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. The automated scan identified a total of 18 security findings, categorized by severity as follows: 1 Critical, 4 High, 5 Medium, 2 Low, and 6 Informational.

The most severe issues discovered include the use of hardcoded credentials (a JWT secret key), which poses a direct and immediate risk of system compromise. Several high-risk vulnerabilities were also identified, including the use of a weak password hashing algorithm (MD5), enabled application debug mode that could expose sensitive information and allow arbitrary code execution, and a potential Cross-Site Scripting (XSS) vulnerability.

Given the application's context within the healthcare sector, these vulnerabilities present a significant risk to the confidentiality, integrity, and availability of Protected Health Information (PHI), and may represent non-compliance with technical safeguards outlined in HIPAA regulations. Immediate remediation of the critical and high-risk findings is strongly recommended.

## Vulnerability Analysis

### Code Security Issues The static analysis of the application's source code revealed several significant security flaws. A critical vulnerability was found in `app/auth.py` where a JWT secret is hardcoded directly into the source. High-risk issues include the use of the deprecated and insecure MD5 hashing algorithm for user passwords, also in `app/auth.py`, and the enabling of debug mode in `app/app.py`, which could lead to remote code execution. Furthermore, the analysis identified a potential SQL injection vulnerability in `app/db.py` due to the use of unsafe string formatting to construct database queries. Finally, a potential Cross-Site Scripting (XSS) issue was noted in `app/patient\_routes.py`, and insecure temporary file handling was identified in `app/file\_upload.py`. These findings indicate systemic weaknesses in secure coding practices.

### Dependency Vulnerabilities The Software Composition Analysis (SCA) identified vulnerabilities within the project's third-party dependencies. The Python dependency `pycrypto==2.6.1` listed in `requirements.txt` is outdated and known to have security weaknesses. It is strongly recommended to migrate to a maintained library such as `pycryptodome`. While no specific CVEs were flagged for the JavaScript dependencies (`express` and `lodash`) in this scan, their versions should be continuously monitored for newly discovered vulnerabilities. Overall, the project's dependency management lacks automated vulnerability scanning, which is essential for maintaining a secure posture.

### Architecture Security Review The identified vulnerabilities suggest several architectural weaknesses that should be addressed. 1. \*\*Lack of Secrets Management:\*\* The hardcoded JWT secret indicates the absence of a secure system for managing application secrets. A robust architecture would leverage environment variables, a secrets management service (e.g., HashiCorp Vault, AWS Secrets Manager), or encrypted configuration files instead of embedding sensitive data in code. 2. \*\*Insecure Data Handling:\*\* The use of MD5 for password hashing is a direct violation of modern security standards and contradicts the project's stated HIPAA policy of using AES-256 for data at rest. Similarly, the construction of raw SQL queries without parameterization points to an insecure data access layer. 3. \*\*Unsafe File Upload Mechanism:\*\* The use of a predictable, world-writable directory (`/tmp/`) for file uploads is an insecure design pattern that can lead to race conditions, information disclosure, or denial-of-service attacks. 4. \*\*Insufficient Environment Separation:\*\* Enabling debug mode in the main application entry point suggests a lack of separation between development and production configurations, a practice that frequently leads to production security incidents.

## Detailed Technical Findings

### Critical Vulnerabilities \*\*1. Hardcoded Credentials\*\* \* \*\*Severity:\*\* CRITICAL \* \*\*Location:\*\* `app/auth.py:6` \* \*\*Description:\*\* A sensitive JWT secret key is hardcoded directly in the source code. This makes the secret accessible to anyone with read access to the codebase, including developers, version control systems, and potentially unauthorized actors. Compromise of this key would allow an attacker to forge authentication tokens and gain unauthorized access to the application and its data. \* \*\*Evidence:\*\* `JWT\_SECRET = 'demo\_jwt\_secret'` \* \*\*Impact:\*\* An attacker with this secret can create valid JWTs for any user, bypassing all authentication controls and gaining privileged access to sensitive patient data. \* \*\*Compliance:\*\* CWE-798 (Use of Hard-coded Credentials), OWASP Top 10 (A07:2021 – Identification and Authentication Failures)

### High-Risk Issues \*\*1. Debug Mode is Enabled in a Production Setting\*\* \* \*\*Severity:\*\* HIGH \* \*\*Location:\*\* `app/app.py:16` \* \*\*Description:\*\* The Flask application is configured to run with `debug=True`. When in debug mode, Flask's Werkzeug development server provides an interactive debugger in the browser upon an unhandled exception. This debugger allows for the execution of arbitrary Python code on the server, leading to a full remote code execution (RCE) vulnerability. \* \*\*Evidence:\*\* `app.run(debug=True)` \* \*\*Impact:\*\* A successful exploit would grant an attacker complete control over the application server, enabling data exfiltration, system compromise, and further lateral movement within the network. \* \*\*Compliance:\*\* OWASP Top 10 (A05:2021 – Security Misconfiguration)

\*\*2. Use of Weak Cryptographic Hash for Passwords\*\*  
\* \*\*Severity:\*\* HIGH  
\* \*\*Location:\*\* `app/auth.py:9`  
\* \*\*Description:\*\* The application uses the MD5 algorithm to hash user passwords. MD5 is a cryptographically broken hashing function that is highly susceptible to collision attacks and can be cracked in seconds using modern hardware or rainbow tables.  
\* \*\*Evidence:\*\* `'password\_hash': hashlib.md5('password123'.encode()).hexdigest()`  
\* \*\*Impact:\*\* In the event of a database breach, an attacker could easily recover the original passwords for all users, leading to widespread account compromise. This is a direct violation of HIPAA's technical safeguards for protecting ePHI.  
\* \*\*Compliance:\*\* CWE-327 (Use of a Broken or Risky Cryptographic Algorithm), OWASP Top 10 (A02:2021 – Cryptographic Failures)

\*\*3. Potential Cross-Site Scripting (XSS) via Unescaped Template Rendering\*\*  
\* \*\*Severity:\*\* HIGH  
\* \*\*Location:\*\* `app/patient\_routes.py:12`  
\* \*\*Description:\*\* User-controlled data is rendered directly into an HTML template using `render\_template\_string` without proper output encoding or escaping. This allows an attacker to inject malicious JavaScript into the web page, which would then execute in the browser of other users.  
\* \*\*Evidence:\*\* `return render\_template\_string(template)`  
\* \*\*Impact:\*\* An attacker could steal session cookies, impersonate other users, deface the website, or redirect users to malicious sites. In a healthcare context, this could be used to trick clinicians into viewing or entering incorrect patient data.  
\* \*\*Compliance:\*\* CWE-79 (Improper Neutralization of Input During Web Page Generation), OWASP Top 10 (A03:2021 – Injection)

### Medium-Risk Issues \*\*1. Potential SQL Injection via String Formatting\*\* \* \*\*Severity:\*\* MEDIUM \* \*\*Location:\*\* `app/db.py:5` \* \*\*Description:\*\* A SQL query is constructed by directly embedding a variable into a query string using an f-string. If the `mrn` variable contains malicious SQL syntax, it could be executed by the database, leading to a SQL injection vulnerability. \* \*\*Evidence:\*\* `return f"SELECT \* FROM patients WHERE mrn = '{mrn}'"` \* \*\*Impact:\*\* An attacker could bypass authentication, read, modify, or delete sensitive data from the database, including all patient records (ePHI). \* \*\*Compliance:\*\* CWE-89 (Improper Neutralization of Special Elements used in an SQL Command), OWASP Top 10 (A03:2021 – Injection)

\*\*2. Insecure Use of Temporary File Directory\*\*  
\* \*\*Severity:\*\* MEDIUM  
\* \*\*Location:\*\* `app/file\_upload.py:6`  
\* \*\*Description:\*\* The application is configured to use the global `/tmp/` directory for file uploads. This directory is often world-writable, making it possible for other users or processes on the same server to access, modify, or delete uploaded files before they are processed. This can lead to race conditions and information disclosure.  
\* \*\*Evidence:\*\* `UPLOAD\_DIR = '/tmp/medsecure\_uploads'`  
\* \*\*Impact:\*\* Sensitive patient documents uploaded to the server could be intercepted or tampered with by other local users on the system.  
\* \*\*Compliance:\*\* CWE-377 (Insecure Temporary File)

\*\*3. Use of Outdated and Insecure Cryptographic Library\*\*  
\* \*\*Severity:\*\* MEDIUM  
\* \*\*Location:\*\* `requirements.txt:6`  
\* \*\*Description:\*\* The project uses `pycrypto==2.6.1`, an old, unmaintained library with known vulnerabilities. It should be replaced with a secure, maintained alternative like `pycryptodome`.  
\* \*\*Evidence:\*\* `pycrypto==2.6.1`  
\* \*\*Impact:\*\* Using this library may introduce cryptographic weaknesses into the application, potentially failing to protect data at rest or in transit as required by compliance mandates like HIPAA.  
\* \*\*Compliance:\*\* OWASP Top 10 (A06:2021 – Vulnerable and Outdated Components)

## Security Testing Results The assessment utilized a combination of Static Application Security Testing (SAST) and Software Composition Analysis (SCA) techniques. SAST tools, including Bandit and custom regex-based pattern matching, were used to analyze the Python source code for security flaws. SCA was performed to identify dependencies with known vulnerabilities in `requirements.txt` and `package.json`.

The overall results are summarized below:

| Severity | Count |  
| :--- | :--- |  
| CRITICAL | 1 |  
| HIGH | 4 |  
| MEDIUM | 5 |  
| LOW | 2 |  
| INFO | 6 |  
| \*\*Total\*\* | \*\*18\*\* |

## Technical Recommendations Based on the findings, the following actions are recommended, prioritized by severity:

1. \*\*Remediate Critical Credential Exposure:\*\* Immediately remove the hardcoded JWT secret from the source code and implement a secure secrets management solution. Rotate the exposed secret immediately.  
2. \*\*Disable Debug Mode:\*\* Ensure debug mode is disabled in all production and production-like environments. Use environment variables to control this setting.  
3. \*\*Strengthen Authentication Mechanisms:\*\* Replace the MD5 password hashing algorithm with a modern, strong, and salted hashing algorithm such as Argon2 or bcrypt.  
4. \*\*Mitigate Injection Vulnerabilities:\*\*  
 \* Refactor all database queries to use parameterized statements or an Object-Relational Mapper (ORM) to prevent SQL injection.  
 \* Ensure all user-supplied data rendered in HTML templates is properly context-aware escaped to prevent XSS.  
5. \*\*Update Vulnerable Dependencies:\*\* Replace the `pycrypto` library with `pycryptodome`. Regularly run SCA scans (e.g., `npm audit`, `pip-audit`) to identify and update vulnerable dependencies.  
6. \*\*Secure File Handling:\*\* Configure file uploads to use a private, non-executable directory with strict permissions, and use secure methods for creating temporary files.

## Implementation Guidelines \* \*\*Secrets Management:\*\* Store the `JWT\_SECRET` in an environment variable. In production, use a dedicated secrets management tool like AWS Secrets Manager, Azure Key Vault, or HashiCorp Vault. Load the variable in the application using `os.environ.get('JWT\_SECRET')`. \* \*\*SQL Injection Prevention:\*\* Modify the `get\_patient\_query` function to use parameterized queries with the `mysql-connector-python` library. \* \*\*Insecure:\*\* `f"SELECT \* FROM patients WHERE mrn = '{mrn}'"` \* \*\*Secure:\*\* Create a cursor and execute with a tuple of parameters: `cursor.execute("SELECT \* FROM patients WHERE mrn = %s", (mrn,))` \* \*\*Password Hashing:\*\* Use a library like `passlib` to simplify the implementation of strong password hashing. \* \*\*Example:\*\* `from passlib.context import CryptContext; pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto"); hashed\_password = pwd\_context.hash(password)` \* \*\*Dependency Update:\*\* 1. `pip uninstall pycrypto` 2. `pip install pycryptodome` 3. Update `requirements.txt` accordingly.

## Security Monitoring and Alerting To enhance the security posture and detect potential threats, the following monitoring and alerting mechanisms are recommended: \* \*\*Centralized Logging:\*\* Implement centralized logging for all application and system events. Ensure logs capture security-relevant information, such as authentication successes and failures, access to sensitive data endpoints, and administrative actions. \* \*\*Intrusion Detection/Prevention System (IDS/IPS):\*\* Deploy an IDS/IPS to monitor network traffic for signatures of common attacks, such as SQL injection and XSS attempts. \* \*\*CI/CD Pipeline Security:\*\* Integrate SAST and SCA scanning tools directly into the CI/CD pipeline. Configure the pipeline to fail the build if new vulnerabilities exceeding a defined severity threshold (e.g., High or Critical) are detected. \* \*\*Alerting:\*\* Configure alerts for high-risk security events, including: \* Multiple failed login attempts for a single user account or from a single IP address. \* Any exceptions or errors originating from security components (e.g., authentication, authorization). \* Attempts to access sensitive API endpoints without proper authentication. \* \*\*File Integrity Monitoring (FIM):\*\* Implement FIM on production servers to monitor for unauthorized changes to application code and critical configuration files.