|  |  |
| --- | --- |
| Name: | Rishitha Pudi |
| Lab User ID: | 23SEK3324\_U01 |
| Date: | 09-01-2024 |
| Application Name: | VulnerableJavaWebApplication |

**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)

Ec2 instance

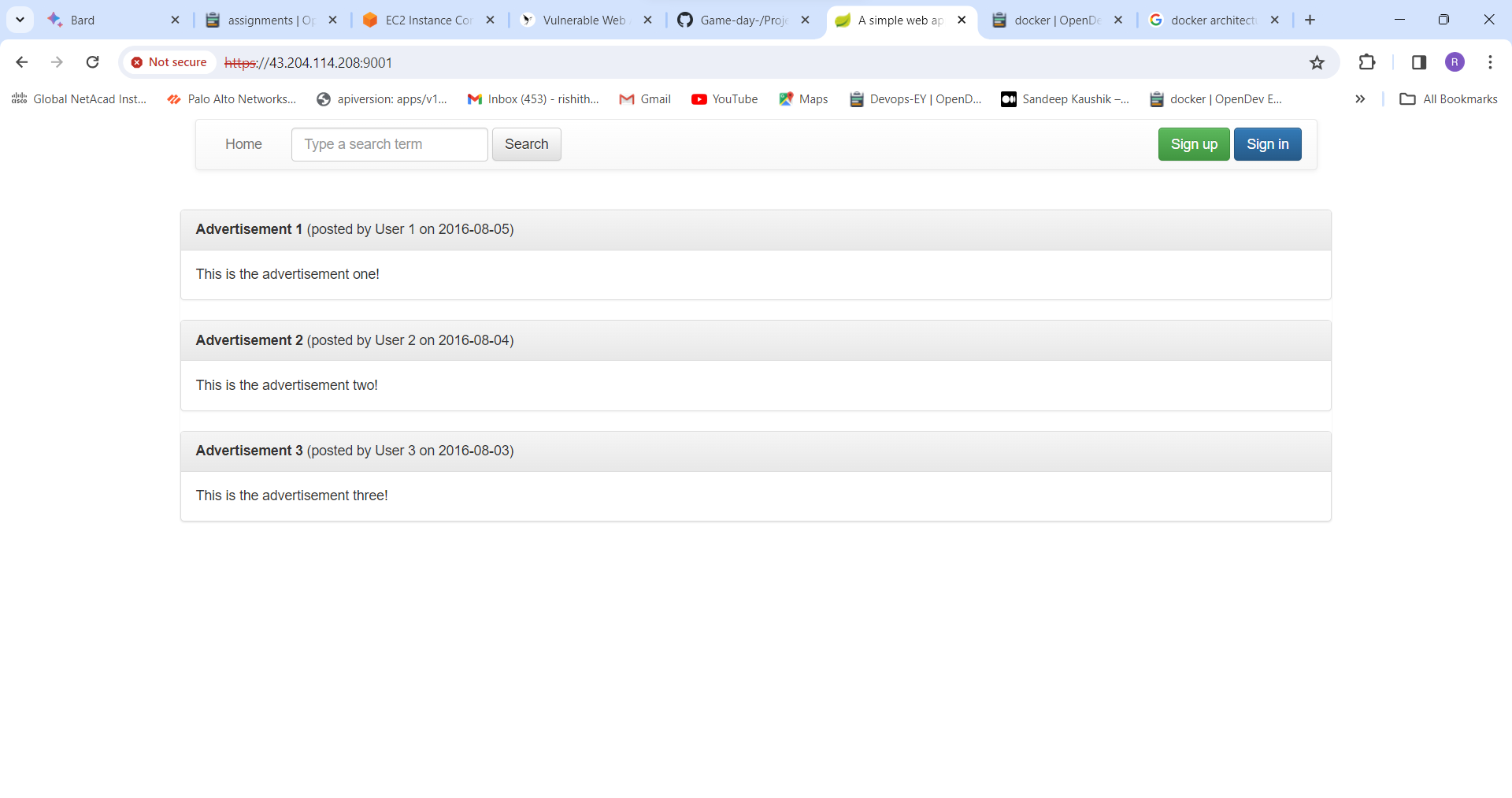
Docker

container

9001:9000

43.204.114.208

https:// 43.204.114.208:9001



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)

This is an simple and self-contained Java web application with security flaws

The application uses Spring Boot and an embedded H2 database that resets every time it starts.

If you break it just restart and everything will be reset.

The application will run on **HTTPS port 9001**

The application uses Spring Boot and an embedded H2 database that resets every time it starts. If you break it just restart and everything will be reset.

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.")



Chaos engineering hypothesis scenarios

**Known**

Simulate the component failure to test recovery procedures, alerting systems, and failover mechanisms. Verify known behavior and response times, but also look for unforeseen bottlenecks or delays

Simulate the event (e.g., controlled power outage) to observe the system's behavior and identify unexpected vulnerabilities or dependencies. You might discover unknown data corruption issues or hidden communication failures.

942236

**Unknown**

Introduce controlled chaos through fuzzing (injecting random data or actions) or stress testing (applying extreme load). Monitor for emergent issues and unexpected behaviors, focusing on identifying entirely new failure modes or vulnerabilities.

 Introduce unexpected data or actions related to the known functionality (e.g., large cache invalidation, invalid cache entries). Monitor for emergent issues and unexpected system behavior.

**Unknown**

**Known**

Experiment:

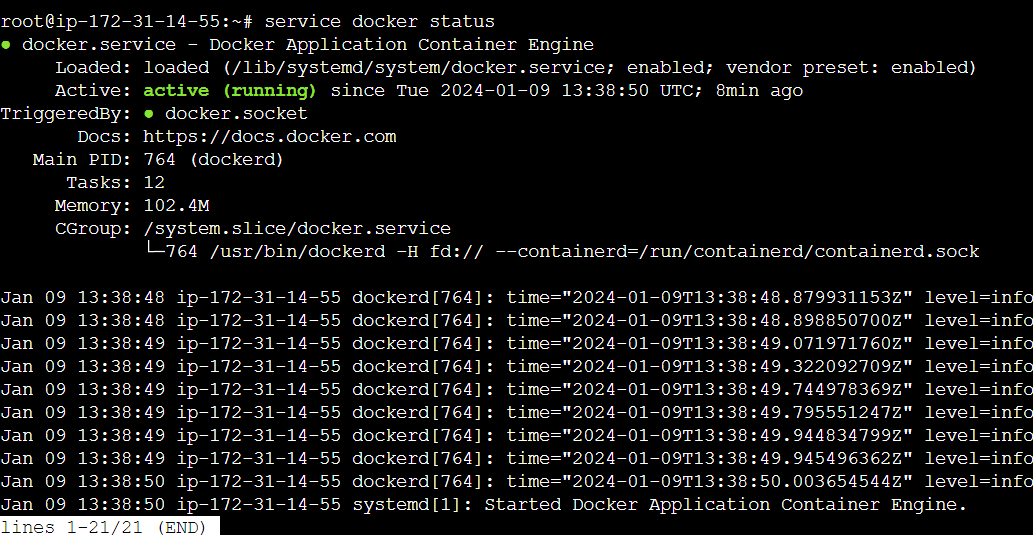
(Document your Preparation, Implementation, Observation and Analysis )

# Overview: Vulnerable Java Web Application is java based vulnerable web application we are

Performing server tools against this application to know its behavior.

Methodology:

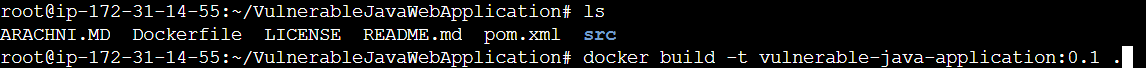
**Step 1 :** create an Ec2 instance , install Docker to that virtual machine. Make sure Docker is running.



**Step 2:** clone that git repo into machine

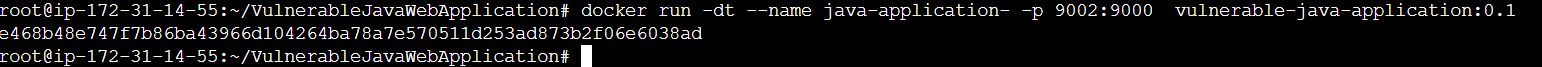


**Step 3:** build an Docker image using the Docker file in the repo

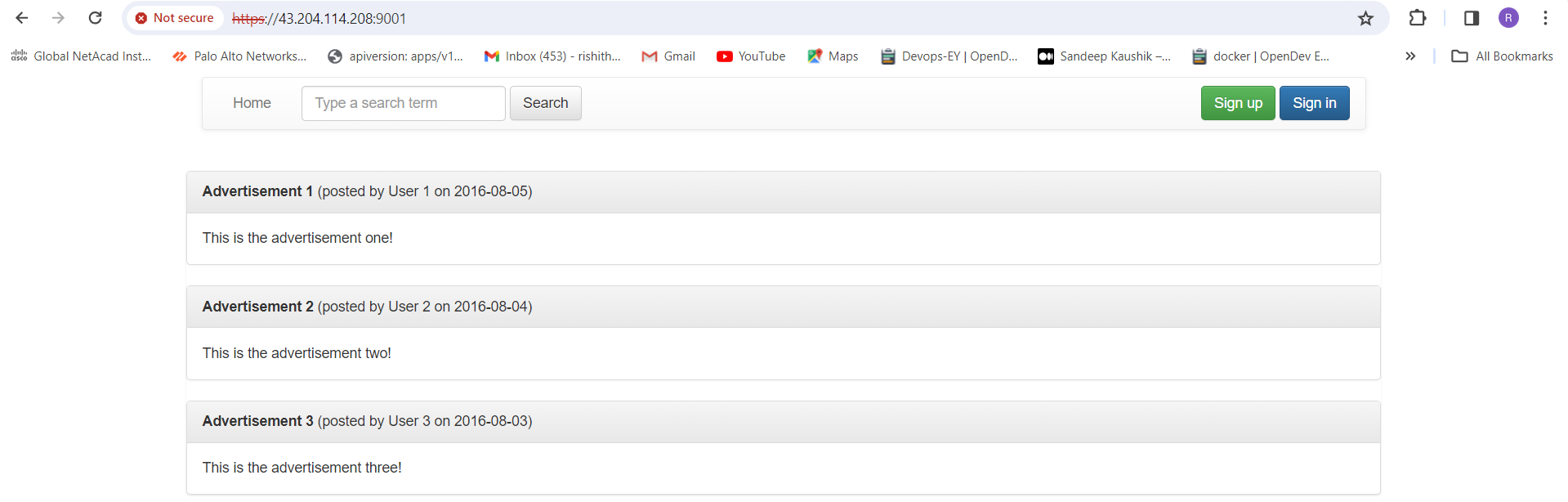




**Step 4:** create a container for the image we build



**Step 5:** live the application on the browser https://<ipaddress>:9002/



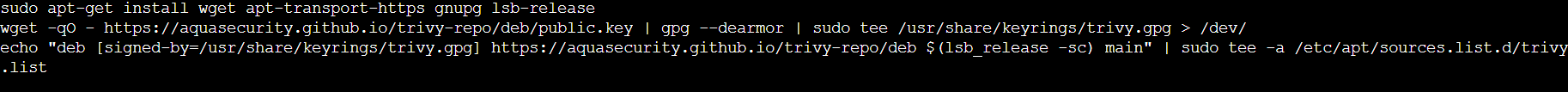
**Tools :** the follow are the tools using in this project

1. Trivy
2. OWASP ZEP
3. synk

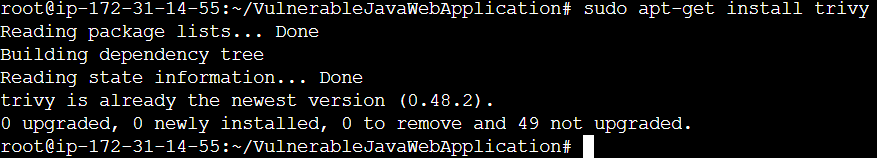
**Observation:**

**Analysis 1:** using Trivy

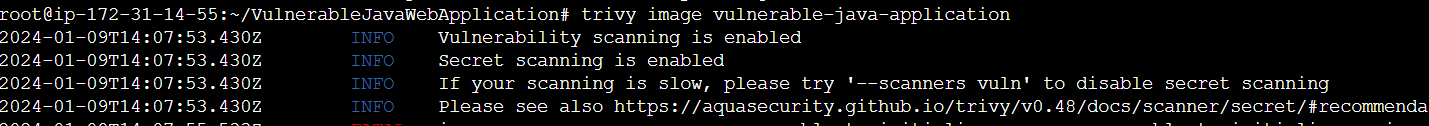
First set up the trivy in the instance by using following commands



After install trivy



Using trivy we can perform scanning on the Docker images

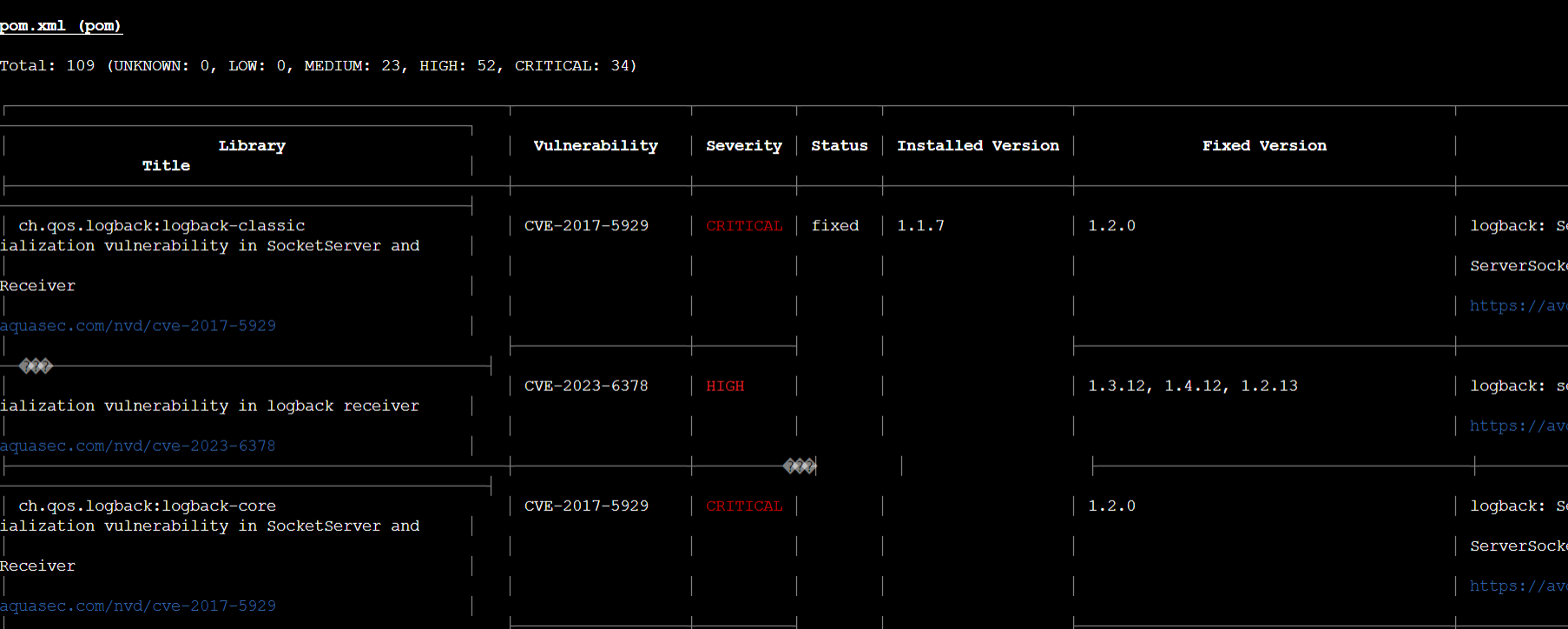


Find 109 vulnerability in the java application

Unknown: 0 , Low : 0

Medium: 23

High: 52, critical: 34



The following are some high vulnrabilies given by trivy

**Deserilization of Data:** The product desterilizes untrusted data without sufficiently verifying that the resulting data will be valid.

**Solution:** Make fields transient to protect them from deserialization. An attempt to serialize and then desterilize a class containing transient fields will result in NULLs where the transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly

**Incomplete List of Disallowed Inputs**: FasterXML jackson-databind through 2.8.11 and 2.9.x through 2.9.3 allows unauthenticated remote code execution because of an incomplete fix for the CVE-2017-7525 and CVE-2017-17485 deserialization flaws. This is exploitable via two different gadgets that bypass a blacklist.

**Solution:** The product implements a protection mechanism that relies on a list of inputs (or properties of inputs) that are not allowed by policy or otherwise require other action to neutralize before additional processing takes place, but the list is incomplete, leading to resultant weaknesses.

**Denial-Of-Service:** A serialization vulnerability in log back receiver component part of log back version 1.4.11 allows an attacker to mount a Denial-Of-Service attack by sending poisoned data.

**Solution:** Make fields transient to protect them from deserialization. An attempt to serialize and then desterilize a class containing transient fields will result in NULLs where the transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly.

**Deserialization of Untrusted Data**: The product desterilizes untrusted data without sufficiently verifying that the resulting data will be valid.

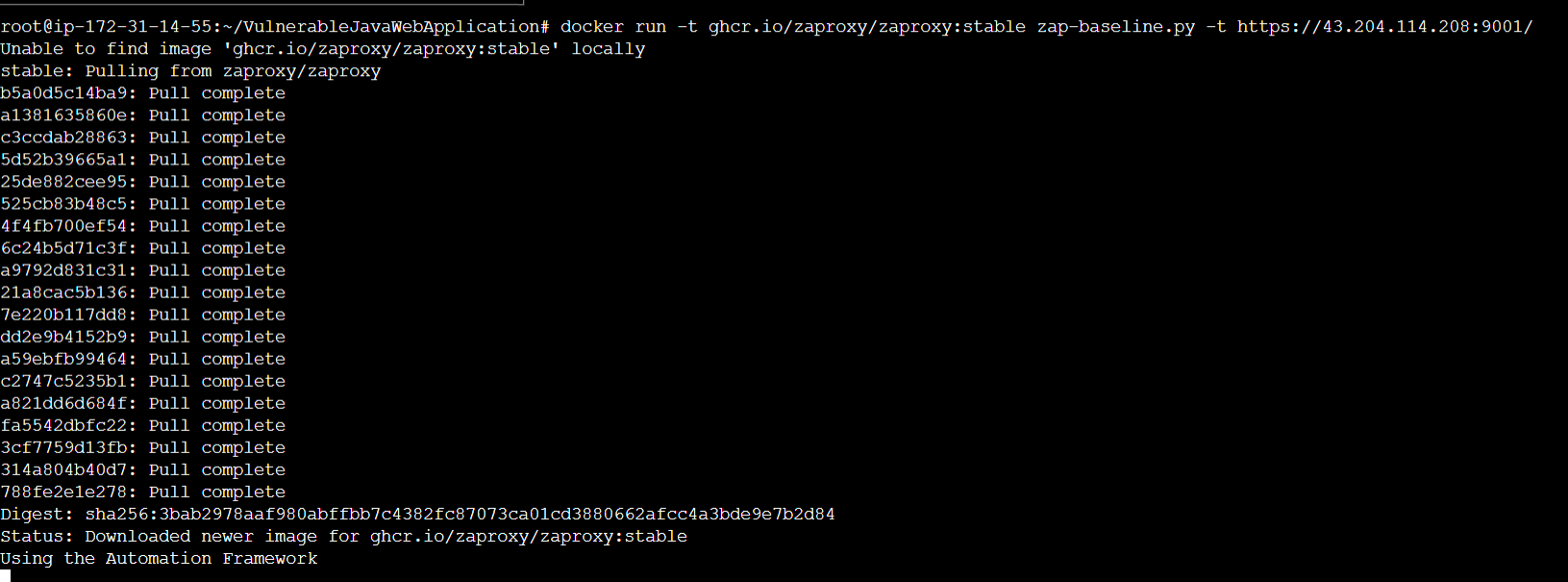
**Solution:** Data that is untrusted cannot be trusted to be well-formed. When developers place no restrictions on “gadget chains,” or series of instances and method invocations that can self-execute during the deserialization process (i.e., before the object is returned to the caller), it is sometimes possible for attackers to leverage them to perform unauthorized actions, like generating a shell.

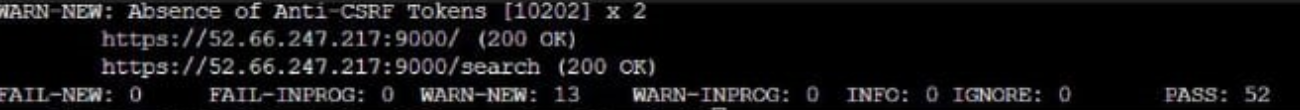
**Improper Neutralization of Argument Delimiters in a Command ('Argument Injection'):** The product constructs a string for a command to be executed by a separate component in another control sphere, but it does not properly delimit the intended arguments, options, or switches within that command string

**solution**: Assume all input is malicious. Use an “accept known good” input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Consider performing repeated canonicalization until your input does not change any more. This will avoid double-decoding and similar scenarios, but it might inadvertently modify inputs that are allowed to contain properly-encoded dangerous content.

**Analysis 2:** using OWASP ZAP

Using Docker run command , we can perform vulnerability scanning by using automation frame work





The following are vulnerabilities I find using ZAP

**Missing Anti-clickjacking Header:** The response does not include either Content-Security-Policy with ‘frame-ancestors’ directive or X-Frame-Options to protect against ‘Clickjacking’ attacks.

**Solution:** Modern Web browsers support the Content-Security-Policy and X-Frame-Options HTTP headers. Ensure one of them is set on all web pages returned by your site/app.

**X-Content-Type-Options Header Missing:** The Anti-MIME-Sniffing header X-Content-Type-Options was not set to ’no sniff’. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type.

**Solution:** Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'no sniff' for all web pages. If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME-sniffing.

**Strict-Transport-Security Header:** HTTP Strict Transport Security (HSTS) is a web security policy mechanism whereby a web server declares that complying user agents (such as a web browser) are to interact with it using only secure HTTPS connections (i.e. HTTP layered over TLS/SSL). HSTS is an IETF standards track protocol and is specified in RFC 6797.

**Solution:** Ensure that your web server, application server, load balancer, etc. is configured to enforce Strict-Transport-Security.

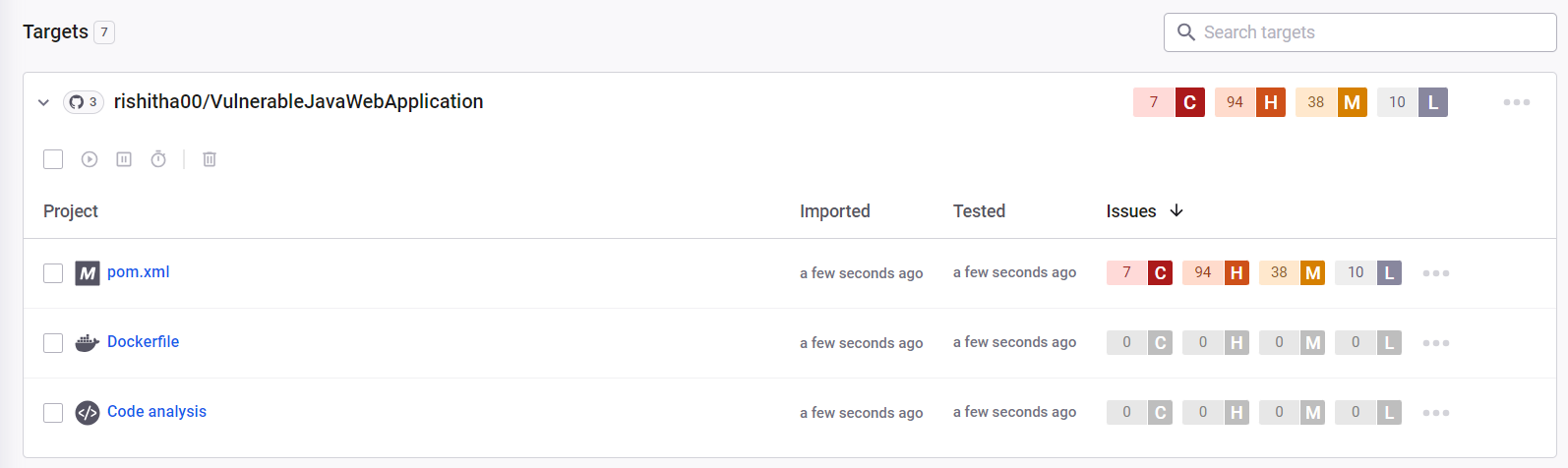
**Session Management Response Identified:** The given response has been identified as containing a session management token. The ‘Other Info’ field contains a set of header tokens that can be used in the Header Based Session Management Method. If the request is in a context which has a Session Management Method set to “Auto-Detect” then this rule will change the session management to use the tokens identified.

**Solution:** This is an informational alert rather than a vulnerability and so there is nothing to fix.

**Deprecated Feature Policy Header Set:** The header has now been renamed to Permissions-Policy.

**Solution:** Ensure that your web server, application server, load balancer, etc. is configured to set the Permissions-Policy header instead of the Feature-Policy header.

**Analysis 3:** using synk tool



**Deserialization of Untrusted Data**: The product desterilizes untrusted data without sufficiently verifying that the resulting data will be valid.

**Solution:** Avoid having unnecessary types or gadgets available that can be leveraged for malicious ends. This limits the potential for unintended or unauthorized types and gadgets to be leveraged by the attacker. Add only acceptable classes to an allow list. Note: new gadgets are constantly being discovered, so this alone is not a sufficient mitigation.

**Improper Control of Generation of Code ('Code Injection'):** The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.

**Solution:** Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force the program to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted

**Uncontrolled Resource Consumption:** The product does not properly control the allocation and maintenance of a limited resource, thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.

**Solution:** Design throttling mechanisms into the system architecture. The best protection is to limit the amount of resources that an unauthorized user can cause to be expended. A strong authentication and access control model will help prevent such attacks from occurring in the first place. The login application should be protected against DoS attacks as much as possible. Limiting the database access, perhaps by caching result sets, can help minimize the resources expended. To further limit the potential for a DoS attack, consider tracking the rate of requests received from users and blocking requests that exceed a defined rate threshold.

**Improper Input Validation:** The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.

**Solution:** Directly convert your input type into the expected data type, such as using a conversion function that translates a string into a number. After converting to the expected data type, ensure that the input's values fall within the expected range of allowable values and that multi-field consistencies are maintained.

**Code injection:** occurs when malicious code is inserted through user input, posing a threat to system security. Prevention involves reevaluating dynamic code execution, particularly in scenarios involving user input, and applying secure coding practices. Regularly updating the library is essential for mitigation.

**Solution:** To mitigate the arbitrary code execution vulnerability in "org.apache.tomcat.embed: tomcat-embed-core," promptly update to the latest secure version. Additionally, adopt secure coding practices, validate and sanitize user inputs, and limit dynamic code execution, especially in scenarios where user input is involved, to prevent code injection attacks.