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 application. The scan identified a total of 18 vulnerabilities, including \*\*1 Critical\*\*, \*\*4 High\*\*, and \*\*5 Medium-risk\*\* issues.

The most critical finding is the presence of hardcoded credentials within the source code, which poses a significant risk of unauthorized access. High-risk vulnerabilities include the potential for remote code execution due to an enabled debug mode, Cross-Site Scripting (XSS), and the use of a cryptographically weak hashing algorithm for storing passwords. Additionally, a medium-risk SQL injection vulnerability was discovered.

Given the application's context within the healthcare sector, these vulnerabilities could lead to the compromise of sensitive Protected Health Information (PHI), posing severe compliance risks under HIPAA and threatening patient privacy. The findings indicate systemic weaknesses in secure coding practices, dependency management, and production environment hardening. Immediate remediation of the critical and high-risk issues is strongly recommended to mitigate the risk of a security breach.

## Vulnerability Analysis

### Code Security Issues The static analysis of the application's source code revealed several significant security flaws. These issues stem from insecure coding patterns that expose the application to common but severe attack vectors: \* \*\*Credential Management:\*\* A critical issue was identified where a JWT secret key is hardcoded directly into the authentication module. This practice completely bypasses secure storage mechanisms and exposes the secret to anyone with access to the codebase. \* \*\*Injection Flaws:\*\* The codebase is susceptible to both SQL Injection and Cross-Site Scripting (XSS). SQL queries are constructed using unsafe string formatting, allowing attackers to manipulate database queries. Similarly, user-controlled data is rendered in templates without proper output escaping, enabling the injection of malicious scripts into users' browsers. \* \*\*Weak Cryptography:\*\* The application utilizes the MD5 algorithm for hashing user passwords. MD5 is considered cryptographically broken and is trivial to reverse using modern techniques, leaving user credentials vulnerable. \* \*\*Insecure Configuration:\*\* The Flask web application is configured to run with `debug=True`. In a production environment, this would expose a powerful interactive debugger, allowing an attacker who can trigger an error to execute arbitrary code on the server.

### Dependency Vulnerabilities The scan of third-party libraries revealed dependencies with known security risks. The most significant finding is the use of `pycrypto==2.6.1`. This library is outdated, unmaintained, and contains several known vulnerabilities. It should be replaced immediately with a secure and actively maintained fork like `pycryptodome`. While the scan flagged JavaScript dependencies (`express` and `lodash`) as low-risk, a more thorough Software Composition Analysis (SCA) is recommended to identify specific CVEs associated with the pinned versions.

### Architecture Security Review The identified vulnerabilities suggest underlying architectural weaknesses that require attention: \* \*\*Lack of Secrets Management:\*\* The hardcoded JWT secret points to a missing or unenforced secrets management architecture. A robust system should rely on a secure vault (e.g., HashiCorp Vault, AWS/GCP/Azure Secrets Manager) or, at a minimum, environment variables injected at runtime, rather than storing secrets in version control. \* \*\*Insufficient Environment Separation:\*\* Running the application with `debug=True` indicates a potential failure in separating development configurations from production-ready deployments. The build and deployment pipeline should enforce production-safe configurations. \* \*\*Insecure Data Handling:\*\* The combined presence of SQL injection, XSS, and weak password hashing demonstrates a lack of consistent, secure data handling principles. An architectural standard for input validation, parameterized database queries, output encoding, and secure password storage is necessary.

## Detailed Technical Findings

### Critical Vulnerabilities | ID | Title | File Path | Line | Snippet | | --- | --- | --- | --- | --- | | HARDCODED-SECRET-1 | Hardcoded Credentials | `app\auth.py` | 6 | `JWT\_SECRET = 'demo\_jwt\_secret'` | | \*\*Description\*\* | A sensitive JWT secret key is hardcoded directly in the source code. This exposes the key to anyone with repository access, allowing them to forge valid authentication tokens and gain unauthorized access to the application. This violates security best practices and compliance requirements (CWE-798, OWASP A07:2021). |

### High-Risk Issues | ID | Title | File Path | Line | Snippet | | --- | --- | --- | --- | --- | | FLASK-DEBUG-RCE-1 | Flask Debug Mode Enabled | `app\app.py` | 16 | `app.run(debug=True)` | | \*\*Description\*\* | The Flask application is configured to run with debug mode enabled. This exposes the Werkzeug interactive debugger, which allows for the execution of arbitrary Python code on the server if an attacker can trigger an application error. This effectively provides a remote shell on the server. | | XSS-UNESC-OUTPUT-1 | Cross-Site Scripting (XSS) | `app\patient\_routes.py` | 12 | `return render\_template\_string(template)` | | \*\*Description\*\* | User-controlled input is rendered directly into a template without proper HTML escaping. This allows an attacker to inject malicious scripts that execute in the context of other users' browsers, leading to session hijacking, data theft, or website defacement (CWE-79, OWASP A03:2021). | | HASHLIB-WEAK-HASH-1 | Use of Weak Cryptographic Hash | `app\auth.py` | 9 | `hashlib.md5('password123'.encode()).hexdigest()` | | \*\*Description\*\* | The application uses the MD5 algorithm to hash passwords. MD5 is a cryptographically broken hashing function susceptible to collision attacks and can be easily reversed using pre-computed rainbow tables, exposing user passwords. |

### Medium-Risk Issues | ID | Title | File Path | Line | Snippet | | --- | --- | --- | --- | --- | | SQL-INJECTION-1 | SQL Injection | `app\db.py` | 5 | `return f"SELECT \* FROM patients WHERE mrn = '{mrn}'"` | | \*\*Description\*\* | A database query is constructed by directly embedding user input into the SQL string. This allows an attacker to manipulate the query logic to bypass authentication, exfiltrate, modify, or delete data from the database (Bandit Test ID: B608). | | INSECURE-TEMPFILE-1 | Insecure Temporary File Storage | `app\file\_upload.py` | 6 | `UPLOAD\_DIR = '/tmp/medsecure\_uploads'` | | \*\*Description\*\* | The application uses a predictable, world-writable directory (`/tmp/`) for storing file uploads. This can lead to race conditions where an attacker could potentially access, modify, or overwrite files uploaded by other users before they are processed. | | DEPENDENCY-RISKY-1 | Use of Insecure Dependency `pycrypto` | `requirements.txt` | 6 | `pycrypto==2.6.1` | | \*\*Description\*\* | The project depends on `pycrypto`, an abandoned library with known security vulnerabilities. Continuing to use this library exposes the application to exploits targeting its flaws. |

## Security Testing Results The assessment was conducted using automated static analysis tools, including the Bandit security linter for Python and custom regex-based pattern matching. This approach provided broad coverage of common security vulnerabilities within the application's source code and dependency manifests.

\*\*Vulnerability Severity Breakdown:\*\*

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

## Technical Recommendations To improve the security posture of the MedSecure application, the following actions are recommended:

1. \*\*Implement Secure Secrets Management:\*\* Immediately remove all hardcoded secrets from the source code. Adopt a centralized secrets management solution such as HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault. For environments where this is not feasible, use environment variables to supply secrets to the application at runtime.  
2. \*\*Strengthen Authentication and Cryptography:\*\* Migrate from the MD5 hashing algorithm to a modern, adaptive, and salted password hashing function like Argon2, scrypt, or PBKDF2. Ensure a strong work factor (cost/rounds) is configured.  
3. \*\*Remediate Injection Vulnerabilities:\*\* Refactor all database queries to use parameterized statements or a trusted Object-Relational Mapper (ORM) to prevent SQL injection. Ensure all user-supplied data rendered in HTML templates is contextually escaped by default to prevent XSS.  
4. \*\*Harden Production Environments:\*\* Ensure that debug mode is disabled in all pre-production and production environments. Use environment-specific configurations to manage application settings securely.  
5. \*\*Manage Third-Party Dependencies:\*\* Immediately replace the `pycrypto` library with its secure, actively maintained successor, `pycryptodome`. Implement a continuous dependency scanning process within the CI/CD pipeline to proactively identify and manage vulnerable libraries.

## Implementation Guidelines \* \*\*For Hardcoded Secret (CRITICAL):\*\* 1. Remove the line `JWT\_SECRET = 'demo\_jwt\_secret'` from `app/auth.py`. 2. Load the secret from an environment variable. Example: `JWT\_SECRET = os.environ.get('JWT\_SECRET\_KEY')`. 3. Provision the `JWT\_SECRET\_KEY` environment variable in the deployment environment using a secure mechanism. 4. Rotate the compromised secret immediately.

\* \*\*For Flask Debug Mode (HIGH):\*\*  
 1. Modify `app.run(debug=True)` in `app/app.py`.  
 2. Control the debug flag with an environment variable. Example: `DEBUG\_MODE = os.environ.get('FLASK\_DEBUG', 'False').lower() in ('true', '1')`.  
 3. Ensure the `FLASK\_DEBUG` variable is never set to `true` in production.

\* \*\*For SQL Injection (MEDIUM):\*\*  
 1. Refactor the function in `app/db.py`.  
 2. Use parameterized queries. The `mysql-connector-python` library supports this.  
 3. \*\*Example Fix:\*\*  
 ```python  
 # Instead of: return f"SELECT \* FROM patients WHERE mrn = '{mrn}'"  
 # Use:  
 query = "SELECT \* FROM patients WHERE mrn = %s"  
 cursor.execute(query, (mrn,))  
 ```

\* \*\*For Weak Password Hash (HIGH):\*\*  
 1. Choose a modern hashing library like `passlib`.  
 2. Replace `hashlib.md5()` with a strong algorithm.  
 3. \*\*Example Fix (using passlib):\*\*  
 ```python  
 from passlib.context import CryptContext  
 pwd\_context = CryptContext(schemes=["argon2"], deprecated="auto")  
 hashed\_password = pwd\_context.hash("some\_password")  
 is\_valid = pwd\_context.verify("some\_password", hashed\_password)  
 ```  
 4. Develop a plan to migrate existing password hashes for users.

\* \*\*For Insecure Dependency `pycrypto` (MEDIUM):\*\*  
 1. Run `pip uninstall pycrypto`.  
 2. Run `pip install pycryptodome`.  
 3. Update `requirements.txt` to replace `pycrypto==2.6.1` with the new version of `pycryptodome`. `pycryptodome` is a drop-in replacement, so code changes are typically not required.

## Security Monitoring and Alerting To maintain security posture and detect potential threats, the following monitoring and alerting mechanisms are recommended:

\* \*\*Configuration Monitoring:\*\* Implement checks in the CI/CD pipeline and/or infrastructure monitoring to alert if the application is ever deployed to a production environment with debug mode enabled.  
\* \*\*Application Security Logging:\*\* Enhance application logging to record and alert on security-sensitive events, such as failed login attempts, potential injection patterns detected in input fields (via a Web Application Firewall or runtime protection), and access control failures.  
\* \*\*Secret Scanning:\*\* Integrate automated secret scanning tools (e.g., Git-secrets, TruffleHog) into the CI/CD pipeline to prevent new hardcoded secrets from being committed to the repository.  
\* \*\*Dependency Vulnerability Alerting:\*\* Utilize a tool like GitHub's Dependabot or Snyk to continuously monitor third-party dependencies and generate alerts when new vulnerabilities affecting the project are discovered.