Security Technical Operational Report

# Technical Security Assessment Report

## Technical Summary A technical security assessment was conducted on the `hospital\_mock` application repository. The scan, completed on 2025-09-25, identified 15 total findings, including \*\*1 High\*\*, \*\*5 Medium\*\*, and \*\*3 Low-risk\*\* vulnerabilities. The most critical issue is the presence of a hardcoded JWT secret within the application's authentication module, which could allow an attacker with source code access to compromise user sessions and impersonate users.

Other significant findings include the use of a cryptographically weak hashing algorithm (MD5), a potential Cross-Site Scripting (XSS) vulnerability due to unsafe template rendering, and the inclusion of a potentially risky and unmaintained cryptographic dependency (`pycrypto`). Given that this application operates in the Healthcare sector and handles Protected Health Information (PHI), as evidenced by the internal privacy policy, these vulnerabilities present a heightened risk and require prompt remediation to protect sensitive patient data and ensure regulatory compliance.

## Vulnerability Analysis

### Code Security Issues The static analysis of the source code revealed several significant security flaws. A high-risk hardcoded JWT secret was discovered in `app/auth.py`, which fundamentally undermines the security of the authentication mechanism. The same file also contains the use of the outdated and insecure MD5 hashing algorithm. Furthermore, the application's patient data routes (`app/patient\_routes.py`) utilize the `render\_template\_string` function from Flask, which creates a significant risk of Cross-Site Scripting (XSS) if any part of the template is influenced by user input.

### Dependency Vulnerabilities The project's dependencies present a medium-risk exposure. The Python dependency list (`requirements.txt`) includes `pycrypto==2.6.1`, an old and unmaintained library with known vulnerabilities and no active support. Developers should migrate to a modern, maintained fork like `pycryptodome`. While the Python dependencies are pinned to specific versions, which is a good security practice for ensuring build reproducibility, the JavaScript dependencies in `package.json` use caret versioning (e.g., `^4.18.2`), which can introduce unintended or vulnerable package versions during installation.

### Architecture Security Review The identified vulnerabilities suggest architectural weaknesses in the application's design. There is a clear lack of a centralized secrets management strategy, forcing developers to hardcode sensitive values directly into the source code. The choice of cryptographic primitives (MD5) indicates a need for updated security standards and guidelines for developers. The pattern of rendering templates from strings in `patient\_routes.py` is inherently insecure and suggests that the architecture may not be enforcing a "secure by default" approach, particularly concerning input handling and output encoding when processing sensitive PHI.

## Detailed Technical Findings

### Critical Vulnerabilities No critical vulnerabilities were identified during this assessment.

### High-Risk Issues \*\*1. Hardcoded Credentials (JWT Secret)\*\* \* \*\*ID:\*\* `HARDCODED-SECRET` \* \*\*File:\*\* `app/auth.py` \* \*\*Line:\*\* 6 \* \*\*Severity:\*\* HIGH \* \*\*Snippet:\*\* `JWT\_SECRET = 'demo\_jwt\_secret'` \* \*\*Description:\*\* The secret key used to sign and verify JSON Web Tokens (JWTs) is hardcoded directly in the source code. This exposes the secret to anyone with access to the codebase, including developers, CI/CD systems, and potentially unauthorized individuals if the repository is compromised. \* \*\*Impact:\*\* An attacker with this secret can forge valid JWTs for any user, bypassing authentication, impersonating users, and gaining unauthorized access to sensitive patient data and application functionality.

### Medium-Risk Issues \*\*1. Use of Weak Cryptographic Hash (MD5)\*\* \* \*\*ID:\*\* `WEAK-CRYPTO-MD5` \* \*\*File:\*\* `app/auth.py` \* \*\*Severity:\*\* MEDIUM \* \*\*Description:\*\* The application utilizes the MD5 hashing algorithm. MD5 is considered cryptographically broken and is susceptible to collision attacks, making it unsuitable for any security-sensitive purpose like password hashing or data integrity checks. \* \*\*Impact:\*\* If used for password storage, it could allow an attacker to more easily crack user passwords via rainbow table attacks. If used for integrity checks, an attacker could create a malicious file with the same MD5 hash as a legitimate one.

\*\*2. Potential Cross-Site Scripting (XSS) via Unsafe Template Rendering\*\*  
\* \*\*ID:\*\* `POTENTIAL-XSS-TEMPLATE`  
\* \*\*File:\*\* `app/patient\_routes.py`  
\* \*\*Lines:\*\* 4, 12  
\* \*\*Snippet:\*\* `from flask import Blueprint, request, jsonify, render\_template\_string`, `return render\_template\_string(template)`  
\* \*\*Description:\*\* The application uses Flask's `render\_template\_string` function, which renders a template from a string variable. If this string can be influenced by user-controlled input, an attacker could inject malicious HTML and JavaScript payloads.  
\* \*\*Impact:\*\* A successful XSS attack could lead to session hijacking, theft of sensitive data (including PHI displayed on the page), and unauthorized actions performed on behalf of the user.

\*\*3. Use of Potentially Risky and Unmaintained Dependency\*\*  
\* \*\*ID:\*\* `DEPENDENCY-POTENTIALLY-RISKY`  
\* \*\*File:\*\* `requirements.txt`  
\* \*\*Line:\*\* 6  
\* \*\*Snippet:\*\* `pycrypto==2.6.1`  
\* \*\*Description:\*\* The project depends on `pycrypto`, which is an abandoned library. It is no longer maintained and may contain unpatched vulnerabilities. The recommended replacement is `pycryptodome`.  
\* \*\*Impact:\*\* Using unmaintained libraries exposes the application to known and future vulnerabilities that will not be fixed by the library authors, increasing the long-term risk of compromise.

## Security Testing Results The automated security scan produced the following breakdown of findings by severity:

\* \*\*Total Findings:\*\* 15  
\* \*\*Severity Breakdown:\*\*  
 \* \*\*High:\*\* 1  
 \* \*\*Medium:\*\* 5  
 \* \*\*Low:\*\* 3  
 \* \*\*Informational:\*\* 6

## Technical Recommendations It is strongly recommended that the development team prioritize and address the identified vulnerabilities in the following order:

1. \*\*Externalize Hardcoded Secrets:\*\* Immediately remove the hardcoded `JWT\_SECRET` from the source code. Store it in a secure secrets management system (e.g., AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault) and have the application fetch it at runtime.  
2. \*\*Replace Weak Cryptography:\*\* Migrate from the MD5 algorithm to a strong, modern hashing standard. For password hashing, use a memory-hard algorithm like \*\*Argon2\*\* or \*\*scrypt\*\*.  
3. \*\*Refactor Unsafe Template Rendering:\*\* Avoid using `render\_template\_string` with any data that could be influenced by external input. Refactor the code to use Flask's standard `render\_template` function with static `.html` template files, which benefit from Jinja2's default auto-escaping context to prevent XSS.  
4. \*\*Upgrade Insecure Dependencies:\*\* Replace the unmaintained `pycrypto` library with its actively maintained drop-in replacement, `pycryptodome`.  
5. \*\*Implement Dependency Management:\*\* Integrate an automated dependency scanning tool (e.g., Dependabot, Snyk) into the CI/CD pipeline to continuously monitor for new vulnerabilities in both Python and JavaScript dependencies.

## Implementation Guidelines To facilitate remediation, follow these technical guidelines:

1. \*\*For the Hardcoded JWT Secret:\*\*  
 \* Provision a new secret value in your chosen secrets manager.  
 \* Grant the application's runtime environment (e.g., IAM role, Kubernetes service account) read-only access to this specific secret.  
 \* In `app/auth.py`, replace `JWT\_SECRET = 'demo\_jwt\_secret'` with code that uses the appropriate SDK (e.g., `boto3` for AWS) to fetch the secret on application startup.  
 \* Remove the hardcoded secret from the file and use a tool to purge it from the Git history to prevent historical exposure.

2. \*\*For Weak Crypto (MD5):\*\*  
 \* If hashing passwords, add a library like `bcrypt` or `argon2-cffi` to `requirements.txt`.  
 \* Replace the code using `hashlib.md5()` with functions provided by the new library (e.g., `bcrypt.hashpw()` and `bcrypt.checkpw()`).  
 \* Plan a migration strategy for existing user passwords, rehashing them with the new algorithm upon their next successful login.

3. \*\*For Potential XSS:\*\*  
 \* Identify the logic that constructs the template string passed to `render\_template\_string` in `app/patient\_routes.py`.  
 \* Create a new, static template file in the `templates/` directory.  
 \* Modify the route to call `render\_template('new\_template.html', var=data)`, passing dynamic data as context variables rather than building the template string itself. Jinja2 will automatically escape this data.

4. \*\*For the `pycrypto` Dependency:\*\*  
 \* Run the following commands in the project's virtual environment:  
 ```bash  
 pip uninstall pycrypto  
 pip install pycryptodome  
 ```  
 \* Update `requirements.txt` to replace `pycrypto==2.6.1` with the new `pycryptodome` version. `pycryptodome` is a drop-in replacement, so no code changes should be necessary in most cases.

## Security Monitoring and Alerting To enhance detection and response capabilities, the following monitoring controls should be implemented:

1. \*\*Authentication Monitoring:\*\* Log and create high-priority alerts for all JWT validation failures. A sudden spike in failures could indicate an attempt to forge or tamper with tokens.  
2. \*\*WAF Implementation:\*\* Deploy a Web Application Firewall (WAF) in front of the application. Configure it with rulesets to detect and block common web attacks, including Cross-Site Scripting (XSS) and other injection attacks.  
3. \*\*Dependency Vulnerability Alerting:\*\* Configure automated alerts within the CI/CD pipeline (e.g., GitHub Dependabot alerts) to notify the security and development teams as soon as a new vulnerability is discovered in a project dependency.  
4. \*\*Incident Response Trigger:\*\* Ensure that alerts generated from these monitoring systems are integrated with the incident response workflow outlined in the `policies/breach\_response\_policy.md`, triggering the appropriate escalation and notification procedures.