Chaos Engineering – Hypothesis

Name:	Gowrav Kumar Baitharu Aripaka
Lab User ID:	23SEK3324_U02
Date:	11-01-2024
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:

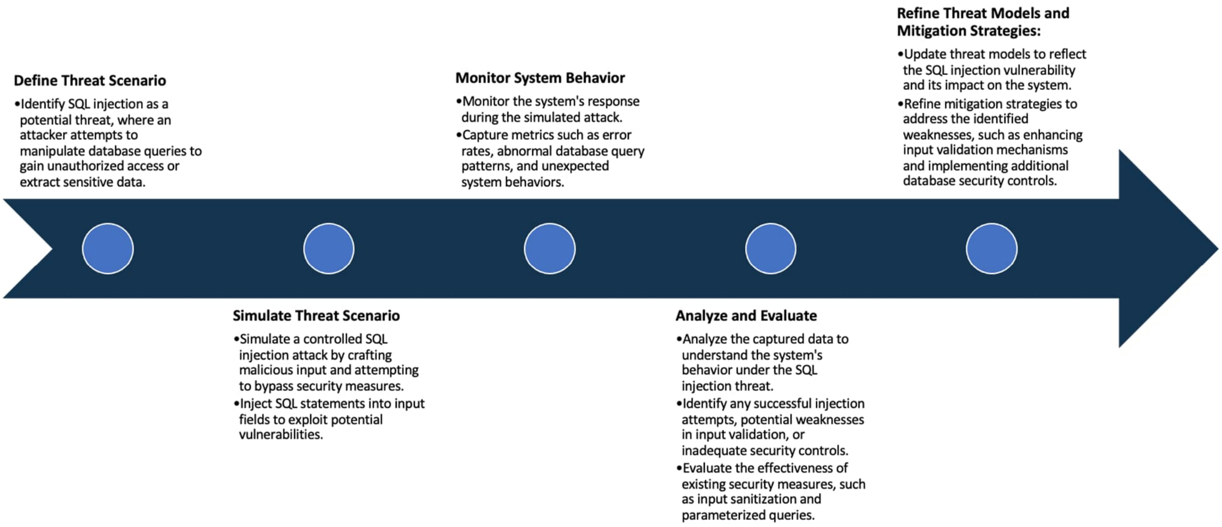
- Introduce controlled failure scenarios that mimic potential attack vectors or vulnerabilities identified in Threat Modeling.
- Simulate failure conditions, such as network disruptions, component failures, or data breaches, to observe the system's response.

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.



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)

```
+-----------------------------------+
|          User Devices             |
| (Desktops, Laptops, Mobile)       |
+--------------------|--------------+
                     |
+--------------------|--------------+
|       Cloud Service               |
| (Hosted Monitoring Platform)       |
+--------------------|--------------+
                     |
+--------------------|--------------+
|       On-Premise Servers          |
|      (Data Center or Server Room) |
+-----------------------------------+
                     |
+--------------------|--------------+
|        Monitored Hosts            |
| (Servers, Containers, Devices)    |
+-----------------------------------+
```

```
+-----------------------------------+
|          User Interface           |
|        (Web Dashboard, API)        |
+--------------------|--------------+
                     |
+--------------------|--------------+
|        Application Logic          |
|   (Data Processing, Analytics)    |
+--------------------|--------------+
                     |
+--------------------|--------------+
|          Data Store              |
| (Time-Series Database, Logs)      |
+-----------------------------------+
                     |
+--------------------|--------------+
|      Agent/Collector              |
|  (Collects Metrics and Logs)       |
+-----------------------------------+
                     |
+--------------------|--------------+
|        External APIs              |
| (Integration with other tools)    |
+-----------------------------------+
```

Chaos Engineering – Hypothesis

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:

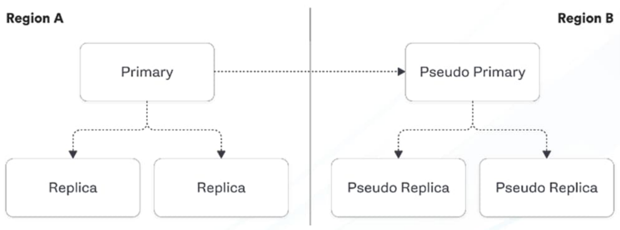
(During an experiment, we need a hypothesis for comparing to a stable control group, and the same applies here too. If there is a reasonable expectation for a particular action according to which we will change the steady state of a system, then the first thing to do is to fix the system so that we accommodate 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.")

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.

Chaos Engineering – Hypothesis

	Known	Unknown
Known	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.
Unknown	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.	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.

Experiment:

(Document your Preparation, Implementation, Observation and Analysis)

Preparation:

- Launched 3 virtual machines on AWS with Ubuntu 20.04
- Set up a Kubernetes cluster with master and two nodes (n1 & n2)

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@kube-n1:~#
```

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@kube-n2:~#
```

```
NAME          STATUS   ROLES                  AGE     VERSION
kube-master   Ready    control-plane,master   8m28s   v1.21.1
kube-n1       Ready    <none>                 6m44s   v1.21.1
kube-n2       Ready    <none>                 6m32s   v1.21.1
root@kube-master:~#
```

Implementation:

- Now creating a deployment using the following script which has replicas = 3

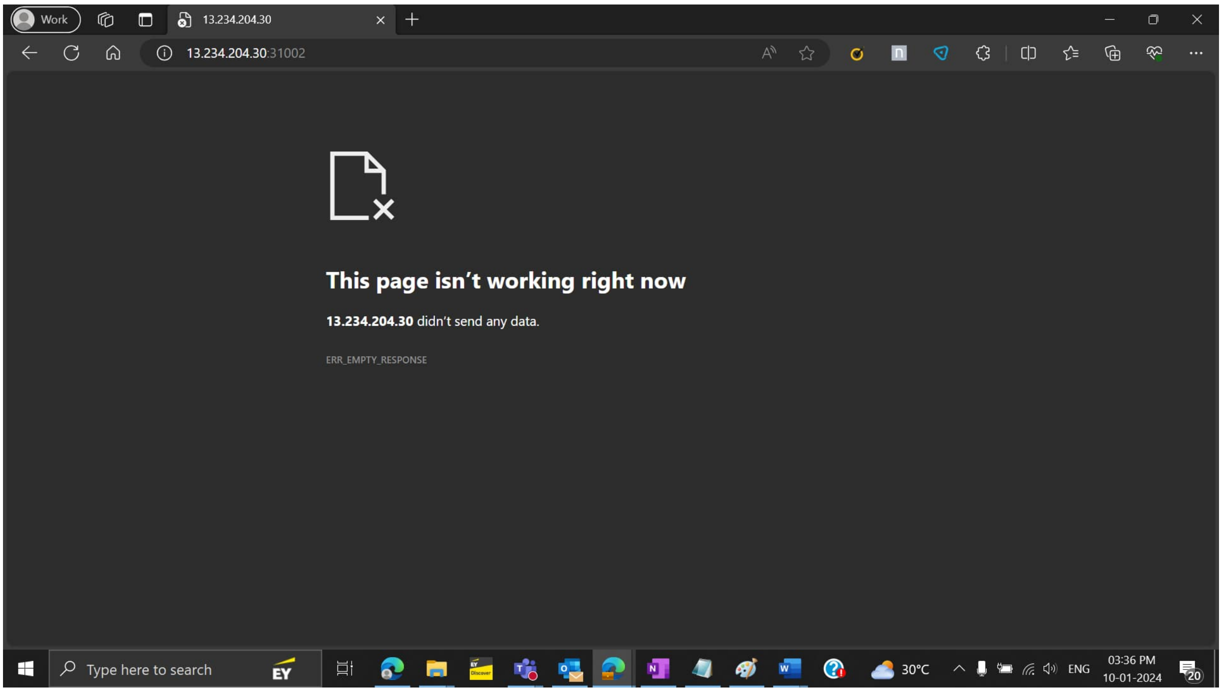
```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: vuljavaapp
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: vuljavaapp
    spec:
      containers:
      - name: vuljavaapp
        image: dhwanit28/vuljapp

  selector:
    matchLabels:
      app: vuljavaapp
```

- Now Creating a service using the following script

```
kind: Service
apiVersion: v1
metadata:
  name: vuljavaapp
spec:
  type: NodePort
  selector:
    app: vuljavaapp
  ports:
  - name: http
    port: 9000
    targetPort: 9000
```

```
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
juice-shop    3/3     3            3           5h28m
vuljavaapp    3/3     3            3           52m
root@kube-master:~# kubectl get svc
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
juice-shop    NodePort    10.106.98.218   <none>        8000:31088/TCP   5h24m
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP          5h44m
vuljavaapp    NodePort    10.101.95.224   <none>        9000:31002/TCP   52m
```

- Now we can see the website gets hosted on localhost:31002 but not able to get any response

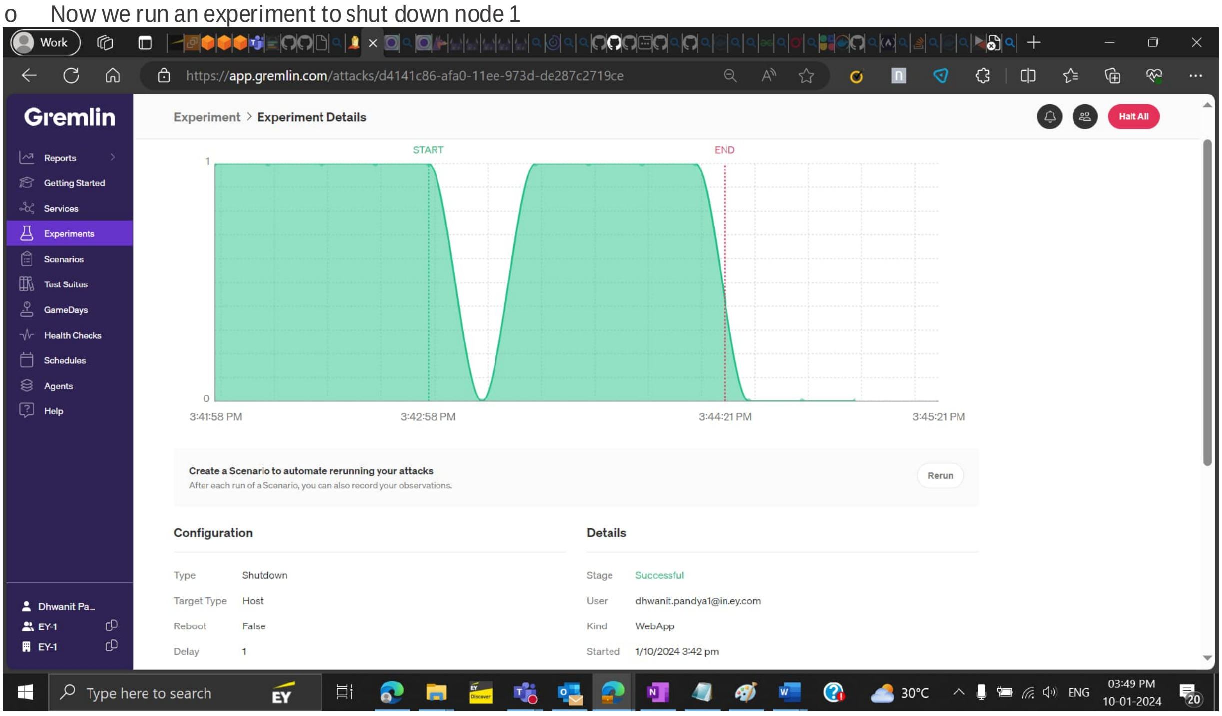


Observation and Analysis

- Performing chaos engineering on this cluster
  - **Initially 2 pods of juice-shop runs on node 1 and 1 pod runs on node 2**

```
vuljavaapp-766dfbbf76-6mps4   1/1   Running   0    54m   192.168.133.73   kube-n2   <none>        <none>
vuljavaapp-766dfbbf76-dbk96   1/1   Running   0    54m   192.168.34.12    kube-n1   <none>        <none>
vuljavaapp-766dfbbf76-pnrl8   1/1   Running   0    54m   192.168.34.11    kube-n1   <none>        <none>
```

  - Now we run an experiment to shut down node 1



  - We observe the following: even if node 1 shuts down then replica takes care of that and the pods that used to run on node 1 started running on node 2

```
NAME                          READY   STATUS        RESTARTS   AGE     IP               NODE      NOMINATED NODE   READINESS GATES
juice-shop-699c69578f-8s2m8   1/1     Running       0          5h5m    192.168.133.68   kube-n2   <none>           <none>
juice-shop-699c69578f-d7d8d   1/1     Running       0          5h5m    192.168.133.69   kube-n2   <none>           <none>
juice-shop-699c69578f-v6lcd   1/1     Running       0          5h44m   192.168.133.65   kube-n2   <none>           <none>
vuljavaapp-766dfbbf76-6mps4   1/1     Running       0          68m     192.168.133.73   kube-n2   <none>           <none>
vuljavaapp-766dfbbf76-dbk96   1/1     Terminating   0          68m     192.168.34.12    kube-n1   <none>           <none>
vuljavaapp-766dfbbf76-fq6kn   1/1     Running       0          63s     192.168.133.75   kube-n2   <none>           <none>
vuljavaapp-766dfbbf76-pnrl8   1/1     Terminating   0          68m     192.168.34.11    kube-n1   <none>           <none>
vuljavaapp-766dfbbf76-qxcr8   1/1     Running       0          63s     192.168.133.74   kube-n2   <none>           <none>
root@kube-master:~#
```

Chaos Engineering – Hypothesis

- Performed vulnerability analysis of the following repo using the snyk tool and found:

Project	Imported	Tested	Issues
pom.xml	2 minutes ago	2 minutes ago	7 C 94 H 38 M 10 L
Code analysis	2 minutes ago	2 minutes ago	0 C 0 H 0 M 0 L
Dockerfile	2 minutes ago	2 minutes ago	0 C 0 H 0 M 0 L

- Out of these Vulnerabilities some critical ones are and their fixes are mentioned below:

1) Vul : ch.qos.logback:logback-classic- **Insecure deserialization**

   Impact: may allow an attacker to manipulate serialized objects and pass harmful data into the application code. Insecure deserialization can lead to denial of service, arbitrary code execution, or privilege escalation

   Fix:
   - Introduce digital signatures and other integrity checks to stop malicious object creation or other data interfering.
   - Run deserialization code in low privilege environments.
   - Keep a log with deserialization exceptions and failures.
   - Use language-agnostic methods for deserialization such as JSON, XML, or YAML.
   - Use safer API which avoids the use of the interpreter.

2) **Vul : org.apache.tomcat.embed:tomcat-embed-core-Information Exposure**

   Impact: If the send file processing completed quickly, it was possible for the Processor to be added to the processor cache twice. This could result in the same Processor being used for multiple requests which in turn could lead to unexpected errors and/or response mix-up.

   Fix: It enables a potential attacker to understand the state of the login function, and could allow an attacker to discover a valid username by trying different values until the incorrect password message is returned. In essence, this makes it easier for an attacker to obtain half of the necessary authentication credentials.

3) **Vul : org.springframework:spring-beans-Remote Code Execution**

   Impact: This vulnerability allows an attacker to perform remote code execution on an application server running a vulnerable configuration, giving them full access to the compromised server.

   Fix: here are some fixes:
   1. Upgrade Spring Framework to a version equal to or greater than 5.2.20 or 5.3.18.
   2. If you are using Spring Boot directly, upgrade to a version equal to or greater than 2.6.6.

4) **Vul : org.apache.tomcat.embed:tomcat-embed-core-Remote Code Execution**

   Impact: It allows an attacker to inject malicious code into an application through a user input field, which is then executed on the fly. Can result in a total loss of integrity, availability, and confidentiality within the application. An attacker may also abuse a code injection vulnerability to execute terminal commands on that server and pivot to adjacent systems.

   Fix: here are some fixes:
   - Avoid the use of dangerous functions

- Reconsider the need for dynamic code execution
- Lock down the interpreter
- Utilize a static analysis tool

5) Vul : socket.io-parser-Denial of Service (DoS)

Impact: ReDoS attack attempts to slow down or even render an application unavailable. processing of the malicious string exhausts the computing power or memory available, thus impacting the application's performance and, in certain circumstances, causing a denial of service (or DoS).

Fix: Avoid using regex for user input validation. Closely review and analyze all patterns before implementation to ensure they do not contain any evil regex patterns.