Security Technical Operational Report

# Technical Security Assessment Report

## Technical Summary This report details the findings of a technical security assessment conducted on the MedSecure Hospital Management System repository on October 12, 2025. The scan identified a total of 13 vulnerabilities and misconfigurations, with a severity breakdown of 2 High, 3 Medium, 2 Low, and 6 Informational findings.

The most significant risks identified include the use of a cryptographically broken hashing algorithm (MD5) for password storage and a configuration that could enable remote code execution in a production environment (Flask debug mode). Additionally, medium-risk vulnerabilities such as a pattern susceptible to SQL injection and the use of an outdated, insecure cryptographic dependency (`pycrypto`) were found. These issues present a direct threat to the confidentiality, integrity, and availability of Protected Health Information (PHI), posing a significant compliance risk under HIPAA's Technical Safeguards. Immediate remediation of the high-risk findings is strongly recommended to mitigate the risk of data breach and unauthorized system access.

## Vulnerability Analysis

### Code Security Issues Static analysis of the application source code revealed several critical flaws in security practices. High-risk issues include the use of the deprecated MD5 algorithm for hashing credentials and running the Flask web framework in debug mode, which could expose a remote debugger. A medium-risk SQL injection vulnerability was identified due to the use of unsafe string formatting to construct database queries. Furthermore, the application insecurely handles temporary file storage by using a world-writable directory (`/tmp`), creating a risk of information disclosure or file tampering. These findings indicate a need for developer training and the enforcement of secure coding standards.

### Dependency Vulnerabilities The application relies on several third-party libraries, some of which introduce security risks. A medium-risk vulnerability was identified due to the inclusion of `pycrypto==2.6.1`, an unmaintained library with known security flaws. The modern and secure fork, `PyCryptodome`, should be used instead. While the JavaScript dependencies (`express`, `lodash`) did not have specific CVEs flagged in this scan, the use of floating versions (e.g., `^4.18.2`) can lead to unintended, potentially vulnerable versions being installed. A comprehensive Software Composition Analysis (SCA) tool should be integrated into the CI/CD pipeline to continuously monitor for new vulnerabilities in all dependencies.

### Architecture Security Review The identified vulnerabilities suggest several architectural weaknesses that deviate from security best practices and HIPAA requirements. The lack of a standardized and secure database abstraction layer has led to SQL injection flaws. Credential management is fundamentally insecure, violating the principle of storing secrets safely. The application's configuration management does not adequately separate development settings (debug mode) from production requirements, a critical oversight for a system handling PHI. The overall architecture would benefit from a defense-in-depth approach, incorporating secure-by-default configurations, centralized input validation, and robust cryptographic standards as outlined in the project's own `hipaa\_compliance.md` policy.

## Detailed Technical Findings

### Critical Vulnerabilities No vulnerabilities classified as Critical were identified during this assessment.

### High-Risk Issues \*\*1. Weak Cryptographic Hash (MD5) Used for Passwords\*\* \* \*\*ID:\*\* B324 \* \*\*File:\*\* `app/auth.py` \* \*\*Line:\*\* 9 \* \*\*Description:\*\* The application stores user password hashes using the MD5 algorithm. MD5 is considered cryptographically broken and is highly susceptible to collision and rainbow table attacks, allowing for rapid recovery of the original password. \* \*\*Impact:\*\* An attacker gaining access to the password hashes could easily crack them, leading to complete account compromise. In a healthcare context, this could result in unauthorized access to sensitive patient PHI, constituting a severe data breach. \* \*\*Context:\*\* `9 'dr.jones': {'password\_hash': hashlib.md5('password123'.encode()).hexdigest(), 'role': 'clinician'}`

\*\*2. Flask Application Running in Debug Mode\*\*  
\* \*\*ID:\*\* B201  
\* \*\*File:\*\* `app/app.py`  
\* \*\*Line:\*\* 16  
\* \*\*Description:\*\* The application is configured with `debug=True`. In a production environment, this setting can expose a Werkzeug debugger if an unhandled exception occurs. This debugger allows for the inspection of application state and the execution of arbitrary Python code.  
\* \*\*Impact:\*\* This misconfiguration could allow an attacker to achieve Remote Code Execution (RCE) on the server, granting them full control over the application, its data, and the underlying system.  
\* \*\*Context:\*\* `16 app.run(debug=True)`

### Medium-Risk Issues \*\*1. SQL Injection via String Formatting\*\* \* \*\*ID:\*\* B608 \* \*\*File:\*\* `app/db.py` \* \*\*Line:\*\* 5 \* \*\*Description:\*\* A database query is constructed by embedding a variable directly into the SQL string using an f-string. This pattern does not sanitize the input, making it vulnerable to SQL Injection. \* \*\*Impact:\*\* An attacker could manipulate the `mrn` parameter to execute arbitrary SQL commands, allowing them to bypass authentication, exfiltrate, modify, or delete any data in the database, including all patient records. \* \*\*Context:\*\* `5 return f"SELECT \* FROM patients WHERE mrn = '{mrn}'"`

\*\*2. Insecure Temporary File Storage\*\*  
\* \*\*ID:\*\* B108  
\* \*\*File:\*\* `app/file\_upload.py`  
\* \*\*Line:\*\* 6  
\* \*\*Description:\*\* The application uses the publicly writable `/tmp/` directory for storing uploads. This can be exploited through race conditions or symlink attacks by other users on the same system.  
\* \*\*Impact:\*\* An attacker with local access could potentially read sensitive uploaded files before they are processed or trick the application into overwriting other critical system files.  
\* \*\*Context:\*\* `6 UPLOAD\_DIR = '/tmp/medsecure\_uploads'`

\*\*3. Use of Deprecated and Insecure `pycrypto` Library\*\*  
\* \*\*ID:\*\* N/A  
\* \*\*File:\*\* `requirements.txt`  
\* \*\*Line:\*\* 6  
\* \*\*Description:\*\* The project depends on `pycrypto==2.6.1`, which is an old, unmaintained library with multiple known vulnerabilities.  
\* \*\*Impact:\*\* Known vulnerabilities in this library could be exploited to compromise cryptographic operations, potentially leading to information disclosure or other security breaches.  
\* \*\*Context:\*\* `pycrypto==2.6.1`

## Security Testing Results The automated security scan was performed using the `bandit` static analysis tool, supplemented by dependency analysis. The scan identified a total of 13 issues, categorized by severity below.

| Severity | Count |  
|-------------|-------|  
| HIGH | 2 |  
| MEDIUM | 3 |  
| LOW | 2 |  
| INFO | 6 |  
| \*\*Total\*\* | \*\*13\*\* |

## Technical Recommendations Remediation should be prioritized based on risk. The following actions are recommended:

1. \*\*Immediate (High-Risk): Remediate Credential Storage.\*\* Replace the MD5 hashing algorithm with a modern, salted, and adaptive password hashing function such as Argon2 or bcrypt. This is critical for protecting user credentials and complying with HIPAA.  
2. \*\*Immediate (High-Risk): Disable Debug Mode.\*\* Ensure that the Flask application's debug mode is disabled in all production and staging environments. Use environment variables to manage this configuration securely.  
3. \*\*Required (Medium-Risk): Implement Parameterized Queries.\*\* Refactor all database access code to use parameterized queries (prepared statements). This is the standard defense against SQL injection and should be applied universally.  
4. \*\*Required (Medium-Risk): Migrate from `pycrypto`.\*\* Uninstall `pycrypto` and replace it with its actively maintained and secure successor, `PyCryptodome`. Update all relevant import statements in the code.  
5. \*\*Required (Medium-Risk): Use Secure Temporary Directories.\*\* Replace the hardcoded `/tmp/` path with Python's `tempfile` module to create secure, temporary files and directories with appropriate permissions.  
6. \*\*Best Practice: Implement Dependency Scanning.\*\* Integrate a Software Composition Analysis (SCA) tool into the development pipeline to automatically scan `requirements.txt` and `package.json` for known vulnerabilities on an ongoing basis.

## Implementation Guidelines

\*\*1. For Stronger Password Hashing (replaces MD5):\*\*  
Use a library like `passlib` to simplify implementation.  
```python  
# Installation: pip install passlib[bcrypt]  
from passlib.context import CryptContext

# Create a context for hashing pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# Hashing a new password hashed\_password = pwd\_context.hash("new\_secure\_password")

# Verifying a password is\_valid = pwd\_context.verify("submitted\_password", hashed\_password) ```

\*\*2. For Preventing SQL Injection (replaces f-string):\*\*  
Use placeholders in the query and pass values as a separate tuple.  
```python  
# Before (Vulnerable)  
# return f"SELECT \* FROM patients WHERE mrn = '{mrn}'"

# After (Secure) def get\_patient\_query(cursor, mrn): query = "SELECT \* FROM patients WHERE mrn = %s" cursor.execute(query, (mrn,)) return cursor.fetchall() ```

\*\*3. For Migrating from `pycrypto`:\*\*  
First, update your dependencies.  
```bash  
pip uninstall pycrypto  
pip install pycryptodome  
```  
Then, update your import statements in the code.  
```python  
# Before  
# from Crypto.Cipher import AES

# After from Cryptodome.Cipher import AES ```

\*\*4. For Disabling Debug Mode:\*\*  
Manage the debug flag using an environment variable.  
```python  
# In app.py  
import os  
# Set debug to False unless FLASK\_DEBUG is explicitly set to '1'  
is\_debug = os.environ.get('FLASK\_DEBUG') == '1'  
app.run(debug=is\_debug)

## In production environment startup script: # unset FLASK\_DEBUG or export FLASK\_DEBUG=0 ```

## Security Monitoring and Alerting To ensure ongoing security and compliance with the `Breach Response Policy`, the following monitoring and alerting mechanisms are recommended:

\* \*\*Application-Level Logging:\*\* Implement structured logging for all security-sensitive events, including user logins (success and failure), password change attempts, and access to PHI. Ensure logs are stripped of sensitive data and forwarded to a centralized Security Information and Event Management (SIEM) system.  
\* \*\*Web Application Firewall (WAF):\*\* Deploy a WAF to inspect incoming traffic and block common attack patterns, such as SQL injection payloads and cross-site scripting (XSS) attempts.  
\* \*\*Alerting Rules:\*\* Configure alerts in your SIEM or monitoring solution for high-risk security events:  
 \* Multiple failed login attempts for a single account or from a single IP address within a short time frame.  
 \* Any application error that triggers a `5xx` server response, which could indicate an attempted exploit.  
 \* Execution of suspicious commands or processes on the application server.  
\* \*\*File Integrity Monitoring (FIM):\*\* Implement FIM on the application servers to detect and alert on unauthorized changes to application code, configurations, or system files.