Security Technical Operational Report

# Technical Security Assessment Report

## Technical Summary This report details the findings of a technical security assessment conducted on the `hospital\_mock` application repository on September 25, 2025. The scan identified a total of 15 vulnerabilities and configuration issues, with a breakdown of 1 High, 5 Medium, 3 Low, and 6 Informational findings.

The most critical issue identified is a \*\*High-risk hardcoded JWT secret\*\* within the application's authentication module. This vulnerability poses a significant risk of unauthorized access and session hijacking. Additionally, several Medium-risk issues were discovered, including the use of the outdated and cryptographically weak MD5 hashing algorithm, a potential for Cross-Site Scripting (XSS) through insecure template rendering, and the inclusion of a potentially risky and unmaintained cryptography dependency (`pycrypto`).

Given that the application handles Protected Health Information (PHI), as indicated by the repository's privacy policy, these vulnerabilities present a heightened risk to data confidentiality and integrity. The overall security posture of the application is considered weak, and immediate remediation of the identified high and medium-risk findings is strongly recommended to mitigate the risk of a security breach.

## Vulnerability Analysis

### Code Security Issues The static analysis of the application source code revealed several significant security flaws. A high-severity hardcoded JWT secret was found directly in the authentication code (`app/auth.py`), which completely undermines the security of user sessions. The same file also contains the use of the MD5 algorithm, a weak cryptographic hash function unsuitable for any security-sensitive purpose. Furthermore, the patient data handling module (`app/patient\_routes.py`) uses a rendering function (`render\_template\_string`) that is highly susceptible to Cross-Site Scripting (XSS) if user-provided data is not properly sanitized, creating a vector for attackers to inject malicious scripts.

### Dependency Vulnerabilities The application relies on several third-party libraries, introducing dependency-related risks. The Python dependency list (`requirements.txt`) includes `pycrypto==2.6.1`, a library that is no longer maintained and is known to have security weaknesses. While Python dependencies are pinned to specific versions, which is a good security practice, the use of this outdated library negates some of that benefit. The `package.json` file indicates that JavaScript dependencies such as `express` and `lodash` are not pinned to exact versions, which could lead to unpredictable builds and the automatic inclusion of newly discovered vulnerabilities in minor version updates.

### Architecture Security Review From an architectural perspective, the findings point to a lack of foundational security controls. There is no evidence of a secure secrets management strategy, with sensitive credentials being stored directly in source code. The application's design for rendering dynamic content containing sensitive patient information is inherently insecure, failing to leverage standard, auto-escaping templating engines. The combination of these issues within an application designed for the healthcare sector suggests that security was not a primary consideration during the design and development phases, leading to a system that is ill-equipped to protect the sensitive PHI it processes.

## Detailed Technical Findings

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

### High-Risk Issues | Finding ID | Vulnerability | Location | Line | Snippet | | :--- | :--- | :--- | :--- | :--- | | HS-001 | Hardcoded Credentials | `app/auth.py` | 6 | `JWT\_SECRET = 'demo\_jwt\_secret'` | | \*\*Description:\*\* A hardcoded JWT secret key is present in the source code. This key is used to sign and verify authentication tokens. An attacker with read access to the source code can obtain this secret, allowing them to forge valid authentication tokens for any user, bypass access controls, and gain unauthorized access to the application and its data. | | \*\*Recommendation:\*\* The hardcoded secret must be removed from the source code immediately. It should be stored and managed securely using a secrets management solution such as HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault. For development environments, environment variables can be used as a temporary, more secure alternative. |

### Medium-Risk Issues | Finding ID | Vulnerability | Location | Line | Snippet | | :--- | :--- | :--- | :--- | :--- | | MS-001 | Use of Weak Cryptographic Algorithm | `app/auth.py` | 0 | `Use of MD5 detected` | | \*\*Description:\*\* The application utilizes the MD5 hashing algorithm. MD5 is considered cryptographically broken and is susceptible to collision attacks, making it unsuitable for security purposes such as password hashing or data integrity checks. | | \*\*Recommendation:\*\* Replace MD5 with a modern, strong hashing algorithm. For password storage, use a password-based key derivation function like Argon2, scrypt, or at a minimum, bcrypt. | | MS-002 | Potential Cross-Site Scripting (XSS) | `app/patient\_routes.py` | 12 | `return render\_template\_string(template)` | | \*\*Description:\*\* The application uses `render\_template\_string` to generate HTML content. If any part of the `template` variable is derived from user-controlled input without proper sanitization, an attacker could inject malicious JavaScript code, leading to XSS attacks. This could be used to steal session cookies, PHI, or perform actions on behalf of the user. | | \*\*Recommendation:\*\* Avoid using `render\_template\_string`. Refactor the code to use Flask's standard `render\_template` function with Jinja2 templates, which provide context-aware auto-escaping by default to mitigate XSS vulnerabilities. | | MS-003 | Use of Potentially Risky Dependency | `requirements.txt` | 6 | `pycrypto==2.6.1` | | \*\*Description:\*\* The application depends on `pycrypto`, an abandoned and outdated cryptography library. This library is no longer maintained and may contain unpatched vulnerabilities. Its use introduces unnecessary risk to the application. | | \*\*Recommendation:\*\* Migrate from `pycrypto` to a well-maintained fork such as `pycryptodome`. Update the `requirements.txt` file and any associated application code to use the new library. |

## Security Testing Results

The following table summarizes the high and medium-risk vulnerabilities identified during the security scan:

| ID | Title | Severity | File Path | Finding Summary |  
|---|---|---|---|---|  
| HS-001 | Hardcoded Credentials | \*\*HIGH\*\* | `app/auth.py` | A static JWT secret key is embedded directly in the source code. |  
| MS-001 | Weak Cryptography | \*\*MEDIUM\*\* | `app/auth.py` | The insecure MD5 hashing algorithm is used within the application. |  
| MS-002 | Potential Cross-Site Scripting | \*\*MEDIUM\*\* | `app/patient\_routes.py`| Use of an insecure template rendering function allows for potential XSS. |  
| MS-003 | Risky Dependency | \*\*MEDIUM\*\* | `requirements.txt` | The application uses the outdated and unmaintained `pycrypto` library. |

## Technical Recommendations

1. \*\*Implement a Secure Secrets Management Strategy:\*\* Immediately prioritize the removal of all hardcoded credentials from the source code. Adopt a centralized secrets management tool (e.g., HashiCorp Vault, AWS Secrets Manager) for production environments. Use environment variables loaded from a secure source for local development and CI/CD pipelines.

2. \*\*Strengthen Cryptographic Standards:\*\* Deprecate and replace all instances of weak cryptographic algorithms like MD5. Adhere to industry best practices by using strong, modern algorithms such as Argon2 for password hashing and SHA-256 for data integrity checks.

3. \*\*Enforce Secure Coding for Web Components:\*\* Mandate the use of secure frameworks and practices to prevent injection flaws. Specifically, eliminate the use of `render\_template\_string` and enforce the use of standard templating engines like Jinja2 that provide automatic output encoding to prevent XSS. All user-supplied input must be treated as untrusted and be properly validated and sanitized.

4. \*\*Establish a Robust Dependency Management Program:\*\* Create and enforce a policy for managing third-party dependencies. This should include replacing unmaintained libraries like `pycrypto` with actively maintained alternatives (`pycryptodome`). Implement automated dependency scanning tools (e.g., Snyk, Dependabot, `pip-audit`) into the CI/CD pipeline to proactively identify and remediate vulnerable packages. Ensure all dependencies are pinned to specific, vetted versions.

## Implementation Guidelines

\* \*\*To address HS-001 (Hardcoded Secret):\*\*  
 1. Replace `JWT\_SECRET = 'demo\_jwt\_secret'` in `app/auth.py` with code to read the secret from an environment variable: `JWT\_SECRET = os.environ.get('JWT\_SECRET')`.  
 2. Provision the `JWT\_SECRET` value in the application's runtime environment using a secrets management service integrated with your deployment platform.  
 3. Rotate the compromised secret immediately, as it has been exposed in the repository's history.

\* \*\*To address MS-001 (Weak Crypto):\*\*  
 1. Identify where MD5 is being used. If for password hashing, introduce a modern library like `passlib`.  
 2. Implement a hashing function using `bcrypt` or `Argon2`.  
 3. Create a data migration path for existing users to have their passwords re-hashed with the new algorithm upon their next login.

\* \*\*To address MS-002 (Potential XSS):\*\*  
 1. Refactor the route in `app/patient\_routes.py` to use `render\_template`.  
 2. Create a dedicated Jinja2 template file (e.g., `templates/patient\_view.html`).  
 3. Change the return statement from `return render\_template\_string(template)` to `return render\_template('patient\_view.html', patient\_data=patient\_data)`. Jinja2 will automatically escape dynamic data.

\* \*\*To address MS-003 (Risky Dependency):\*\*  
 1. In `requirements.txt`, replace `pycrypto==2.6.1` with `pycryptodome>=3.15.0` (or the latest stable version).  
 2. Run `pip uninstall pycrypto` and `pip install -r requirements.txt`.  
 3. Search the codebase for any `import Crypto` statements and test thoroughly, as `pycryptodome` is a drop-in replacement but may require minor adjustments.

## Security Monitoring and Alerting To enhance the application's security posture and ensure ongoing visibility, the following monitoring and alerting mechanisms are recommended:

1. \*\*Audit and Security Event Logging:\*\* Implement structured logging for all security-sensitive events, including successful/failed login attempts, password changes, and access to sensitive patient data. Logs should be forwarded to a centralized SIEM (Security Information and Event Management) system.

2. \*\*Secret Scanning:\*\* Integrate an automated secret scanning tool into the CI/CD pipeline. Configure it to fail any build that attempts to commit hardcoded secrets, API keys, or other credentials to the repository.

3. \*\*WAF and Intrusion Detection:\*\* Deploy a Web Application Firewall (WAF) to monitor and block common web-based attacks, such as XSS and SQL injection. Configure rules to specifically look for attack patterns targeting the application's known vulnerabilities until they are patched.

4. \*\*Dependency Vulnerability Alerting:\*\* Configure automated tools like Dependabot or Snyk to continuously monitor the application's dependencies. Set up high-priority alerts to be sent to the security and development teams whenever a new vulnerability is discovered in a library currently in use, enabling rapid response and patching. Alerting procedures should align with the timelines defined in the `breach\_response\_policy.md`.