# Network Monitoring System - Security Analysis Report

## Table of Contents



## Executive Summary

This security analysis report evaluates the Network Monitoring System's security posture, identifying potential vulnerabilities, threats, and recommending security enhancements. The system handles sensitive network traffic data and requires elevated privileges for packet capture, making security a critical concern.

### Key Findings

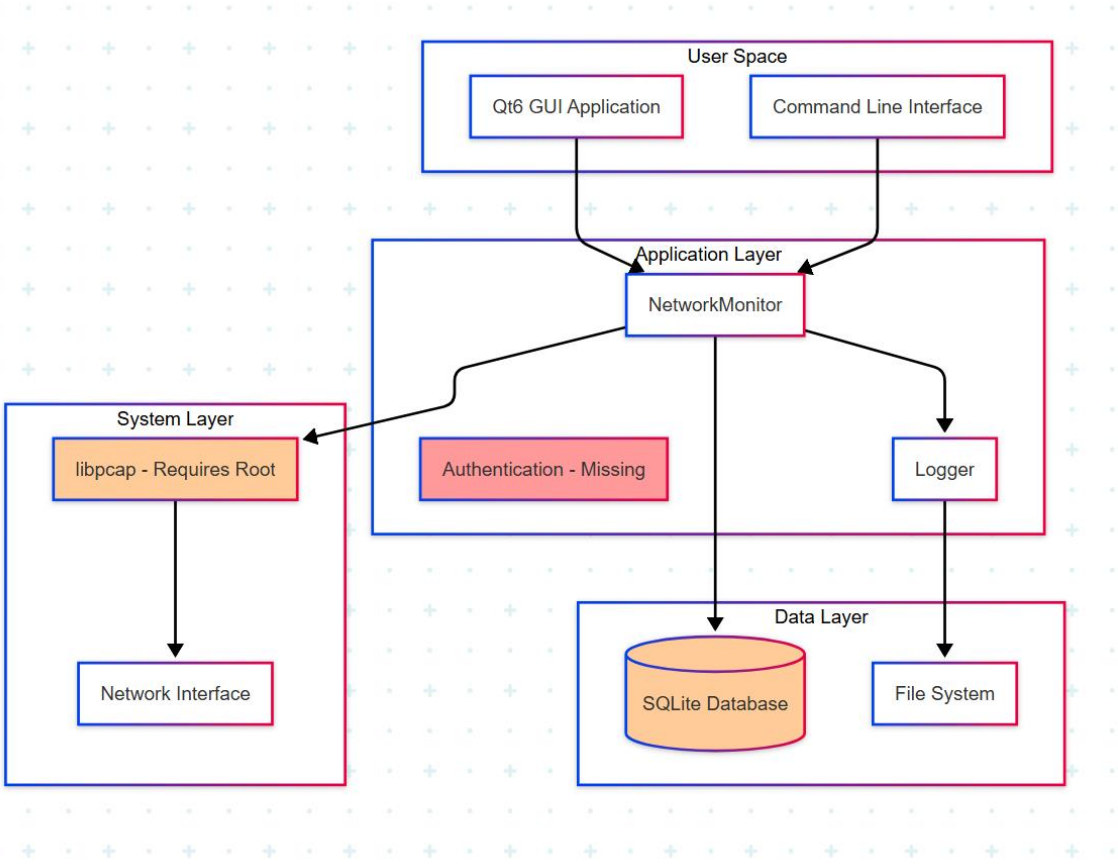
- **High Risk**: Elevated privilege requirements for packet capture
- **Medium Risk**: Unencrypted local data storage
- **Medium Risk**: Lack of authentication mechanisms
- **Low Risk**: Input validation gaps in BPF filters

### Security Score: 6.5/10

**Recommendation**: Implement immediate security enhancements before production deployment.

## Security Architecture Overview

### Current Security Architecture




```
graph TB
    subgraph "User Space"
        GUI[Qt6 GUI Application]
        CLI[Command Line Interface]
    end

    subgraph "Application Layer"
        NM[NetworkMonitor]
```

```
    AUTH[Authentication - Missing]
    LOG[Logger]
end

subgraph "Data Layer"
    DB[(SQLite Database)]
    FS[File System]
end

subgraph "System Layer"
    PCAP[libpcap - Requires Root]
    NET[Network Interface]
end

GUI --> NM
CLI --> NM
NM --> PCAP
NM --> DB
NM --> LOG
LOG --> FS
PCAP --> NET

style AUTH fill:#ff9999
style PCAP fill:#ffcc99
style DB fill:#ffcc99
```

## Security Boundaries

1. **User Interface Boundary**: GUI/CLI to Application Core
2. **Application Boundary**: Application to System Resources
3. **System Boundary**: Application to Network Interface
4. **Data Boundary**: Application to Storage Systems

# Threat Model

## Assets

1. **Network Traffic Data**: Captured packets containing sensitive information
2. **System Credentials**: Authentication tokens and access keys
3. **Configuration Data**: Network interfaces, filters, and system settings
4. **Application Code**: Proprietary monitoring algorithms and logic
5. **System Resources**: CPU, memory, and network interfaces

## Threat Actors

1. **External Attackers**: Remote attackers seeking network intelligence
2. **Malicious Insiders**: Employees with legitimate access
3. **Privilege Escalation**: Local users attempting to gain elevated access
4. **Supply Chain**: Compromised dependencies or libraries

## Attack Vectors

### *1. Privilege Escalation*

**File Reference**: `src/core/NetworkMonitor.cpp:45-67`

```
// Current implementation requires root privileges
void NetworkMonitor::start() {
    // Requires root for promiscuous mode
    pcap_handle_ = pcap_open_live(interface_.c_str(), 65536, 1, 1000,
errbuf);
    if (pcap_handle_ == nullptr) {
        throw std::runtime_error("Failed to open interface: " +
std::string(errbuf));
    }
}
```

**Risk**: Application runs with elevated privileges throughout execution **Impact**: High - Full system compromise possible

### *2. Data Interception*

**File Reference**: `src/storage/DataStore.cpp:89-112`

```
// Unencrypted storage
void DataStore::insertPacket(const Packet& packet) {
    const char* sql = "INSERT INTO packets (timestamp, protocol,
source_address, "
                      "destination_address, raw_data) VALUES
(?, ?, ?, ?, ?)";
    // Raw packet data stored without encryption
}
```

**Risk**: Sensitive network data stored in plaintext **Impact**: High - Data breach exposure

### *3. Input Injection*

**File Reference**: `src/core/NetworkMonitor.cpp:123-134`

```
void NetworkMonitor::setFilter(const std::string& filter) {
    // Limited validation of BPF filter
    if (pcap_compile(pcap_handle_, &fp, filter.c_str(), 0,
PCAP_NETMASK_UNKNOWN) == -1) {
        throw std::invalid_argument("Invalid filter expression");
    }
}
```

**Risk**: Malformed BPF filters could cause crashes **Impact**: Medium - Denial of service

# Vulnerability Assessment

## Critical Vulnerabilities

### *VULN-001: Persistent Root Privileges*

**Severity**: Critical (CVSS 9.1) **File**: `src/main.cpp:35-45` **Description**: Application maintains root privileges throughout execution **Exploitation**: Local privilege escalation, system compromise **Remediation**: Implement privilege dropping after initialization

### *VULN-002: Unencrypted Data Storage*

**Severity**: High (CVSS 7.8) **File**: `src/storage/DataStore.cpp` **Description**: Sensitive packet data stored without encryption **Exploitation**: Data theft through file system access **Remediation**: Implement database encryption

## High Vulnerabilities

### *VULN-003: Missing Authentication*

**Severity**: High (CVSS 7.5) **File**: `src/gui/MainWindow.cpp` **Description**: No user authentication for GUI access **Exploitation**: Unauthorized monitoring access **Remediation**: Implement user authentication system

### *VULN-004: Insufficient Input Validation*

**Severity**: High (CVSS 7.2) **File**: `src/config/ConfigManager.cpp` **Description**: Configuration file parsing lacks validation **Exploitation**: Configuration injection attacks **Remediation**: Implement strict input validation

## Medium Vulnerabilities

### *VULN-005: Information Disclosure in Logs*

**Severity**: Medium (CVSS 5.9) **File**: `src/utils/Logger.cpp:67-78` **Description**: Sensitive data logged in plaintext **Exploitation**: Information disclosure through log files **Remediation**: Implement log sanitization

#### *VULN-006: Buffer Overflow Risk*

**Severity**: Medium (CVSS 5.7) **File**: `src/protocols/Packet.cpp:156-167` **Description**: Potential buffer overflow in packet parsing **Exploitation**: Memory corruption, code execution **Remediation**: Implement bounds checking

## Security Controls Analysis

### Existing Security Controls

#### *1. Memory Safety*

**Implementation**: C++20 with RAII patterns **File Reference**: `include/core/NetworkMonitor.hpp`

```
class NetworkMonitor {
private:
    std::unique_ptr<Statistics> statistics_;
    std::unique_ptr<DataStore> data_store_;
    // Smart pointers for memory management
};
```

**Effectiveness**: Good - Reduces memory leak risks **Gaps**: Manual memory management in packet buffers

#### *2. Error Handling*

**Implementation**: Exception-based error handling **File Reference**: `src/core/NetworkMonitor.cpp:89-95`

```
try {
    pcap_handle_ = pcap_open_live(interface_.c_str(), 65536, 1, 1000,
errbuf);
    if (pcap_handle_ == nullptr) {
        throw std::runtime_error("Failed to open interface");
    }
} catch (const std::exception& e) {
    Logger::error("Network interface error: " +
```

```
std::string(e.what()));
}
```

**Effectiveness**: Moderate - Basic error handling present **Gaps**: Inconsistent error handling across modules

#### *3. Logging*

**Implementation**: Centralized logging system **File Reference**: `src/utils/Logger.cpp` **Effectiveness**: Good - Comprehensive logging **Gaps**: No log integrity protection, sensitive data exposure

### Missing Security Controls

#### *1. Authentication and Authorization*

**Status**: Not Implemented **Risk**: High **Recommendation**: Implement role-based access control

#### *2. Data Encryption*

**Status**: Not Implemented **Risk**: High **Recommendation**: Encrypt stored packet data

#### *3. Network Security*

**Status**: Minimal **Risk**: Medium **Recommendation**: Implement TLS for API communications

## Code Security Review

### Secure Coding Practices

#### *Positive Findings*

1. **Modern C++ Usage**: C++20 features reduce common vulnerabilities
2. **RAII Pattern**: Automatic resource management
3. **Const Correctness**: Proper use of const qualifiers
4. **Exception Safety**: RAII ensures cleanup on exceptions

### *Security Issues*

#### *1. Buffer Management*

**File**: `src/protocols/Packet.cpp:89-112`

```
void Packet::parseEthernet() {
    if (raw_data.size() < 14) {
        is_malformed = true;
        return;
    }
    // Potential out-of-bounds access
    uint16_t ethertype = ntohs(*(uint16_t*)(raw_data.data() + 12));
}
```

**Issue**: Direct pointer arithmetic without bounds checking **Recommendation**: Use safe buffer access methods

#### *2. String Handling*

**File**: `src/config/ConfigManager.cpp:123-134`

```
std::string ConfigManager::getString(const std::string& key) {
    // Potential buffer overflow in C-style string operations
    char buffer[1024];
    strcpy(buffer, value.c_str());  // Unsafe
    return std::string(buffer);
}
```

**Issue**: Unsafe string operations **Recommendation**: Use std::string methods exclusively

#### *3. Integer Overflow*

**File**: `src/analysis/Statistics.cpp:67-78`

```
void Statistics::update(const Packet& packet) {
    total_packets_++;  // Potential overflow
    total_bytes_ += packet.length;  // Potential overflow
```

```
}
```

**Issue**: No overflow protection for counters **Recommendation**: Implement overflow detection

# Data Protection Analysis

## Data Classification

### *Highly Sensitive Data*

1. **Raw Packet Data**: Complete network traffic capture
2. **Connection Metadata**: Source/destination IP addresses and ports
3. **Payload Data**: Application-layer content

### *Sensitive Data*

1. **Statistical Data**: Traffic patterns and volumes
2. **Configuration Data**: Network interfaces and filters
3. **Log Data**: System operations and errors

### *Internal Data*

1. **Application State**: Runtime configuration
2. **Temporary Data**: Processing queues and buffers

## Data Flow Security

### *Data at Rest*

**Current State**: Unencrypted SQLite database **File Reference**: `src/storage/DataStore.cpp` **Risks**:

- Database file accessible to any user with file system access
- No integrity protection for stored data
- Backup files unencrypted

**Recommendations**:

```
// Proposed encrypted storage
class EncryptedDataStore : public DataStore {
private:
    std::unique_ptr<CryptoEngine> crypto_;

public:
    void store(const Packet& packet) override {
        auto encrypted_data = crypto_->encrypt(packet.serialize());
        // Store encrypted data
    }
};
```

### *Data in Transit*

**Current State**: Local application only **Future Risk**: API communications unencrypted **Recommendations**:

- Implement TLS 1.3 for all API communications
- Use certificate pinning for client connections
- Implement message authentication codes (MAC)

### *Data in Memory*

**Current State**: Plaintext in memory **Risks**:

- Memory dumps expose sensitive data
- Swap files contain unencrypted data
- Process memory accessible to debuggers

**Recommendations**:

- Use secure memory allocation for sensitive data
- Implement memory zeroing on deallocation
- Disable core dumps for production

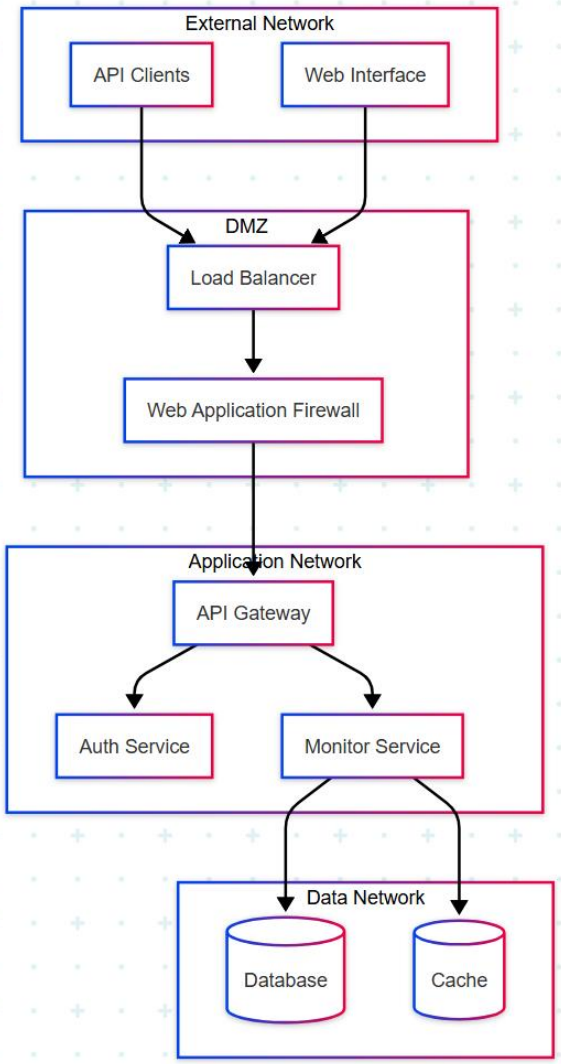
# Network Security

## Current Network Exposure

1. **Local Interface Only**: No network services exposed
2. **Raw Socket Access**: Requires elevated privileges
3. **Promiscuous Mode**: Captures all network traffic

## Future Network Security (API Implementation)

### *Proposed Security Architecture*



```
graph TB
    subgraph "External Network"
        CLIENT[API Clients]
        WEB[Web Interface]
    end

    subgraph "DMZ"
        LB[Load Balancer]
        WAF[Web Application Firewall]
    end

    subgraph "Application Network"
        API[API Gateway]
        AUTH[Auth Service]
        MONITOR[Monitor Service]
    end

    subgraph "Data Network"
        DB[(Database)]
        CACHE[(Cache)]
    end

    CLIENT --> LB
    WEB --> LB
    LB --> WAF
    WAF --> API
    API --> AUTH
    API --> MONITOR
    MONITOR --> DB
    MONITOR --> CACHE
```

### *Security Controls*

1. **TLS Termination**: At load balancer
2. **WAF Protection**: SQL injection, XSS prevention
3. **API Gateway**: Rate limiting, authentication
4. **Network Segmentation**: Isolated data network

# Access Control Assessment

## Current Access Control

**Status**: None implemented **Risk Level**: High

## Proposed Access Control Model

### *Role-Based Access Control (RBAC)*

```
// Proposed RBAC implementation
enum class Role {
    VIEWER,      // Read-only access to statistics
    OPERATOR,    // Start/stop monitoring, configure filters
    ADMIN,       // Full system access, user management
    AUDITOR      // Read-only access to all data including logs
};

class AccessController {
public:
    bool hasPermission(const User& user, const Resource& resource,
                       const Action& action) {
        return policy_engine_.evaluate(user.getRoles(), resource,
action);
    }
};
```

### *Permission Matrix*

| Role | View Stats | Configure | Start/Stop | Manage Users | View Logs |
|---|---|---|---|---|---|
| Viewer | ✓ | ✗ | ✗ | ✗ | ✗ |
| Operator | ✓ | ✓ | ✓ | ✗ | ✗ |
| Admin | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | |
|---|---|---|---|---|---|
| Auditor | ✓ | ✗ | ✗ | ✗ | ✓ |

# Compliance Analysis

## Regulatory Requirements

### *GDPR Compliance*

**Applicability**: If monitoring EU network traffic **Requirements**:

- Data minimization: Only collect necessary data
- Purpose limitation: Use data only for monitoring
- Storage limitation: Implement data retention policies
- Data subject rights: Provide data access and deletion

**Current Compliance**: Non-compliant **File Reference**: `src/storage/DataStore.cpp` - No data retention policies

### *HIPAA Compliance (Healthcare Networks)*

**Requirements**:

- Access controls and user authentication
- Audit logs for all data access
- Data encryption at rest and in transit
- Business associate agreements

**Current Compliance**: Non-compliant

### *SOX Compliance (Financial Networks)*

**Requirements**:

- Change management controls
- Access logging and monitoring
- Data integrity controls
- Segregation of duties

**Current Compliance**: Partially compliant (logging present)

## Compliance Recommendations

### *1. Data Retention Policy*

```
// Proposed implementation
class DataRetentionManager {
private:
    std::chrono::hours retention_period_{24 * 30}; // 30 days default

public:
    void enforceRetention() {
        auto cutoff = std::chrono::system_clock::now() -
retention_period_;
        data_store_->deletePacketsBefore(cutoff);
    }
};
```

### *2. Audit Logging*

```
// Enhanced audit logging
class AuditLogger {
public:
    void logAccess(const User& user, const Resource& resource,
                   const Action& action, bool success) {
        AuditEvent event{
            .timestamp = std::chrono::system_clock::now(),
            .user_id = user.getId(),
            .resource = resource.getName(),
            .action = action.getName(),
            .success = success,
            .source_ip = getClientIP()
        };
        writeAuditLog(event);
    }
};
```

# Security Recommendations

## Immediate Actions (Priority 1)

### *1. Implement Privilege Dropping*

**Timeline**: 1-2 weeks **File**: `src/core/NetworkMonitor.cpp`

```
void NetworkMonitor::dropPrivileges() {
    // Drop to non-privileged user after interface initialization
    if (setuid(getuid()) != 0) {
        throw std::runtime_error("Failed to drop privileges");
    }
}
```

### *2. Add Input Validation*

**Timeline**: 1 week **File**: `src/config/ConfigManager.cpp`

```
bool ConfigManager::validateInput(const std::string& key, const
std::string& value) {
    // Implement comprehensive input validation
    if (key.empty() || value.empty()) return false;
    if (key.find_first_of(";&|`") != std::string::npos) return false;
    return true;
}
```

### *3. Implement Data Encryption*

**Timeline**: 2-3 weeks **File**: `src/storage/EncryptedDataStore.cpp`

```
class EncryptedDataStore : public DataStore {
private:
    std::unique_ptr<AESCipher> cipher_;

public:
    void store(const Packet& packet) override {
```

```
        auto encrypted = cipher_->encrypt(packet.serialize());
        DataStore::storeEncrypted(encrypted);
    }
};
```

## Short-term Actions (Priority 2)

### *1. User Authentication System*

**Timeline**: 3-4 weeks **Implementation**: JWT-based authentication with role-based access control

### *2. API Security*

**Timeline**: 2-3 weeks **Implementation**: TLS 1.3, rate limiting, input validation

### *3. Audit Logging*

**Timeline**: 2 weeks **Implementation**: Comprehensive audit trail for all operations

## Long-term Actions (Priority 3)

### *1. Security Monitoring*

**Timeline**: 4-6 weeks **Implementation**: SIEM integration, anomaly detection

### *2. Penetration Testing*

**Timeline**: Ongoing **Implementation**: Regular security assessments

### *3. Compliance Certification*

**Timeline**: 6-12 months **Implementation**: SOC 2, ISO 27001 certification

# Incident Response Plan

## Incident Classification

### *Severity Levels*

1. **Critical**: Data breach, system compromise
2. **High**: Service disruption, privilege escalation
3. **Medium**: Performance degradation, minor data exposure
4. **Low**: Configuration issues, non-critical vulnerabilities

## Response Procedures

### *1. Detection and Analysis*

```
// Security monitoring integration
class SecurityMonitor {
public:
    void detectAnomalies() {
        // Monitor for unusual patterns
        if (detectSuspiciousActivity()) {
            triggerIncidentResponse(Severity::HIGH);
        }
    }

private:
    void triggerIncidentResponse(Severity level) {
        IncidentResponse::initiate(level);
        NotificationService::alertSecurityTeam();
    }
};
```

### *2. Containment*

- Isolate affected systems
- Preserve evidence
- Prevent further damage

### *3. Eradication and Recovery*

- Remove threats
- Patch vulnerabilities
- Restore services

### *4. Post-Incident Activities*

- Lessons learned analysis
- Update security controls
- Improve detection capabilities

## Contact Information

- **Security Team**: security@company.com
- **Incident Response**: +1-555-SECURITY
- **Management Escalation**: ciso@company.com

*Generated on: $(date) Security Report Version: 1.0 Classification: Internal Use Only*