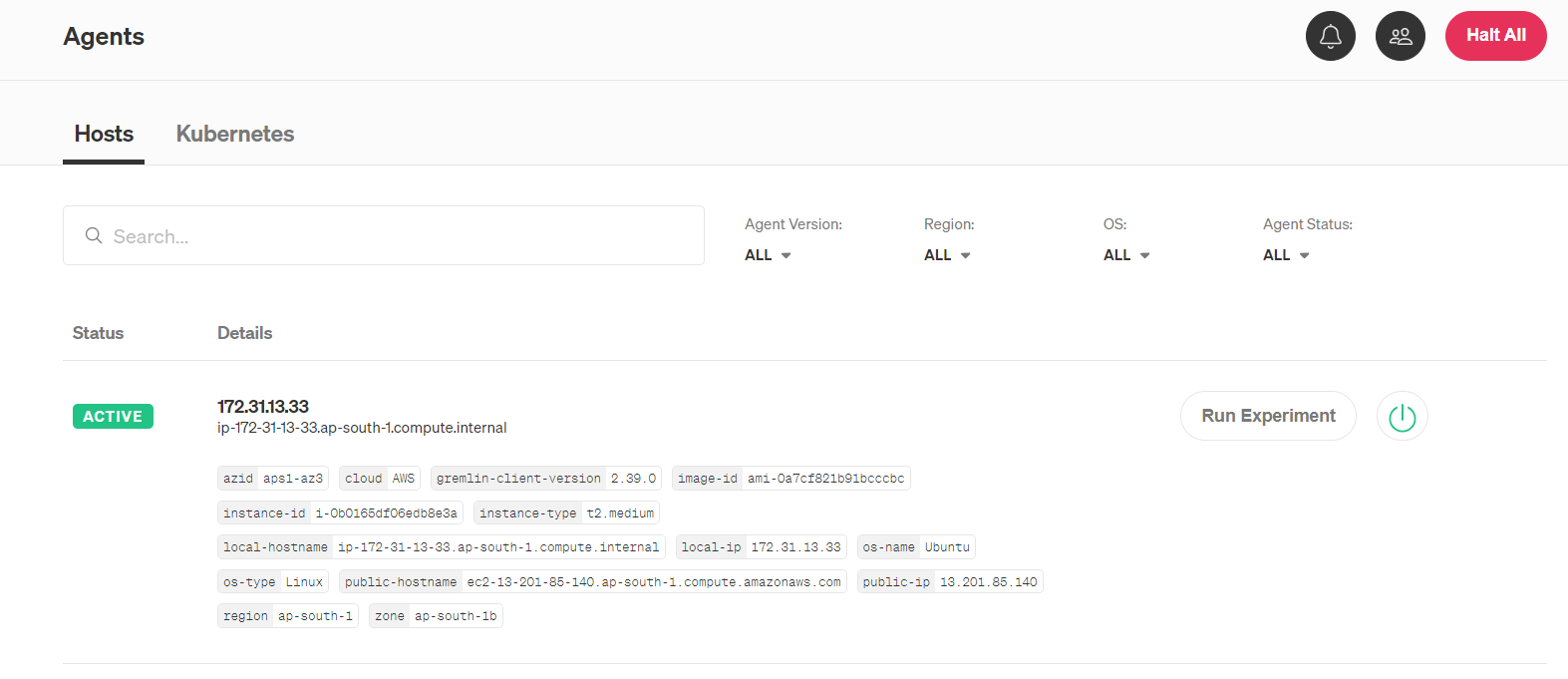
1. **AWS Infrastructure Setup:**
   * Launched an EC2 instance within an AWS Virtual Private Cloud (VPC).
   * Configured a public subnet to enable internet connectivity.
   * Connected the VPC to an Internet Gateway, allowing communication with the internet.



1. **Application Deployment:**
   * Installed a Git repository containing the Vulnerable Java application on the EC2 instance.
   * Ensured the web application is accessible via a web browser over the internet.



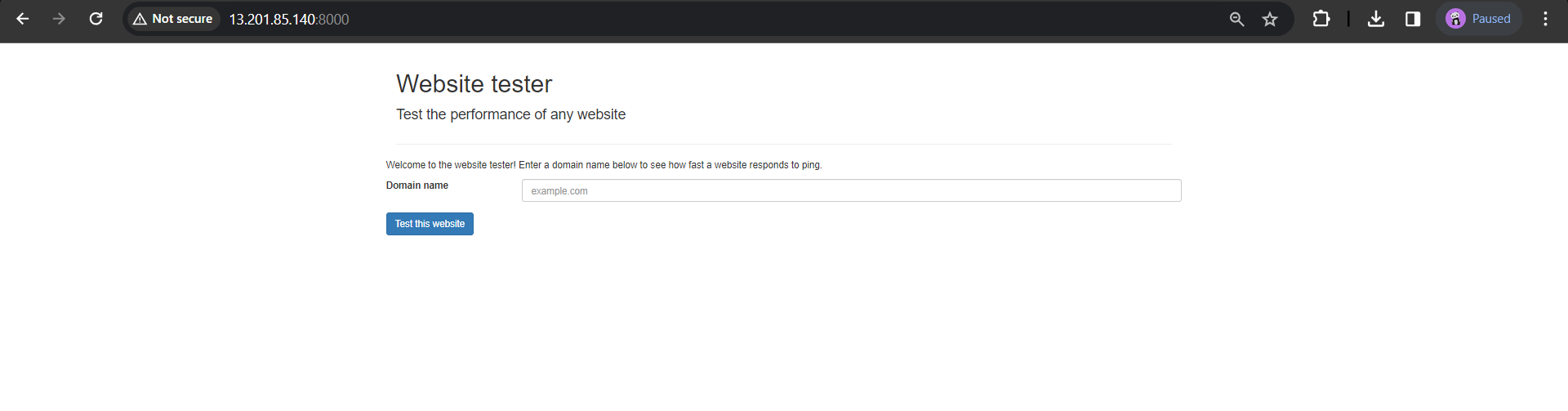
1. **Chaos Engineering Setup:**
   * Deployed a Gremlin Agent on a separate EC2 instance within the same public subnet.
   * Established connectivity between the Gremlin Agent and the Gremlin Chaos Engineering Platform.

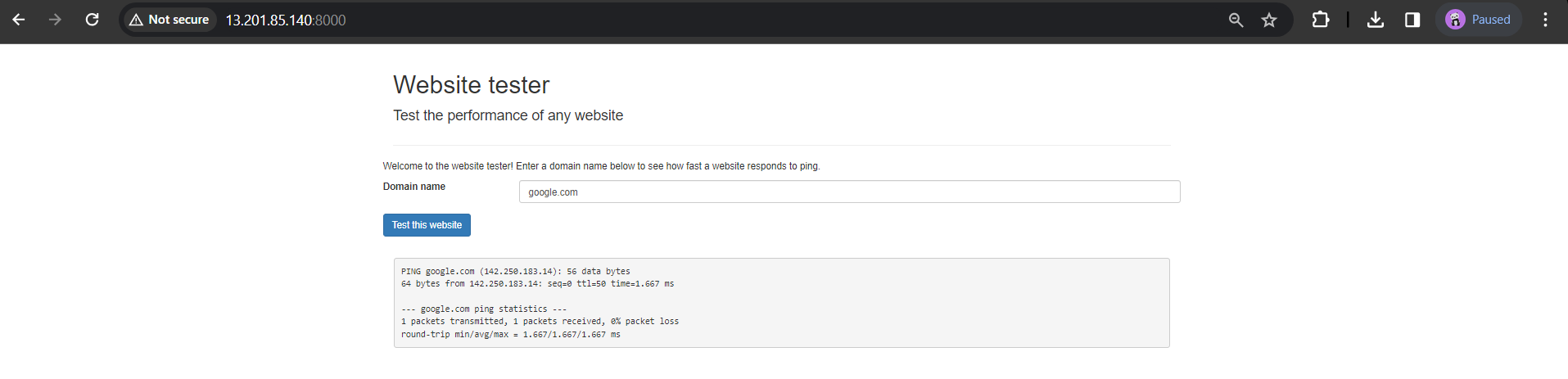


1. **Vulnerability Scanning:**
   * Install Trivy on the EC2 instance hosting the vulnerable Vulnerable Java application.
   * Configure Trivy to scan the Docker image regularly for vulnerabilities.

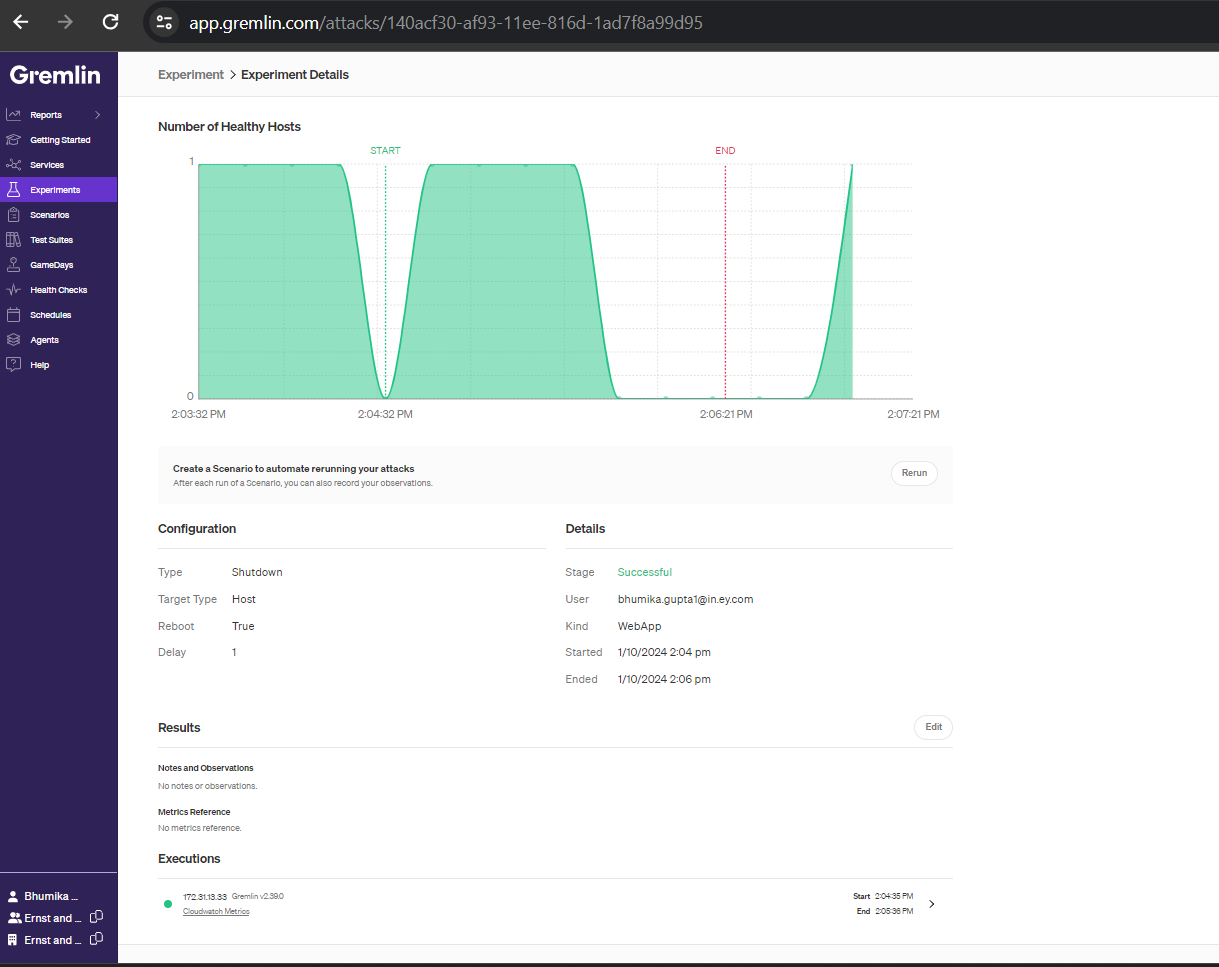
**Implementation:**

1. **Web Application Usage:**
   * Ensure the Docker container with the Vulnerable Java application is running on the EC2 instance.
   * Monitor the Git repository for version control and manage changes to the Vulnerable Java application and configurations.

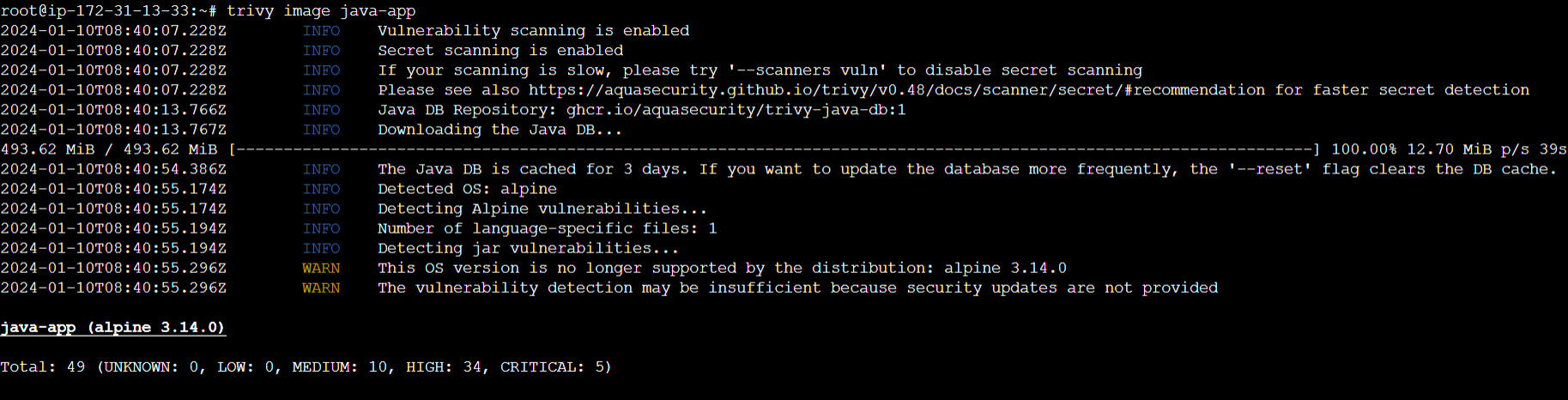




1. **Chaos Engineering Experiment:**
   * Conducted controlled chaos experiments using Gremlin, such as simulating a shutdown scenario, to evaluate system resilience.
   * Monitored system behaviour during Gremlin experiments to observe how the Vulnerable Java application handles disruptions.



1. **Vulnerability Mitigation:**
   * Utilized Trivy to scan the OWASP Juice Shop codebase for vulnerabilities.
   * Analysed Trivy scan results to identify and understand potential security flaws within the application.



|  |  |  |
| --- | --- | --- |
| **Vulnerability** | **Severity** | **Weakness** |
| CVE-2021-36159 | CRITICAL | The product reads data past the end, or before the beginning, of the intended buffer. |
| CVE-2021-42378 | HIGH | Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code. |
| CVE-2021-42374 | MEDIUM | The product reads data past the end, or before the beginning, of the intended buffer. |
| CVE-2022-28391 | MEDIUM | BusyBox through 1.35.0 allows remote attackers to execute arbitrary code if netstat is used to print a DNS PTR records value to a VT compatible terminal. Alternatively, the attacker could choose to change the terminals colors. |

**Observation:**

1. **Web Application Deployment:**
   * Observe user access to the Vulnerable Java application through a web browser.
   * Monitor the EC2 instance for stability and resource utilization during normal usage.
2. **Chaos Engineering Configuration:**
   * During Gremlin shutdown experiments, observed the system's behaviour, focusing on downtime, recovery time, and impact on user access.
   * Documented any unexpected behaviour or failures during the chaos experiments.
3. **Vulnerability Scanning:**
   * Trivy scan results revealed vulnerabilities present in the Vulnerable Java application codebase.
   * Categorized vulnerabilities based on severity levels and identified potential areas of improvement.

**Analysis:**

1. **Web Application Stability:**
   * Analyse user access logs and EC2 instance monitoring data to ensure the system's stability and performance.
   * Identify any unexpected issues or resource bottlenecks during normal operation.
2. **Chaos** **Experiment Outcomes:**
   * Analysed the impact of simulated shutdowns on the Vulnerable Java application.
   * Assessed the system's ability to recover and maintain functionality after the chaos experiment.
3. **Vulnerability Analysis:**
   * Analyse Trivy scan reports to identify and prioritize vulnerabilities.
   * Implement security measures based on the severity of vulnerabilities discovered.
   * Evaluate the effectiveness of vulnerability mitigation measures over time.