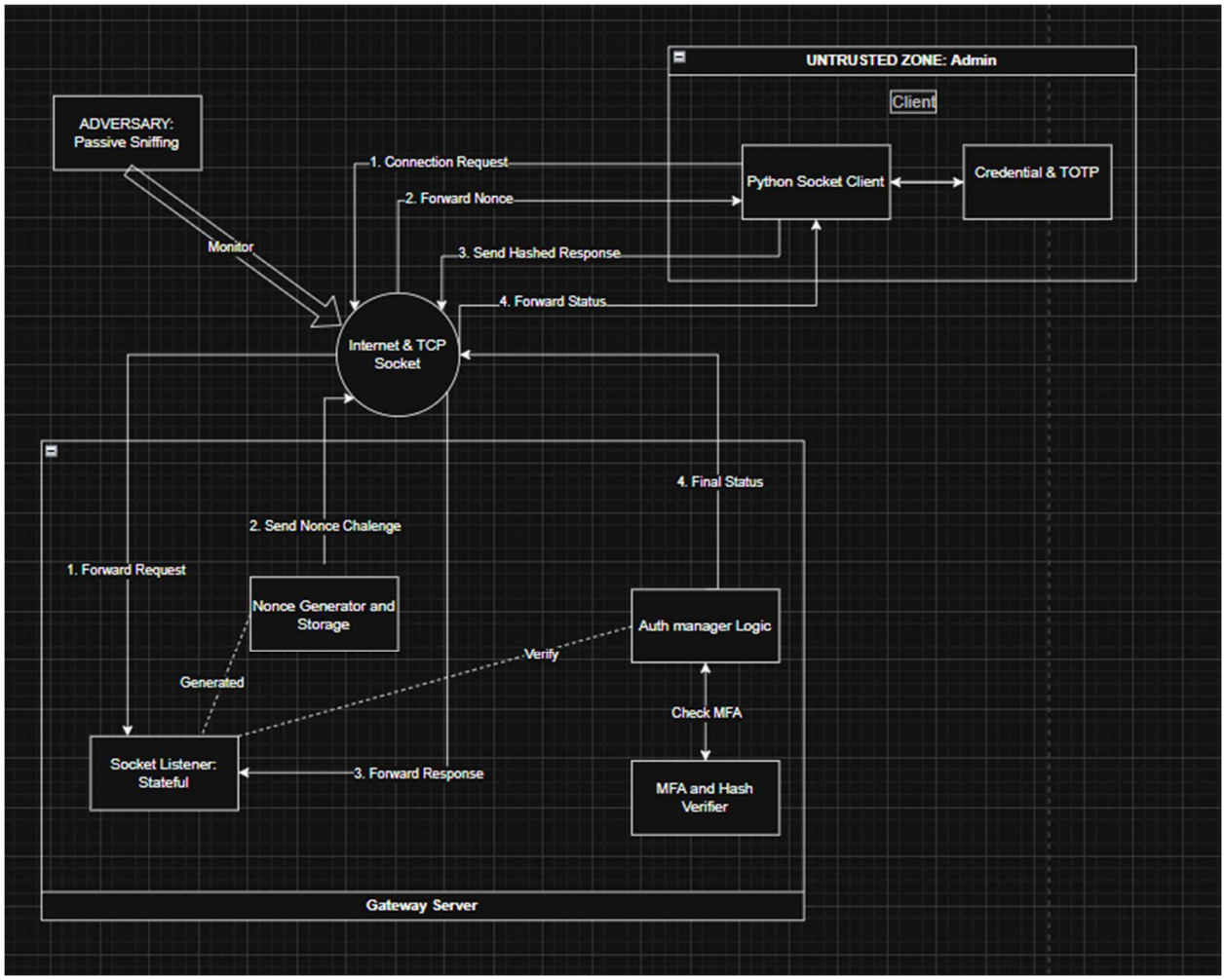
FSCT 8561: Mid-Term Exam

By: Thomas Duong – A01317687

DAY 1: System Design & Threat Modeling

Part A: Architecture Design



Client & Server Components

A custom Python script is used to create communication between both ends of the system. Socket library is used to work with the raw TCP connections. The Admin Client initiates the connection by opening a socket and making a call to the public IP of the Gateway to connect to it. On the other end, the Gateway Listener operates under a while loop of continuous socket.listen() and monitors incoming requests..

The Gateway Logic

My system does not connect to the diagnostic tools of the backend directly by an intermediary gateway. The SRDS gateway acts as a security barrier just as a WAP gateway. It terminates the connection between the external socket and validates the requester and then executes any internal health checks. Therefore, although an attacker may be spying on the channel as indicated in the diagram, they will not be able to gain access to sensitive modules inside the firm.

Justification: Stateful Communication

The model of communication to the SRDS architecture that I selected is stateful to ensure internal coherence and security. The model is necessary since the SRDC mechanism involves the Gateway to store and trace the nonce which is unique and randomized in step 2, to subsequently verify the hashed response by the client in step 3. Using a stateful model, the Gateway is able to maintain an authenticated session so that the administrator need not re-authenticate his/her MFA credentials with every diagnostic command.

Part B: Adversarial Threat Model

Asset	Threat	Attack Vector	Impact	Mitigation
Admin Credentials	**Replay Attack** (Sockets)	**Sniffing Socket traffic:** Adversary captures valid login packets on the monitored channel.	**Unauthorized Access:** Attacker gains full administrative control.	**Challenge-Response Nonce:** Gateway issues a one-time random challenge for every login attempt (Step 2 in diagram).
Server Availability	**SYN Flood / DoS** (Scapy)	**TCP Sockets:** Attacker uses custom Scapy scripts to overwhelm the Socket Listener with	**Service Disruption:** Legitimate administrators cannot connect to perform	**Scapy Detection:** Use Scapy to monitor for abnormal SYN rates and automatically

		half-open connections.	diagnostic scans.	drop offending IP addresses.
Diagnostic Data	**Info Leakage** (Scanning)	**Nmap Recon:** Adversary sniffs the socket during a health check to see raw network scan results.	**Privacy Breach:** Attacker learns sensitive internal network architecture details.	**Payload Encryption:** Encrypt all diagnostic results within the custom protocol before transmission over the socket.
MFA Token	**Brute-Force Attack** (MFA)	**MFA Logic:** Attacker attempts to script multiple login attempts using guessed pyotp codes.	**Account Compromise:** Bypassing the possession factor of the MFA module.	**Account Lockout:** Implement a temporary lockout or IP ban after 3 failed pyotp verification attempts.
Internal Logic	**Command Injection** (Web)	**Custom Protocol:** Attacker injects malicious shell commands into the "target IP" field of a scan request.	**Remote Code Execution:** Attacker gains control of the Gateway server itself.	**Strict Input Validation:** Use a whitelist of allowed characters and strict regex to validate target IP addresses before passing them to Nmap.

DAY 2: Protocol & Component Specification

Part C: Secure Protocol Design

1. Message Format: SRDS-JSON

The protocol uses a standard JSON structure for all exchanges to ensure internal coherence between the Client and Gateway.

Standard Message Schema:

- **version**: Protocol version (e.g., "1.0") to prevent compatibility issues.
- **step**: The current phase of the handshake (AUTH_REQ, CHALLENGE, AUTH_RESP, STATUS).
- **payload**: A nested object containing the data specific to that step.
- **id**: A unique session ID or tracking number.

2. The Authentication Handshake (Challenge-Response)

This protocol explicitly incorporates a Challenge-Response logic to ensure that even a monitored channel cannot be exploited.

Step 1: Initiation (AUTH_REQ)

The handshake begins when the Admin Client sends an AUTH_REQ message to the Gateway. The JSON packet is used to identify the user attempting to log in, and the Gateway checks the account against the database and establishes the state specific to the session. With this initial step being a simplistic one we minimize how much data is transferred before even a secure challenge is issued hence making sure that the server does not handle complex data before it has been verified by the server.

Step 2: The Challenge (CHALLENGE)

After the identification of the user by the Gateway, a cryptographically secure random nonce is created. This nonce is returned to the client in a CHALLENGE message and is stored in the stateful memory of the Gateway with a very tight 60 second expiration time. This Challenge element of the logic makes the client demonstrate that they have the right credentials by applying a value that is only valid to this one and only connection attempt.

Step 3: The Response (AUTH_RESP)

The nonce is sent to the Client who calculates a SHA-256 hash in combination of the password of the user and the received nonce. This Proof is in turn retransmitted in the AUTH_RESP packet together with the existing 6-digit TOTP code produced by pyotp. We make sure that the real password is never sent across the channel under observation by hashing the password with nonce and therefore at this stage the resulting string is unique to this session only.

Step 4: Establishment (STATUS)

In the last process, the MFA and Hash Verifier of the Gateway compare the proof of the client with its internal calculation. When the hashes are identical and the TOTP code is correct, the Gateway enters a nonce as consumed to prevent any reuse in the future and responds with a STATUS message with a 200 response. This officially goes to create the session, switching the Gateway out of the authentication phase, into the diagnostic service phase in which the Gateway is now accepting health-check commands.

3. Adversarial Defense: Replay Prevention

My protocol prevents an attacker from re-injecting a captured valid packet through three primary mechanisms:

1. **Nonce Volatility**: The nonce in Step 2 is unique per session. When the Gateway obtains the response of Step 3, it removes such nonce out of its storage. In case an opponent attempts to re-send the Step 3 packet, the operation will be unsuccessful as the nonce is not valid or active anymore.

2. **Cryptographic Salt**: When the password is hashed together with the nonce it forms a one-time credential. A nonce obtained during a captured proof cannot be repurposed in a subsequent login since a new nonce will be generated.

3. **Strict State Sequencing**: The state machine of the Gateway discards the packet that comes out of order. In case the attacker tries to re-inject an AUTH_RESP packet when the Gateway is idle, the packet has been dropped.

Part D: Component-Level Design

1. MFA Module (hashlib + PYOTP)

- **Inputs**: User-provided password, 6-digit TOTP code, and the session-specific Nonce.

- **Outputs**: Boolean (True/False) authentication status.

- **Security Controls**:
    - Uses hashlib to combine the password with the Nonce, ensuring the raw password is never processed or stored in memory in plain text.
    - Uses pyotp to verify the "possession factor" with a 30-second window to prevent old codes from being used.

2. Recon Module (python-nmap)

- **Inputs**: Validated IP address or hostname and specific scan flags.
- **Outputs**: JSON-formatted string containing open ports and service versions.
- **Security Controls**:
    - The module is configured to run nmap with the lowest possible system privileges to prevent an attacker from gaining root access if the module is compromised.

3. Detection Module (Scapy)

- **Inputs**: Raw network packets captured from the Socket Listener interface.
- **Outputs**: Alert logs and automatic IP "shunning" (blocking) commands.
- **Security Controls**:
    - Uses Scapy to monitor the ratio of SYN packets to ACKs. If a threshold is exceeded, it identifies a SYN Flood attack and drops further traffic from the source IP to protect the Gateway.

4. Negative Space Decision

Data NOT Collected: Persistent Administrative Command Logs

- I have intentionally chosen not to store the specific command history or diagnostic results on the Gateway's local hard drive once the session is closed.
- This reduces the attack surface by ensuring that if the Gateway server is ever compromised, there is no "treasure trove" of previous network scans or administrative activity for an attacker to exfiltrate. It follows the principle of Data Minimization, keeping the "Impact" of a breach as low as possible

DAY 3: Constrained Implementation & Analysis

Part E: Minimal Implementation

GitHub link: https://github.com/DThomas230/FSCT-8561_Security-Applications/tree/master/midterm

srds__gateway.py

```
#!/usr/bin/env python3

import socket
import hashlib
import pyotp
import json
import os
import time

HOST = "127.0.0.1"
PORT = 9999
NONCE_LIFETIME = 60          # Nonce expires after 60 seconds (Part C, Step 2)
MAX_FAILED_ATTEMPTS = 3      # Lockout threshold (Part D/ Part B threat table)
LOCKOUT_DURATION = 300       # 5-minute lockout window

# ──────────────────────────────────────────────
# Passwords are stored as SHA-256 hashes - never in plaintext.
# ──────────────────────────────────────────────
TOTP_SECRET = pyotp.random_base32()   # Generated once; share with the client

# ──────────────────────────────────────────────
# LOGIN CREDENTIALS (for testing)
# User: admin_01
# Password: SecurePass123
# ──────────────────────────────────────────────
users_db = {
    "admin_01": {
        # hash of "SecurePass123" - raw password is never stored (Part D §1)
        "password_hash": hashlib.sha256("SecurePass123".encode()).hexdigest(),
        "totp_secret": TOTP_SECRET,
    }
}

# ──────────────────────────────────────────────
# RUNTIME STATE (in-memory only - Part D, Data Minimization)
```

```python
# ──────────────────────────────────────────────────
nonce_store = {}        # {session_id: {"nonce": str, "created": float, "user": str}}
failed_attempts = {}    # {username: {"count": int, "last_time": float}}
session_state = {}      # {session_id: current_step}  - enforces strict sequencing


# ═══════════════════════════════════════════════
# NONCE GENERATOR & STORAGE
# ═══════════════════════════════════════════════
def generate_nonce():
    """
    Create a cryptographically secure random nonce.
    Security Purpose: Each login attempt gets a unique nonce so that
    an attacker who captures Step 3 cannot replay it later (Part C).
    """
    return os.urandom(16).hex()   # 32-char hex string


def store_nonce(session_id, nonce, username):
    """Store the nonce with a timestamp so it can expire after 60 seconds."""
    nonce_store[session_id] = {
        "nonce": nonce,
        "created": time.time(),
        "user": username,
    }


def validate_and_consume_nonce(session_id, nonce):
    """
    Check that the nonce exists and has not expired, then DELETE it.
    Security Purpose - Nonce Volatility (Part C):
    Once consumed the nonce can never be reused, defeating replay attacks.
    """
    entry = nonce_store.get(session_id)
    if entry is None:
        return False, "Nonce not found (possible replay)"

    # Check expiration
    if time.time() - entry["created"] > NONCE_LIFETIME:
        del nonce_store[session_id]
        return False, "Nonce expired"

    # Check value matches
    if entry["nonce"] != nonce:
```

```
        del nonce_store[session_id]
        return False, "Nonce mismatch"

    # Consume - delete immediately so it cannot be reused
    del nonce_store[session_id]
    return True, "OK"


# ═══════════════════════════════════════════════
# MFA & HASH VERIFIER
# ═══════════════════════════════════════════════
def verify_proof(username, proof, nonce):
    """
    Verify the SHA-256 proof sent by the client.
    The client hashes (password + nonce) so the raw password is
    never transmitted over the monitored channel (Part C).
    """
    stored_hash = users_db[username]["password_hash"]
    # Recreate the expected proof: SHA-256( password_hash + nonce )
    expected = hashlib.sha256((stored_hash + nonce).encode()).hexdigest()
    return proof == expected


def verify_totp(username, code):
    """
    Verify the 6-digit TOTP code (possession factor).
    Security Purpose: Even if an attacker knows the password, they
    cannot authenticate without the physical TOTP device (Part D).
    A 30-second window is used; old codes are rejected.
    """
    secret = users_db[username]["totp_secret"]
    totp = pyotp.TOTP(secret)
    return totp.verify(code, valid_window=1)


# ═══════════════════════════════════════════════
# ACCOUNT LOCKOUT  (Part B - Brute-Force Mitigation)
# ═══════════════════════════════════════════════
def is_locked_out(username):
    """Return True if the user has exceeded the failed-attempt threshold."""
    record = failed_attempts.get(username)
    if record is None:
        return False
    # Reset after lockout duration
    if time.time() - record["last_time"] > LOCKOUT_DURATION:
        del failed_attempts[username]
```

```
        return False
    return record["count"] >= MAX_FAILED_ATTEMPTS


def record_failure(username):
    """Increment the failure counter for brute-force tracking."""
    record = failed_attempts.get(username, {"count": 0, "last_time": 0})
    record["count"] += 1
    record["last_time"] = time.time()
    failed_attempts[username] = record


def reset_failures(username):
    """Clear failures on successful login."""
    failed_attempts.pop(username, None)


# ============================================
# SRDS-JSON MESSAGE HELPERS  (Part C)
# ============================================
def build_message(step, payload, session_id=""):
    """Build a protocol-compliant SRDS-JSON message."""
    return json.dumps({
        "version": "1.0",
        "step": step,
        "payload": payload,
        "id": session_id,
    })


# ============================================
# AUTH MANAGER LOGIC  (Core handler from diagram)
# ============================================
def handle_client(client_socket, address):
    """
    Manage the full 4-step handshake for one client connection.
    Strict State Sequencing (Part C): the handler expects
    messages to arrive in order - AUTH_REQ then AUTH_RESP.
    """
    session_id = os.urandom(8).hex()       # Unique ID per session
    session_state[session_id] = "IDLE"     # Start in IDLE state
    print(f"[+] New connection from {address}  |  session={session_id}")

    try:
        # STEP 1: Receive AUTH_REQ
        raw = client_socket.recv(4096).decode("utf-8")
```

```
        msg = json.loads(raw)

        # Strict sequencing - only accept AUTH_REQ when IDLE
        if msg.get("step") != "AUTH_REQ" or session_state[session_id] !=
"IDLE":
            client_socket.send(build_message(
                "STATUS", {"code": 400, "msg": "Bad request sequence"},
session_id
            ).encode())
            return

        username = msg["payload"]["user"]
        print(f"Step 1 ← AUTH_REQ from user '{username}'")

        # Check if user exists
        if username not in users_db:
            client_socket.send(build_message(
                "STATUS", {"code": 401, "msg": "Unknown user"}, session_id
            ).encode())
            return

        # Check lockout before issuing a challenge
        if is_locked_out(username):
            client_socket.send(build_message(
                "STATUS", {"code": 403, "msg": "Account locked. Try again
later."}, session_id
            ).encode())
            print(f"[!] Account '{username}' is locked out")
            return

        # STEP 2: Send CHALLENGE (nonce)
        nonce = generate_nonce()
        store_nonce(session_id, nonce, username)
        session_state[session_id] = "CHALLENGE_SENT"

        challenge_msg = build_message("CHALLENGE", {"nonce": nonce},
session_id)
        client_socket.send(challenge_msg.encode())
        print(f"Step 2 → CHALLENGE sent  (nonce={nonce[:12]}...)")

        # STEP 3: Receive AUTH_RESP
        raw = client_socket.recv(4096).decode("utf-8")
        msg = json.loads(raw)

        # Strict sequencing - only accept AUTH_RESP after CHALLENGE
```

```
        if msg.get("step") != "AUTH_RESP" or session_state[session_id] !=
"CHALLENGE_SENT":
            client_socket.send(build_message(
                "STATUS", {"code": 400, "msg": "Out-of-order packet dropped"},
session_id
            ).encode())
            print(f"[!] Rejected out-of-order packet")
            return

        proof = msg["payload"]["proof"]
        mfa_code = msg["payload"]["mfa"]
        print(f"Step 3 ← AUTH_RESP received")

        # Validate nonce (consume it to prevent replay)
        nonce_ok, nonce_msg = validate_and_consume_nonce(session_id, nonce)
        if not nonce_ok:
            client_socket.send(build_message(
                "STATUS", {"code": 401, "msg": nonce_msg}, session_id
            ).encode())
            print(f"[!] Nonce validation failed: {nonce_msg}")
            return

        # Verify the hashed proof
        if not verify_proof(username, proof, nonce):
            record_failure(username)
            client_socket.send(build_message(
                "STATUS", {"code": 401, "msg": "Invalid credentials"},
session_id
            ).encode())
            print(f"[!] Proof verification failed")
            return

        # Verify the TOTP code (possession factor)
        if not verify_totp(username, mfa_code):
            record_failure(username)
            attempts = failed_attempts.get(username, {}).get("count", 0)
            client_socket.send(build_message(
                "STATUS", {"code": 401, "msg": f"Invalid MFA
({attempts}/{MAX_FAILED_ATTEMPTS})"}, session_id
            ).encode())
            print(f"[!] TOTP verification failed
({attempts}/{MAX_FAILED_ATTEMPTS})")
            return

        # STEP 4: Send STATUS - Authenticated
```

```
        reset_failures(username)
        session_state[session_id] = "AUTHENTICATED"
        client_socket.send(build_message(
            "STATUS", {"code": 200, "msg": "Authenticated"}, session_id
        ).encode())
        print(f"Step 4 → STATUS: Authenticated ✓")

    except (json.JSONDecodeError, KeyError) as e:
        print(f"[!] Malformed message: {e}")
        client_socket.send(build_message(
            "STATUS", {"code": 400, "msg": "Malformed message"}, session_id
        ).encode())
    finally:
        # Clean up session data - Data Minimization (Part D §4)
        session_state.pop(session_id, None)
        nonce_store.pop(session_id, None)
        client_socket.close()


# ═══════════════════════════════════════════
# SOCKET LISTENER - STATEFUL  (Component from diagram)
# ═══════════════════════════════════════════
def start_gateway():
    """
    Start the Gateway's socket listener.
    It runs in a continuous loop accepting one connection at a time
    (stateful), as described in the architecture diagram.
    """
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((HOST, PORT))
    server.listen(5)

    print("=" * 55)
    print("SRDS GATEWAY SERVER - Challenge-Response Auth")
    print("=" * 55)
    print(f"[*] Listening on {HOST}:{PORT}")
    print(f"[*] TOTP Secret (share with client): {TOTP_SECRET}")
    print(f"[*] Nonce lifetime: {NONCE_LIFETIME}s  |  Lockout after
{MAX_FAILED_ATTEMPTS} failures")
    print()

    try:
        while True:
            client_socket, address = server.accept()
```

```
            handle_client(client_socket, address)
    except KeyboardInterrupt:
        print("\n[*] Gateway shutting down.")
    finally:
        server.close()


if __name__ == "__main__":
    start_gateway()
```

srds_client.py

```
#!/usr/bin/env python3

import socket
import hashlib
import pyotp
import json

GATEWAY_HOST = "127.0.0.1"
GATEWAY_PORT = 9999


# ═══════════════════════════════════════════
# CREDENTIAL & TOTP MODULE
# ═══════════════════════════════════════════
def compute_proof(password, nonce):
    """
    Build the one-time proof that the Gateway expects.
    Steps:
      1. Hash the raw password with SHA-256 (so plaintext is never kept).
      2. Hash (password_hash + nonce) to bind the credential to this session.
    Security Purpose (Part C - Cryptographic Salt):
      The nonce acts as a salt, making the proof unique per session.
      Even if an adversary captures this proof, it cannot be replayed
      because the next session will use a different nonce.
    """
    password_hash = hashlib.sha256(password.encode()).hexdigest()
    proof = hashlib.sha256((password_hash + nonce).encode()).hexdigest()
    return proof


def generate_totp(secret):
```

```
    """
    Generate a 6-digit TOTP code from the shared secret.
    Security Purpose (Part D - Possession Factor):
      This proves the admin physically possesses the TOTP device/secret.
      The code changes every 30 seconds, limiting the window for misuse.
    """
    totp = pyotp.TOTP(secret)
    return totp.now()


# ═══════════════════════════════════════════
# PYTHON SOCKET CLIENT
# ═══════════════════════════════════════════
def authenticate(username, password, totp_secret):
    """
    Execute the full 4-step handshake with the SRDS Gateway.
    Follows the SRDS-JSON message format defined in Part C.
    """
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        # Connect to the Gateway's public socket
        client.connect((GATEWAY_HOST, GATEWAY_PORT))
        print(f"[*] Connected to Gateway at {GATEWAY_HOST}:{GATEWAY_PORT}\n")

        # STEP 1: Send AUTH_REQ
        # Tell the Gateway which user wants to authenticate.
        auth_req = json.dumps({
            "version": "1.0",
            "step": "AUTH_REQ",
            "payload": {"user": username},
        })
        client.send(auth_req.encode())
        print(f"Step 1 → AUTH_REQ sent (user='{username}')")

        # STEP 2: Receive CHALLENGE
        # The Gateway sends back a random nonce (Part C §2, Step 2).
        raw = client.recv(4096).decode("utf-8")
        msg = json.loads(raw)

        # If the Gateway responded with STATUS, something went wrong
        if msg["step"] == "STATUS":
            print(f"[!] Gateway rejected: {msg['payload']['msg']}")
            return False
```

```
        nonce = msg["payload"]["nonce"]
        session_id = msg["id"]
        print(f"Step 2 ← CHALLENGE received (nonce={nonce[:12]}...)")

        # STEP 3: Send AUTH_RESP
        # Compute the hash proof and generate the TOTP code.
        proof = compute_proof(password, nonce)
        mfa_code = generate_totp(totp_secret)

        auth_resp = json.dumps({
            "version": "1.0",
            "step": "AUTH_RESP",
            "payload": {
                "proof": proof,    # SHA-256(SHA-256(password) + nonce)
                "mfa": mfa_code,   # 6-digit TOTP code
            },
            "id": session_id,
        })
        client.send(auth_resp.encode())
        print(f"Step 3 → AUTH_RESP sent (proof + MFA code)")

        # STEP 4: Receive STATUS
        raw = client.recv(4096).decode("utf-8")
        msg = json.loads(raw)
        code = msg["payload"]["code"]
        message = msg["payload"]["msg"]

        if code == 200:
            print(f"Step 4 ← STATUS: {message}  ✓")
            return True
        else:
            print(f"Step 4 ← STATUS: {message}  ✗  (code {code})")
            return False

    except ConnectionRefusedError:
        print("[!] Cannot connect - is the Gateway running?")
        return False
    except Exception as e:
        print(f"[!] Error: {e}")
        return False
    finally:
        client.close()


# ═══════════════════════════════════════════
```

```
# MAIN - Interactive Client Menu
# ═══════════════════════════════════════════
def main():
    print("=" * 50)
    print("SRDS ADMIN CLIENT - Challenge-Response Auth")
    print("=" * 50)

    # Collect credentials from the admin user
    username = input("\n  Username : ").strip()
    password = input("  Password : ").strip()
    totp_secret = input("  TOTP Secret: ").strip()

    if not all([username, password, totp_secret]):
        print("[!] All fields are required.")
        return

    print()
    success = authenticate(username, password, totp_secret)

    if success:
        print("\n[+] Session established - you may now issue diagnostic
commands.")
    else:
        print("\n[-] Authentication failed.")


if __name__ == "__main__":
    main()
```

Part F: Security Analysis & Reflection

1. Fragility: SYN Flood; Which component fails first?

The Socket Listener in srds_gateway.py fails first. Specifically, this line:

“server.listen(5)”

The backlog queue that has been generated by listen(5) only holds five pending connections. A SYN flood attacks with thousands of half-open TCP connections, which rapidly fill this small queue. Once the queue is completed, the operating system will no longer accept any new connections and thus even the legitimate administrators will not be able to initialize the authentication handshake.
Since the TCP handshake is never finished, authentication logics, such as the generation of nonce and MFA checks, are never performed. The nonce store dictionary and the failed

attempts tracker remains untouched because no connection is ever directed to the application layer.

2. Bypass: Exploiting the Possession Factor (MFA)

The attacker may abuse the MFA mechanism in my design by a brute-force timing attack on the TOTP code. In srds_gateway.py, the verification uses:

```
totp.verify(mfa_code)
```

This process tolerates the use of codes within a 30-second range. A 6 digit TOTP provides just one million different outcomes. Though my implementation blocks out an account after three failed attempts based on IP (traced in the failed attempts dictionary), an attacker can bypass the per-IP block by rotating through large numbers of source IPs - using a botnet or a proxy - to maintain a window of guessing MFA codes within the same 30 seconds.

3. Encryption Termination: Where is data most vulnerable to sniffing?

My data is the most susceptible between the Admin Client and the public socket of the Gateway - the Internet/TCP Socket segment that I have in my diagram. My code uses raw, unencrypted TCP sockets:

```
# Client
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Gateway
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Both sides do not have a TLS/SSL wrapping. In case there is a WAP Gateway, of any translation proxy between the client and the server, the data is sent through it in plaintext JSON. Although the raw password is never transmitted (only the SHA 256 proof), an attacker eavesdropping at the point of translation may still obtain:

- The username (Step 1 AUTH_REQ)
- The nonce (Step 2 CHALLENGE)
- The MFA code and hash proof (Step 3 AUTH_RESP)

Replay prevention (nonce volatility) defends against the re-use of packets that have been captured, however, the contents of the packet can be read completely in transit at any switching point. Step 3 AUTH_RESP is the most vulnerable one as the proof and MFA code are sent in one unencrypted JSON message.

The TOTP_SECRET has also been hard-coded in plaintext in the gateway script. Assuming the attacker can access the file system of the server in a read mode (e.g. due to some other vulnerability), the attacker may also produce valid TOTP codes, defeating the possession aspect entirely.