Data integrity

NTLMv2 Authentication Server Documentation

Team Members :

1. Walaa Ahmed Hassan 2206192 (cybersecurity)
2. Youstena Malak Tadros 2206199 (cybersecurity)
3. Ahmed Adel Abdelhady 2206194 (cybersecurity)
4. Mohamed Said Mekawy 2206201 (cybersecurity)
5. Moaz Aboelwafa Abozaid 2206195 (cybersecurity)
6. Karim Basuny Abdelrazik 2206205 (cybersecurity)
7. Samaa Mohamed Shawky 2206200 (cybersecurity)
8. Seif Ahmed Abbas 2206215 (cybersecurity)

Overview

This project implements a custom **NTLMv2-style challenge-response authentication protocol** with optional **TOTP-based two-factor authentication (2FA)** over a secure, multi-threaded TCP server.

It is designed for educational and demonstrative purposes to illustrate how authentication protocols can be securely built from the ground up.

1.Server Implementation (`server.py`)

The server is the core component that handles authentication requests using NTLMv2 with TOTP two-factor authentication.

Key Features:

- **TCP Socket Server**: Listens on a configurable port for incoming connections
- **Multi-threaded Architecture**: Handles each client connection in a separate thread
- **Stateful Authentication**: Manages the multi-step NTLMv2 authentication process
- **TOTP Verification**: Implements time-based one-time password validation
- **Logging**: Records all authentication attempts with timestamps

Detailed Workflow:

1. **Initialization**:
    - Creates a TCP socket bound to the configured host and port
    - Sets socket options for reusing addresses
    - Establishes database connection using `database.py`
2. **Client Connection Handling**:
    - Accepts new connections in a loop
    - Spawns a new thread for each client via `ClientThread` class
    - Each thread maintains its own connection state
3. **Authentication Process**:
    - **Phase 1 (Negotiation)**:
        - Receives client's username
        - Verifies username exists in database

    - **Phase 2 (Challenge)**:
        - Generates a random 8-byte challenge (nonce)
        - Retrieves stored NTLMv2 hash for the user
        - Sends challenge to client
    - **Phase 3 (Authentication)**:
        - Receives client's response and TOTP code
        - Verifies response matches expected NTLMv2 computation
        - Validates TOTP code using `totp_generator.py`
        - Sends authentication result to client
4. **Security Measures**:
    - Uses cryptographic nonces for challenge generation
    - Implements timeout for client responses
    - Limits failed attempts
    - Stores only password hashes (never plaintext)

2. Client Implementation (`client.py` and `GUI.py`)

The client components handle user interaction and protocol communication.

Console Client (`client.py`):

- **Command-line Interface**: Simple text-based interface
- **User Input Handling**: Collects username, password, and TOTP code
- **Protocol Implementation**:
    - Establishes TCP connection to server
    - Handles all three NTLMv2 phases
    - Computes NTLMv2 responses using `crypto_utils.py`
- **Result Display**: Shows authentication success/failure

Graphical Client (`GUI.py`):

- **PyQt5 Interface**: Modern GUI with input validation
- **UI Components**:
    - Username/password fields
    - TOTP code input
    - Login button
    - Status display area
- **Features**:
    - Input validation
    - Password masking
    - Responsive design
    - Clear feedback messages

3. Cryptographic Utilities (`crypto_utils.py`)

Core cryptographic operations for NTLMv2 implementation.

Key Functions:

1. **NTLMv2 Hash Generation**:

```
def generate_ntlmv2_hash(password, username, domain=""):
    # Creates the NTLMv2 hash using HMAC-MD5
    nt_hash = hashlib.new('md4', password.encode('utf-16le')).digest()
    hmac_md5 = hmac.new(nt_hash, (username.upper() + domain).encode('utf-16le'), hashlib.md
5).digest()
    return hmac_md5
```

2. **Challenge-Response Computation**:

```
def compute_ntlmv2_response(ntlmv2_hash, server_challenge, client_challenge):
    # Computes the response to server's challenge
    session_hash = hmac.new(ntlmv2_hash, server_challenge, hashlib.md5).digest()
    return hmac.new(session_hash, client_challenge, hashlib.md5).digest()
```

3. **Supporting Functions**:
    - Random byte generation
    - Hex encoding/decoding
    - Message integrity checks

4. TOTP Implementation (`totp_generator.py`)

Time-based One-Time Password functionality using RFC 6238.

Key Features:

TOTP Generation:

```python
def generate_totp(secret, interval=30, digits=6):
    # Generates current TOTP code
    counter = int(time.time()) // interval
    hmac_hash = hmac.new(secret.encode(), counter.to_bytes(8, 'big'), hashlib.sha1).digest()
    offset = hmac_hash[-1] & 0xf
    code = ((hmac_hash[offset] & 0x7f) << 24 |
            ((hmac_hash[offset+1] & 0xff) << 16) |
            ((hmac_hash[offset+2] & 0xff) << 8) |
            (hmac_hash[offset+3] & 0xff)
    return str(code % 10**digits).zfill(digits)
```

5. Database Management (database.py, db_setup.py, reset_database.py)

MySQL-based storage system for user credentials and logs.

Database Schema:

1. **Users Table**:
    - username (VARCHAR, PRIMARY KEY)
    - ntlmv2_hash (BINARY) - Stored password hash
    - totp_secret (VARCHAR) - TOTP shared secret
    - created_at (TIMESTAMP)
2. **Auth Logs Table**:
    - log_id (INT, AUTO_INCREMENT)
    - username (VARCHAR)
    - attempt_time (TIMESTAMP)
    - success (BOOLEAN)
    - ip_address (VARCHAR)

Key Operations:

- User CRUD operations
- Secure credential storage
- Authentication logging
- Database initialization/reset

6. Configuration and Utilities (config.py, utils.py)

Configuration (config.py):

```
DATABASE_CONFIG = {
    'host': 'localhost',
    'user': 'ntlm_auth',
    'password': 'securepassword',
    'database': 'ntlm_auth_db'
}

SERVER_CONFIG = {
    'host': '0.0.0.0',
    'port': 5000,
    'timeout': 30
}

TOTP_CONFIG = {
    'interval': 30,
    'digits': 6,
    'valid_window': 1
}
```

And we also have utilities.py:

- Input validation functions
- Log formatting helpers
- Common exception handling
- Data conversion utilities

7. Message Protocols (admin_messages.py, message_types.py, messages.py)

Defines the communication protocol between client and server.

Message Structure:

1. **Message Types**:
    - AUTH_INITIATE - Client starts authentication
    - CHALLENGE - Server sends nonce
    - RESPONSE - Client responds to challenge
    - AUTH_RESULT - Server sends final result
2. **Message Format**:
    - 4-byte message type
    - 4-byte message length
    - Variable-length payload

Example Message Definition:

```
class AuthChallengeMessage:
    def __init__(self, nonce):
        self.message_type = MessageTypes.CHALLENGE
        self.nonce = nonce

    def serialize(self):
        return self.message_type.value.to_bytes(4, 'big') + self.nonce
```

NTLMv2 + TOTP Authentication Protocol

This protocol implements a secure authentication mechanism by combining **NTLMv2**-style challenge-response with **Time-Based One-Time Password (TOTP)** for two-factor authentication (2FA). Below is a step-by-step explanation of the authentication flow.

1. Client Initiation Phase

Objective: Begin the authentication process by collecting and sending the initial user identity.

Steps:

1. **User Input Collection:**
    - The client (client.py or GUI.py) prompts the user to enter:
        - Username (e.g., admin)
        - Password (e.g., SecurePass123)
        - TOTP Code (6-digit code from an authenticator app such as Google Authenticator)
2. **Local NTLMv2 Hashing:**
    - The client computes a password hash locally using NTLMv2 logic:

ntlmv2_hash = HMAC_MD5(MD4(UTF-16-LE(password)), username.upper())

    - This hash is not stored or transmitted—it is computed on-the-fly.
3. **Server Connection:**
    - The client opens a TCP connection to the server (default: localhost:5000).
    - It sends an initial authentication message containing:
        - The plaintext username (used by the server to fetch stored credentials)

Security Considerations:

- Passwords are never sent over the network.
- Authentication is performed via cryptographic challenge-response, mitigating replay risks.

2. Server Challenge Phase

Objective: Validate the username and issue a challenge nonce.

Steps:

1. **Username Verification:**
    - The server checks whether the provided username exists in the database.
    - If not found, an error response is returned.
2. **Challenge Generation:**
    - If valid, the server:
        - Generates a random 8-byte nonce
        - Temporarily stores this nonce in memory for validation
3. **Challenge Response:**
    - Sends the nonce back to the client in a CHALLENGE message.

Security Considerations:

- Each nonce is unique per session to ensure freshness.
- The server never exposes stored password hashes.

3. Client Response Phase

Objective: Prove knowledge of the password and possession of the TOTP secret.

Steps:

1. **Challenge Response Calculation:**
    - The client computes:

    response = HMAC_MD5(ntlmv2_hash, server_challenge)

2. **TOTP Code Generation:**
    - The client calculates the current TOTP value:

    totp_code = pyotp.TOTP(user_totp_secret).now()

3. **Authentication Message Submission:**
    - The client sends a RESPONSE message containing:
        - The HMAC-MD5 challenge response
        - The TOTP code

Security Considerations:

- The password is never transmitted, only a derived HMAC.
- The TOTP is time-bound and valid for ~30 seconds.

4. Server Verification Phase

Objective: Validate both the password response and the TOTP code.

Steps:

1. **Challenge Response Verification:**
    - The server recalculates:

      expected_response = HMAC_MD5(stored_hash, challenge_nonce)

    - Compares this to the client's response.
2. **TOTP Verification *(if implemented)*:**
    - Verifies the TOTP code using the stored secret:

      is_valid = pyotp.TOTP(user_secret).verify(totp_code)

3. **Decision Making:**
    - **If both checks pass:**
        - Creates a session
        - Sends an AUTH_SUCCESS message

    - **If either fails:**
        - Sends an AUTH_FAILURE
        - Logs the failure in the database

Security Considerations:

- Enforces two-factor authentication
- Failed attempts are logged and may be rate-limited

5. Authentication Outcome

Objective: Finalize authentication and provide user feedback.

Server Actions:

- Logs the outcome (success or failure) with:
    - Timestamp
    - IP address
    - Username
- Sends a final AUTH_RESULT message to the client

Client Actions:

- **On success:**
    - Proceeds to secure access/session
- **On failure:**
    - Displays error message and may terminate the session

Why This Design is Secure

1. **No Password Transmission**
    - Only hashes and challenge-responses are exchanged.
    - Prevents interception of plaintext passwords.
2. **Prevents Replay Attacks**
    - Each session uses a **unique nonce**.
    - Responses are **time-sensitive** (TOTP).
3. **Two-Factor Authentication (2FA)**
    - Requires **something you know** (password).
    - Requires **something you have** (TOTP generator).
4. **Secure Cryptographic Primitives**
    - Uses **HMAC-MD5** for NTLMv2 (as per the protocol).
    - Uses **SHA-1** for TOTP (RFC 6238 compliant).