Adaptive eBPF Based Memory Forensics System

- Dwarakesh Venkatesh

- CB.SC.U4CSE23020

1. APPLICATION OVERVIEW

This project implements an Adaptive eBPF-based Memory Forensics System designed to securely monitor, store, and analyze live memory events. The system integrates authentication, multi-factor authentication, role-based access control, encryption, hashing, digital signatures, and encoding techniques to ensure confidentiality, integrity, and controlled access to forensic evidence. The application simulates real-time eBPF memory events and demonstrates secure forensic workflows aligned with foundational cybersecurity principles.

2. AUTHENTICATION (Single + MFA)

2.1 Single-Factor Authentication

Where it is implemented

- auth/login.py
- auth/password_hash.py
- app.py (/route)

The system implements single-factor authentication using a username and password mechanism. User credentials are securely stored in a SQLite database after being hashed with bcrypt and a unique salt. During login, the entered password is verified against the stored hash, ensuring that plaintext passwords are never stored or transmitted.

Why this method

- bcrypt provides built-in salting
- Resistant to brute-force and rainbow-table attacks
- Meets NIST SP 800-63-2 recommendations

2.2 Multi-Factor Authentication (MFA)

Where it is implemented

- auth/login.py → verify_mfa()
- mfa.html
- app.py (/mfa route)

Multi-factor authentication is enforced using Time-based One-Time Passwords (TOTP). After successful password verification, users must provide a time-synchronized OTP generated by an authenticator application. This ensures that access requires both **knowledge** and **possession** (OTP) significantly reducing the risk of credential compromise.

Why this method

- TOTP is industry-standard
- Lightweight and demo-friendly

3. AUTHORIZATION / ACCESS CONTROL (ACL)

3.1 Access Control Model

Where it is implemented

- users table (role column)
- app.py (role checks per route)

- Role-specific dashboards

The application implements a Role-Based Access Control (RBAC) model using an Access Control List (ACL). Three subjects - Admin, Investigator, and Auditor - are defined, each with distinct permissions over system objects such as live memory streams, encrypted dumps, and forensic reports.

Access Control Matrix (include in report)

Role	Live Memory	Encrypted Dumps	Decryption	Integrity Verification
Admin	No	Yes	Yes	No
Investigator	Yes	Yes	No	No
Auditor	No	No	No	Yes

3.2 Policy Definition & Enforcement

Where it is implemented

- @app.before_request
- Route-level role checks in app.py

Access control policies are enforced programmatically on the server side. Each request is validated against the user's role stored in the session, ensuring that unauthorized users cannot access restricted functionality even if URLs are manually manipulated.

Why

- Prevents privilege escalation
- Server-side enforcement is mandatory for security

4. ENCRYPTION & KEY MANAGEMENT

4.1 Key Generation / Exchange

Where it is implemented

- crypto/keygen.py

The system generates a secure RSA key pair using cryptographically strong random number generation. The public key is used to protect sensitive cryptographic material, while the private key is retained securely for decryption and signing operations.

Why

- RSA demonstrates asymmetric cryptography
- Clean separation of encryption and access

4.2 Encryption & Decryption

Where it is implemented

- crypto/aes_crypto.py
- /generate-report
- /decrypt-dump

Forensic memory snapshots are encrypted using AES-256 in CBC mode to ensure confidentiality at rest. Only authorized administrators can decrypt stored evidence, ensuring that sensitive memory artifacts remain protected even if storage is compromised.

Why

- AES is fast and secure for bulk data
- Hybrid approach (AES + RSA-ready)

5. HASHING & DIGITAL SIGNATURE

5.1 Hashing with Salt

Where it is implemented

- auth/password_hash.py

Passwords are hashed using bcrypt with an automatically generated salt before storage. Salting ensures that identical passwords do not produce identical hashes, mitigating precomputed hash attacks.

5.2 Digital Signature using Hash

Where it is implemented

- crypto/signature.py
- /generate-report
- /verify-report

Each encrypted forensic dump is digitally signed using an RSA private key after hashing. Auditors can verify the signature using the corresponding public key to confirm data integrity and authenticity without accessing plaintext evidence.

Why

- Separates integrity verification from confidentiality
- Mirrors real forensic audit workflows

6. ENCODING TECHNIQUES

Where it is implemented

- Live event streaming (SSE uses text encoding)
- Optional Base64-safe transport (mention in report)

Encoding techniques are used to safely transmit structured forensic data over HTTP streams. This ensures compatibility with text-based transport mechanisms and prevents data corruption during transmission.

7. SECURITY LEVELS, RISKS & ATTACKS (Theory)

The system enforces multiple security layers including authentication, authorization, encryption, and integrity verification. Potential risks include credential theft, unauthorized access, and evidence tampering. These are mitigated using MFA, role-based access control, encrypted storage, and digital signatures.

The authentication flow follows the NIST SP 800-63-2 electronic authentication model by combining identity proofing, secure credential storage, single-factor authentication, and multi-factor verification before granting access to protected resources