Securing Flask Microservices: Vulnerability Analysis & Hardening Techniques

Securing Containerized Microservices

Resilient Microservices: Analyzing & Fortifying Flask Security
Anetta Nichols
31 May 2025

Table of Contents

I. Initial Assessment & Risk Identification **3**

Table 1: *Environment Setup & Security Risks* 3

II. Application and Container Security Enhancements **3**

Table 2: *Application Security Enhancements* 3

Table 3: *Container Security Enhancements* 3

III. Threat Modeling **4**

Table 4*: STRIDE Threat Categories & Defense Strategies* 4

Table 5: *MITRE ATT&CK Mapping* 4

IV. Post-Remediation Security Scan Results **4**

Table 6: *Post-Remediation Security Scans* 4

V. Security Architecture Implementation **4**

VI. References **5**

Security Hardening: Containerized Microservices Protection

Abstract

This report secures a vulnerable Flask microservices app by identifying risks, implementing fixes, and applying STRIDE and MITRE ATT&CK frameworks for resilience.

I. Initial Assessment & Risk Identification

The vulnerable Flask application contained hardcoded credentials, command injection risks, and unrestricted network exposure. Due to WSL and Docker Desktop instability on Windows 11, migration to Ubuntu 24.04.1 ensured stable execution. Security scans (make check, make scan, make host-security) confirmed additional risks, highlighting the need for remediation.

Table 1: Environment Setup & Security Risks: Key vulnerabilities and mitigation strategies.

Security Issue	Affected Area	Risk Level	Mitigation
Hardcoded Secrets	Credential Storage	High	Store secrets in .env to prevent plaintext credentials.
Command Injection	API Execution (app.py:27)	Critical	Restrict subprocess.run(["ping", "-c", "1", ip]) to /bin/ping.
Code Execution	Input Parsing (app.py:42)	High	Replace eval() with ast.literal_eval() to block execution risks.
Flask Exposure	Network Binding	Medium	Restrict Flask to 127.0.0.1 instead of 0.0.0.0.

II. Application and Container Security Enhancements

Security improvements eliminated vulnerabilities, strengthened authentication, and enforced execution restrictions. Figure 1 in the GitHub repository, "Container Hardening & Security Architecture Design", visually outlines security strategies for containerized microservices. Credentials migrated to environment variables, risky functions replaced, and input sanitized.

Table 2: Application Security Enhancements: Security fixes for the Flask app.

Vulnerability Type	Component Affected	Threat Severity	Mitigation Action
Credential Exposure	app.py:8	High	Store secrets in .env file to prevent credential theft
Remote Execution	app.py:30	Critical	Replace eval() with ast.literal_eval() prevent code execution
Input Validation	app.py:22	High	Sanitize inputs to block malicious payloads
Network Binding	app.py:34	Medium	Restrict Flask to localhost to block external access.

Table 3: Container Security Enhancements: Hardening measure for containers.

Security Measure	Implementation Step	Protection Level	System Impact
Least Privilege	USER appuser	High	Prevents root execution and privilege escalation risks.
Read-Only Access	read_only: true	High	Blocks unauthorized changes to the filesystem.
Resource Limits	mem_limit=512m, cpu_shares=1024	Medium	Prevents resource exhaustion and denial-of-service attacks.
Port Restriction	127.0.0.1:15000:5000	High	Eliminates external attack vectors by limiting exposure.
Health Checks	HEALTHCHECK CMD curl	High	Enables runtime monitoring for container security.

III. Threat Modeling

STRIDE and MITRE ATT&CK assessments strengthened the Flask microservices deployment by categorizing vulnerabilities, mapping security risks to real-world threats, and identifying industry-standard countermeasures.

Table 4*:* STRIDE Threat Categories & Defense Strategies: Security controls mapped to threats.

Threat Category	Attack Surface	Impact Severity	Control Measures	Threat Impact
Spoofing	/calculate API	High	Token-based authentication	Prevents unauthorizes access.
Tampering	/ping Input	Critical	Input validation	Blocks commands injection risks
Data Exposure	Hardcoded Secrets	High	Store secrets in .env	Prevents unauthorized access.
Denial of Service	Unlimited Requests	Medium	Rate-limiting protection	Attackers flood /calculate, exhausting resources

Table 5: MITRE ATT&CK Mapping**:** Attack techniques and mitigation strategies.

Tactic Type	Technique ID	Attack Method	Defense Plan
Initial Access	T1190	Exploit APIs for unauthorized access	Validate input, sanitize request, and restrict exposure.
Execution	T1059.006	(Python Execution): Exploit eval() for arbitrary code execution	Replace eval() with ast.literal_eval() to block execution risks.
Privilege Escalation	T1611	Escape to Host: Running containers as root allows full system compromise	Enforce non-root execution (USER appuser) and privilege restrictions.
Persistence	T1525	Inject malicious images to maintain persistence	Apply image verification to prevent unauthorized deployments.

IV. Post-Remediation Security Scan Results

Following the security hardening measures, the application was re-evaluated using the predefined scanning tools. The results confirmed a significant reduction in vulnerabilities.

Table 6: Post-Remediation Security Scans: Security scan comparison before and after fixes.

Scan Tool	Before Fixes (Initial Scan Results)	After Fixes (Post-Remediation Results)
make check	Hardcoded secrets detected	No hardcoded secrets remain, now stored in .env files; only two low severity warnings (B404: Subprocess & B603:Validation)
make scan	Command injection possible	Input validation blocks malicious payloads
make host-security	Containers running as root	Enforced least privilege (USER appuser), reduced access risk.

V. Security Architecture Implementation

The Automated Container Security Hardening Script (docker_security_fixes.py) applies execution restrictions, privilege controls, and resource limits across app.py, Dockerfile, Makefile and docker-compose.yml. These files enforce security through privileges, sanitization, and restrictions. This process reinforced the importance of environmental stability, security automation, and layered defenses using STRIDE and MITRE ATT&CK. The script is in the GitHub repository for validation and deployment.

VI. References

The following cybersecurity frameworks, standards, and sources informed this report's security assessments and threat mitigation strategies:

1. Center for Internet Security (CIS). (2025). *CIS Docker Security Benchmark.* Retrieved from https://www.cisecurity.org/benchmark/docker/
2. Docker, Inc. (2025). *Docker Security Best Practices.* Retrieved from https://docs.docker.com/security/
3. MITRE Corporation. (2025). *MITRE ATT&CK for Containers and Enterprise.* Retrieved from https://attack.mitre.org/
4. National Institute of Standards and Technology (NIST). (2025). *Security and Privacy Controls for Information Systems (NIST SP 800-53 Rev. 5).* Retrieved from https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final
5. Open Web Application Security Project (OWASP). (2021). *OWASP Top Ten 2021.* Retrieved from https://owasp.org/www-project-top-ten/
6. Security Magazine. (n.d.). Cybersecurity illustration [Image]. https://www.securitymagazine.com/ext/resources/cyber-security-freepik-3.jpg?1627047472