

Assignment 2 Report

Executive Summary

Overview of Findings

This assignment focused on evaluating the security of a containerized network environment consisting of a web application, a backend database, and several undocumented services. The objective was to identify attack surfaces through active reconnaissance, exploit internal communication vulnerabilities, and subsequently engineer defensive mechanisms to harden the network.

The investigation revealed a network with significant internal trust issues. While the public-facing web application appeared functional, the underlying infrastructure lacked fundamental security controls. Using a custom-built multi-threaded port scanner, I successfully mapped the Docker subnet (172.20.0.0/16), identifying five active services: a Flask web app (Port 5000), a MySQL database (Port 3306), a Redis instance (Port 6379), and two hidden services—a "Secret SSH" service (Port 2222) and a "Secret API" (Port 8888).

Critical Vulnerabilities Discovered

The most critical vulnerability identified was the lack of encryption in the communication channel between the web application and the database.

- **Cleartext Transmission:** Configuration analysis revealed that the MySQL server was explicitly configured with `skip_ssl`, and the Python application client forced `ssl_disabled=True`.
- **MITM Exploitation:** This misconfiguration allowed for a passive Man-in-the-Middle (MITM) attack. By capturing traffic on the Docker bridge interface using `tcpdump`, I was able to intercept SQL queries and responses in plaintext.
- **Data Exfiltration:** This exploit resulted in the compromise of a sensitive API token (`FLAG{n3tw0rk_tr4ff1c_1s_n0t_s3cur3}`), which provided unauthorized access to the restricted "Secret API" service.

Recommended Fixes & Implemented Defenses

To remediate these issues and secure the environment, several defensive measures were proposed and implemented:

1. **Encryption:** It is imperative to enable TLS/SSL for all database connections to render traffic sniffing useless.

2. **Port Knocking:** To protect the SSH service, I implemented a Port Knocking system. Access to the SSH port is now blocked by default and only opens dynamically when a client triggers a specific sequence of UDP packets (1234 → 5678 → 9012).
3. **Honeypot Deployment:** A low-interaction honeypot was deployed to mimic a vulnerable service. This system successfully logs scanning attempts and connection metadata, providing early warning of network intruders.

Demo Video (All Parts): <https://www.youtube.com/watch?v=FhllEWnLShk>

Part 1: Reconnaissance

Port Scanner Design and Implementation

To effectively map the network attack surface, I created a custom multi-threaded port scanner developed in Python. The tool was designed with a focus on speed, accuracy, and service identification.

- **Core Architecture:** The scanner uses the standard socket library to perform **TCP Connect Scans**. It attempts a full 3-way handshake (SYN, SYN-ACK, ACK) with target ports; a successful connection (return code 0) indicates an "Open" state.
- **Concurrency:** To mitigate the slowness of sequential scanning, the tool implements the `concurrent.futures.ThreadPoolExecutor`. This allows for **100+ concurrent threads**, enabling the scanning of the full 65,535 port range on a target in seconds rather than minutes.
- **Service Fingerprinting (Banner Grabbing):** Upon establishing a connection, the scanner does not immediately disconnect. Instead, it attempts to `recv(1024)` bytes of data. This allows it to capture "welcome messages" (banners) from services like SSH, which is critical for identifying the specific software and version running on a discovered port.
- **User Interface:** The tool utilizes `argparse` for flexible command-line input (targets, port ranges, thread counts) and integrates `tqdm` to provide a real-time visual progress bar, enhancing the user experience during long scans.

| Target IP | Port | Service Type | Banner / Identification | Purpose & Findings |
|-------------|------|-----------------|-------------------------|---|
| 172.20.0.10 | 5000 | Web Application | (Silent/HTTP) | The main Flask web interface. Generating traffic here triggers DB queries. |
| 172.20.0.11 | 3306 | MySQL Database | (MySQL Protocol) | Backend storage. Critical Vulnerability: SSL is disabled, allowing plaintext interception. |
| 172.20.0.20 | 2222 | Secret SSH | SSH-2.0-Open SSH | Hidden Service. A non-standard SSH port. Contains Flag 2 (access requires credentials). ssh -p 2222 sshuser@172.20.0.20 FLAG{h1dd3n_s3rv1c3s_n33d_pr0t3ct10n} |
| 172.20.0.21 | 8888 | Secret API | (Silent/JSON) | Hidden Service. An undocumented API. Requires an Auth Token (Flag 1) to access Flag 3 . |
| 172.20.0.22 | 6379 | Redis | (Redis Protocol) | A data structure store. Identified as a distractor service with no flags. |

Tool: Custom Python Port Scanner.

Methodology: The Docker subnet 172.20.0.0/16 was identified via docker network ls. The scanner was executed against specific container IPs to identify open ports. Banner grabbing was used to differentiate between the Secret SSH server (port 2222) and other generic services. This phase located the entry points for Flags 2 and 3.

Part 2: MITM Attack

Vulnerability Analysis

The primary vulnerability enabling the Man-in-the-Middle (MITM) attack was the lack of encryption in the communication between the Web Application (2_network_webapp) and the MySQL Database (2_network_database).

Configuration files revealed how SSL/TLS was disabled on both ends:

- **Database Configuration:** The `mysql.cnf` file contained the directive `skip_ssl` and `ssl=0`, effectively forcing the database to communicate in plaintext.
- **Application Code:** The `app.py` file utilized `pymysql` with the arguments `ssl=None` and `ssl_disabled=True`.

This misconfiguration meant that all SQL queries, including those retrieving sensitive user data and secrets, were transmitted across the Docker bridge network in cleartext, readable by any compromised container or attacker on the same network segment.

Attack Methodology

To exploit this, I first identified the Docker bridge network interface (`br-0555bd9df806`) using `docker network ls`. I then used `tcpdump` to capture traffic specifically on port 3306 (MySQL default port).

- **Command Used:** `sudo tcpdump -i br-0555bd9df806 -A -s 0 'port 3306'`.
- **Execution:** While the packet sniffer was running, I triggered traffic by accessing the `/api/secrets` endpoint on the web application.

Captured Data Analysis

The `tcpdump` output provided a raw hex and ASCII dump of the network packets. Because encryption was disabled, I was able to read the SQL response packets directly.

- **Findings:** The capture revealed the contents of the `secrets` table. Specifically, I intercepted the `api_token` with the value `FLAG{n3tw0rk_tr4ff1c_1s_n0t_s3cur3}`.
- **Flag Capture:** Using this intercepted token, I authenticated against the "Secret API" service discovered in Part 1 on port 8888. A request to `http://172.20.0.21:8888/flag?token=...` returned Flag 3: `FLAG{p0rt_kn0ck1ng_4nd_h0n3yp0ts_s4v3_th3_d4y}`.

Real-world Impact Assessment

In a real-world scenario, this vulnerability would be catastrophic. It allows an attacker with network presence (e.g., via a compromised microservice) to passively collect credentials, PII (Personally Identifiable Information), and API keys without alerting the victim. The exposed API token could be used to pivot to other internal systems, as demonstrated by the transition from the database attack to the hidden API service.

Part 3: Security Fixes

Fix 1: Port Knocking

Design Decisions

To protect the hidden SSH service on port 2222, I implemented a Port Knocking system. The core design is based on a **State Machine** model rather than a simple packet sniffer. This ensures that the specific sequence of ports must be hit in the exact order within a limited time window.

- **Knock Sequence:** 1234 → 5678 → 9012.
- **Mechanism:** The server listens on these UDP ports and tracks the state of each source IP.
- **Firewall Integration:** I used `iptables` to dynamically insert and delete rules in the INPUT chain.

Implementation Details

The implementation consists of a client and a server:

- **Server (`knock_server.py`):** Uses Python's `select` module for non-blocking I/O to listen on multiple UDP sockets simultaneously. It maintains a `client_state` dictionary to track which stage of the sequence an IP address is currently in. When the sequence is completed, it executes `iptables -I INPUT -s <IP> -p tcp --dport 2222 -j ACCEPT`.
- **Client (`knock_client.py`):** A script that sends UDP packets to the target ports with a configurable delay.
- **Automatic Relocking:** A threading timer is used to automatically remove the allow rule after 30 seconds, ensuring the port does not remain open indefinitely.

Security Analysis

The system provides "Security through Obscurity" but also adds a tangible layer of defense.

- **State Reset:** If a user knocks on the wrong port, their state is reset to 0, preventing brute-force attempts.
- **Timeout Window:** The server enforces a 10-second window for the sequence. If the knocks are too far apart, the state resets.
- **Limitations:** The sequence is currently hardcoded and susceptible to replay attacks if an attacker is sniffing the network.

Fix 2: Honeypot

Architecture and Design

The honeypot was designed to mimic a vulnerable SSH service to attract and log unauthorized access attempts.

- **Service Simulation:** The honeypot listens on port 22 (exposed as 2222) and presents a realistic banner to trick scanners.
- **Logging System:** A custom logger records connection metadata, including timestamp, source IP, and source port, to a JSON Lines file (`connections.jsonl`).

Logging Mechanisms

The logging implementation captures events in a structured format for easy parsing.

- **Log Location:** `/app/logs/connections.jsonl`
- **Data Fields:** `timestamp`, `event` (e.g., "connection"), `client_ip`, and `client_port`.

Analysis of Captured Attacks

During the testing phase, the honeypot successfully logged multiple interaction attempts.

- **Observed Activity:** The logs show repeated connections from `172.20.0.1` (the gateway/host), indicating scanning activity.
- **Pattern:** Connections were made to high ephemeral ports (e.g., 40642, 39300), which is consistent with a port scanner or a client attempting to connect.

Part 4: Remediation Recommendations

1. Fixing the MITM Vulnerability (TLS/SSL):

- **Database:** Modify `mysql.cnf` to remove `skip_ssl` and configure `ssl-ca`, `ssl-cert`, and `ssl-key` pointing to valid certificates.
- **Application:** Update `app.py` to remove `ssl_disabled=True` and provide the path to the CA certificate in the `ssl` dictionary passed to `pymysql.connect`.

2. Service Discovery Protection:

- Implement a "default deny" firewall policy. Only explicitly allowed traffic should pass.
- Disable the "Secret API" and "Secret SSH" services if they are not business-critical, or move them to a completely separate internal network not accessible by the Web App container.

3. Network Segmentation:

- Isolate the Database: The database should not be on the same bridge network as the public-facing Web App if possible. Use Docker overlay networks to strictly control communication paths.
- The Web App should only be allowed to communicate with the Database on port 3306, blocking all other internal traffic.

4. Monitoring and Detection:

- Deploy a permanent Intrusion Detection System (IDS) like Snort or Suricata to monitor for cleartext SQL patterns (e.g., `SELECT * FROM secrets`).
- Centralize logs from the Honeypot and Port Knocking server to a SIEM (Security Information and Event Management) system to alert on scanning activity in real-time.

Conclusion

Lessons Learned

This assignment demonstrated the critical importance of defense-in-depth. I learned that "hidden" services are easily found with basic reconnaissance tools like port scanners. The MITM attack highlighted that internal networks are not safe zones; encryption must be enforced everywhere, not just at the network edge. Building the port knocker and honeypot provided practical experience with low-level socket programming and firewall manipulation using iptables.

Skills Acquired

- **Network Reconnaissance:** Proficiently using nmap logic and Python sockets to map attack surfaces.
- **Packet Analysis:** Using tcpdump to intercept and analyze cleartext traffic.
- **Defensive Coding:** Implementing state machines for port knocking and logging mechanisms for honeypots.
- **Container Security:** Understanding Docker networking namespaces and bridge vulnerabilities.

Future Work

Future improvements could include implementing a rolling code sequence (TOTP) for the port knocker to prevent replay attacks and expanding the honeypot to simulate a full shell environment (using a library like cowrie) to capture attacker commands rather than just connection attempts.