Design and Evaluation of Security Mechanisms for Hosting Student Web Applications (Master research project, SYSTEMF, Spring 2025)

François Théron

June 30, 2025

Abstract

This project analyzes and implements security mechanisms for a web application hosting platform for student projects, running multiple apps within a single JVM process. Approximately one hundred student-developed Scala applications are executed within this shared JVM. The solution employs static bytecode analysis, primarily using the ASM framework for bytecode inspection to detect unsafe operations (e.g., filesystem access, JNI calls, serialization), and integrates GraalVM native image compilation to restrict reflection and reduce the runtime attack surface. Application loading is shifted to compile-time through automated class hierarchy analysis, eliminating the need for dynamic class loading. The deployment is containerized using Podman in rootless mode, with distroless Chainguard images and strict least-privilege configurations, including read-only filesystems and minimized capabilities. Network isolation is enforced by routing all external traffic through an Nginx reverse proxy. Server hardening includes process minimization, port restriction, Fail2ban for SSH protection, and Linux auditd for critical file and process monitoring. The entire build and deployment process is automated through CI pipelines using Ansible with extensive documentation. Security evaluation with OpenSCAP and Chainguard GPOS STIG profiles shows high compliance, with cryptographic controls delegated to the appropriate system layers. Limitations include lack of strong outbound network controls, absence of seccomp profiles, and reliance on manual GraalVM configuration for reflection metadata. The approach achieves significant attack surface reduction but leaves room for improvement in runtime monitoring and network isolation.

Contents

1 Introduction **3**

2 Threat Model **3**
2.1 Escalating Threat Actor Activity Against Universities 3
2.2 System Core Components 3
2.3 Asset Identification and Valuation 4
2.4 Threat Analysis 4
2.4.1 Malicious Student Actors 4
2.4.2 External Threat Actors 5

3 Design & Implementation **6**
3.1 Application Hardening Through Static Bytecode Analysis 6
3.1.1 Bytecode Analysis using ASM framework 6
3.1.2 Compile-Time App Loading 6
3.2 OS Hardening 7
3.2.1 Containerization 7
3.2.2 Server Hardening 9
3.2.3 CI & Automation 9
3.3 Secret Management 9

4 Evaluation **10**
4.1 Containers Security Assessment 10
4.2 Bytecode Analysis & GraalVM Evaluation 12
4.2.1 Bytecode Analysis with GraalVM 12
4.2.2 Attack Surface Analysis 12

5 Discussion **14**

6 Timeline Expectations vs. Reality **14**

7 References **15**

1 Introduction

This project addresses the challenge of securing a web platform that hosts a variety of student-developed applications. With around a hundred different apps running in a single JVM instance on a web server, the risks of insecure coding practices and single points of failure are significant. Furthermore, since we do not have full insight into the behavior of each application, our only guarantee is that all apps use a controlled library provided for the course.

2 Threat Model

2.1 Escalating Threat Actor Activity Against Universities

Higher education institutions face increasing cyberattack frequency, with 91% experiencing attacks in the past 12 months (in UK) and ransomware attacks surging 69% in Q1 2025 [5, 3]. EPFL represents a high-value target due to its high quantity of personal data and sprawling network.

This threat landscape directly impacts our threat model within EPFL's network, as compromising our student web application server could provide attackers with a strategic foothold to potentially compromise broader institutional systems.

2.2 System Core Components

The analyzed system comprises a web application hosting environment within EPFL's network infrastructure. This architecture enables students to develop, deploy, and execute Scala-based web applications within a shared hosting platform, creating multiple attack surfaces and trust boundaries that require careful security analysis. The system consists of four primary components that interact to provide the educational hosting service:

A Linux-based server operating within EPFL's internal network. This machine serves as the foundation for all application hosting activities and maintains access to university network resources.

An Nginx Reverse Proxy that functions as the primary entry point for external traffic, providing SSL/TLS termination and request routing to the backend application. The Nginx configuration handles both HTTP-to-HTTPS redirects and static file serving, while implementing basic rate limiting and security headers.

A hundred of students developed independent Scala-based web applications executing within the shared JVM environment. Each application operates with access to the Java Standard Library, Scala reflection APIs, and shared system resources including file systems and network interfaces.

Common dependencies including the WebApp library, scala.reflect, static resources, and system utilities are accessible across all student applications.

2.3 Asset Identification and Valuation

The primary asset requiring protection is the host machine operating within EPFL's internal network. This system represents a high-value target due to its network position, potentially serving as a pivot point for lateral movement across hundreds of connected devices, computers, users, data etc.

2.4 Threat Analysis

The threat landscape is characterized by two distinct categories: malicious internal actors with legitimate system access who deliberately exploit their privileges, and external threat actors who systematically target vulnerabilities introduced by inexperienced student developers.

2.4.1 Malicious Student Actors

Malicious students represent a deliberate threat vector with the technical capability to develop and deploy harmful web applications within the hosting environment. These actors are considered technically medium, but do not have a lot of resources.

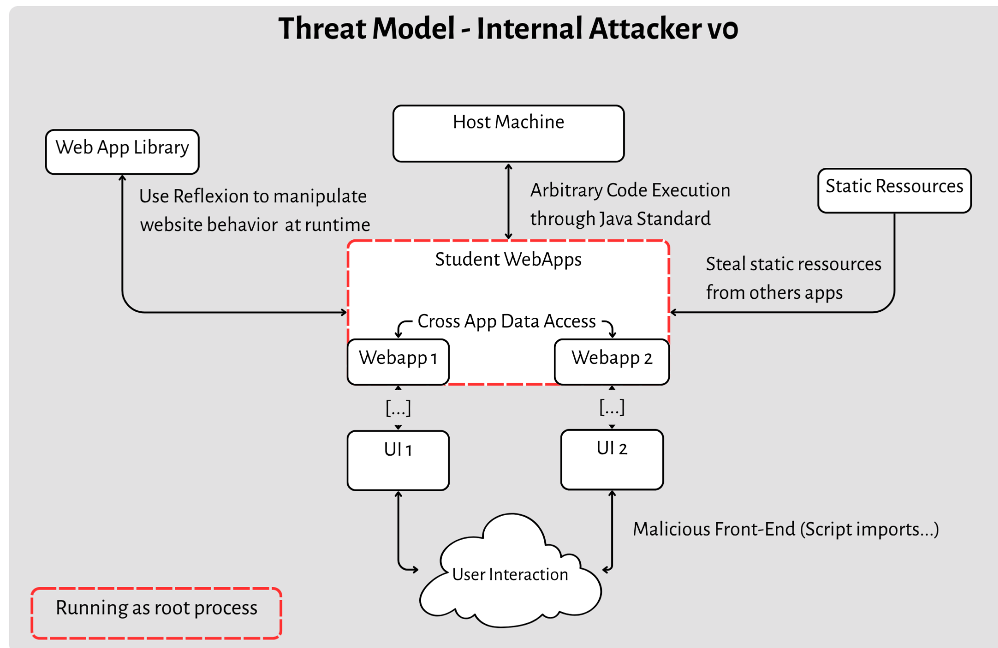

Figure 1: Threat model from a student point of view

As illustrated in Figure 1, malicious internal students possess multiple attack vectors to exploit their privileged access, ranked by threat severity: deploying malicious payloads that execute harmful actions once infiltrated into the system, achieving arbitrary code execution on the host machine through privilege escalation techniques, exploiting Java/Scala reflection mechanisms to create unexpected control flow paths and bypass security checks, establishing

persistent backdoors that enable continued unauthorized access even after detection, and implementing malicious frontend interfaces that manipulate user interactions.

2.4.2 External Threat Actors

The external threat landscape we face includes several types of attackers who will be part of our threat model. Automated attackers represent the most common threat - they use readily available scanning tools and exploit frameworks that make them easy to detect, but they can still exploit obvious vulnerabilities if we leave them exposed. More concerning are financially motivated cybercriminals who bring moderate to advanced skills along with access to commercial exploit kits. These attackers target such infrastructures as stepping stones for larger operations, making them a threat we will defend against.

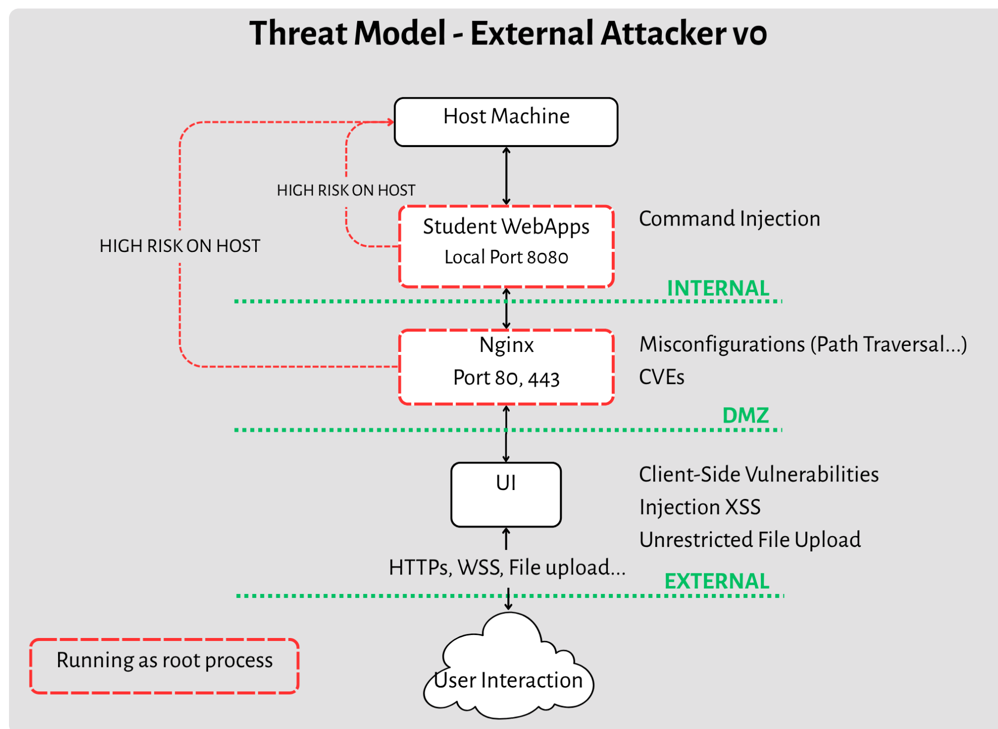

Figure 2: Threat model for an external attacker point of view

As illustrated in Figure 2, external threat actors can exploit various attack vectors through student web applications, including command injection vulnerabilities that allow remote execution of arbitrary operating system commands, client-side vulnerabilities such as XSS, and Nginx misconfigurations or emerging CVEs like CVE-2025-1974 (CVSS 9.8) that enable configuration injection attacks and potential cluster takeover. In this first assessment the most concerning vulnerabilities are the main processes that are running as root.

3 Design & Implementation

3.1 Application Hardening Through Static Bytecode Analysis

To harden our current configuration we used a multi-layered security hardening approach that addresses the fundamental challenges of monitoring and controlling Scala application behavior in hosted environments.

3.1.1 Bytecode Analysis using ASM framework

ASM bytecode analysis was selected to achieve maximum proximity to JVM execution without code execution. This approach enables direct inspection of bytecode instructions, avoiding the security gaps inherent in higher-level analysis methods that may overlook critical operations.

The ASM visitor pattern implementation systematically traverses all bytecode instructions to identify: direct unsafe operations (filesystem access, network operations, system property queries), JNI method invocations (native calls that bypass Java security controls and enable direct system manipulation), serialization mechanisms (representing one of the most significant attack vectors in Java applications through deserialization exploits), and resource access patterns (file and network resource loading operations with potential for data exfiltration or unauthorized access). In the other hand, traditional bytecode analysis doesn't check for reflective calls, since it doesn't have information about dynamic analysis. Reflective calls then become a bypass to this mitigation. That's why we will use GraalVM compilation to whitelist all reflexive calls and have an almost full knowledge over the runtime behavior.

3.1.2 Compile-Time App Loading

The monorepo architecture containing multiple independent web applications necessitated reflexive app loading, as the wrapper could not statically determine each webapp's class names and instantiation requirements. To eliminate these reflection calls while maintaining functionality, we implemented a bytecode analysis solution.

The system systematically analyzes all classes to determine their inheritance hierarchy and instantiability, enabling reliable compile-time class instantiation generation. Our algorithm achieved 100% accuracy in identifying target classes for instantiation with no false positives detected.

This process eliminates reflection dependencies but introduces a two-stage build pipeline. The workflow requires: (1) initial JAR compilation, (2) bytecode analysis to extract instantiable applications, and (3) regeneration of the final JAR with compile-time compatible app loading mechanisms. This approach trades build complexity for runtime performance and security benefits by removing all dynamic class loading dependencies.

3.2 OS Hardening

3.2.1 Containerization

Containerization serves as our primary defense against machine compromise, implementing multiple layers of isolation and privilege restriction. We selected Podman as our container runtime due to its security-focused architecture, which operates without a privileged root daemon and provides native support for rootless containers.

Our container security model follows strict least-privilege principles. Both containers operate in rootless mode with all capabilities dropped and privilege escalation explicitly disabled through the no-new-privileges security option. The webapp container runs with a completely read-only filesystem, while the nginx container utilizes targeted tmpfs mounts only for essential runtime directories such as /var/lib/nginx/tmp and /var/run. This approach prevents any unauthorized file system modifications while preserving necessary operational capabilities.

Base image selection prioritizes security through the use of Chainguard's minimal container images, see Figure 4 and 5, which maintain zero known CVEs at deployment time. These distroless images significantly reduce the attack surface by eliminating unnecessary packages, shell access, and potential vulnerability vectors that could facilitate container escape attempts.

Network isolation is enforced through our pod architecture, see Figure 3, where the webapp container has no published ports and cannot receive direct external connections. All inbound traffic must traverse the nginx reverse proxy, which acts as a security gateway filtering and validating requests before forwarding them to the application.

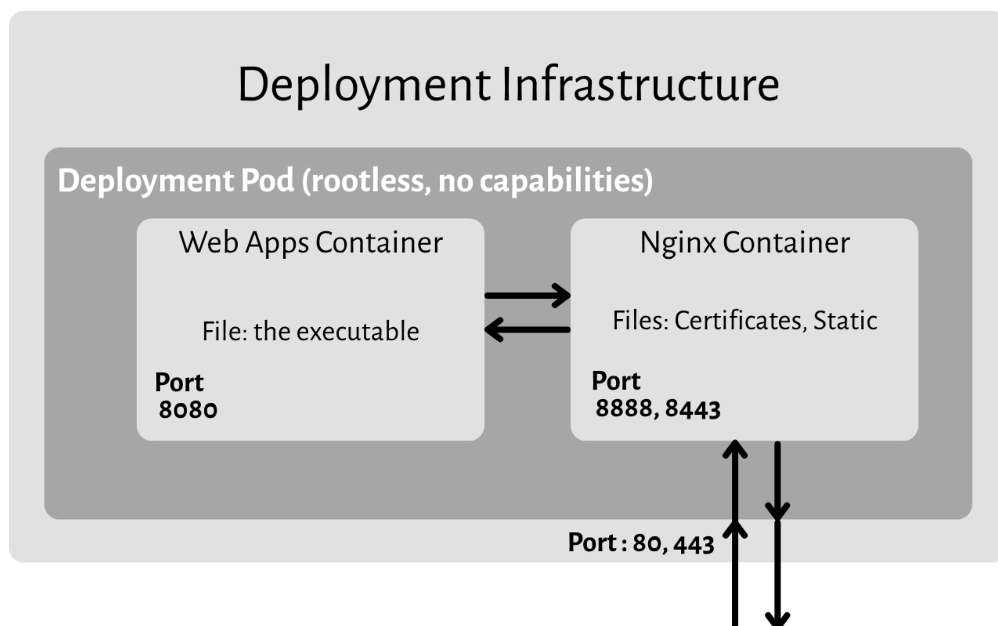

Figure 3: Deployment Pod

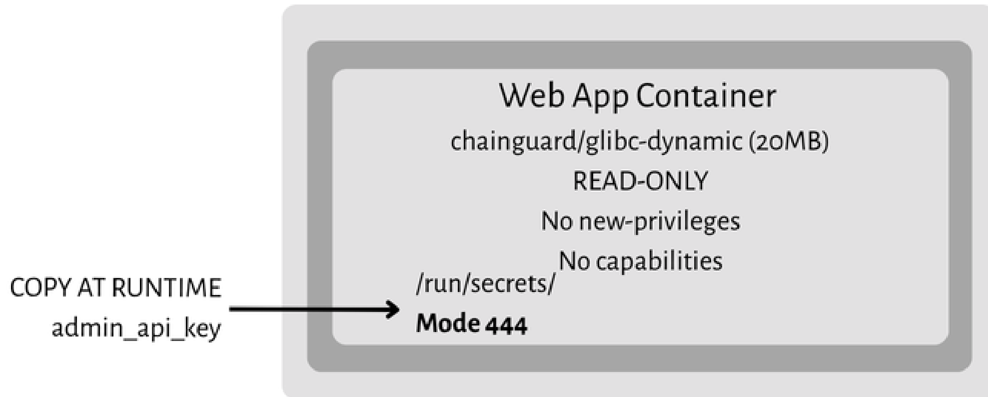

Figure 4: Web App container schema

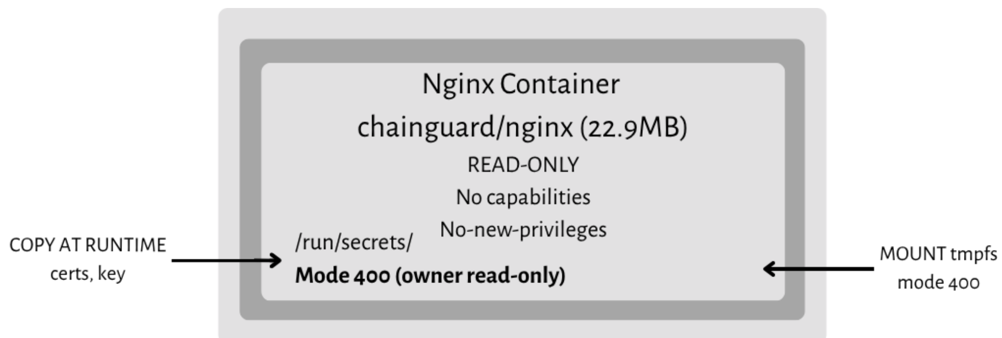

Figure 5: Deployment Pod

3.2.2 Server Hardening

For server hardening, we implemented a simple setup. We removed unused processes and reviewed open ports to minimize the attack surface. Fail2ban was deployed to prevent SSH brute force attacks by automatically banning IP addresses that exceed authentication failure thresholds, providing protection beyond the per-connection limits of ssh MaxAuthTries.

A Linux basic audit system was configured with monitoring capabilities, including file access monitoring for critical system files (/etc/passwd, /etc/shadow, /etc/sudoers), process execution tracking via execve system calls, and immutable configuration protection (-e 2) to prevent tampering. The audit system operates in maximum security mode (enabled 2) with zero data loss, providing complete visibility into user activities, privilege escalations, and system modifications.

3.2.3 CI & Automation

This security architecture introduces significant operational complexity—managing server hardening, multiple containerized applications, diverse secret management, and complex software supply chains—that could become unmanageable for our course staff.

Without proper automation, this security-focused architecture risks becoming a maintenance burden where configuration drift, inconsistent deployments, and manual errors undermine both security and reliability.

To address these challenges, I developed a CI pipeline that automates the entire build, security validation, and deployment process. The system uses documented Ansible playbooks and roles, where each task clearly defines its security purpose and operational requirements. This approach includes local testing capabilities through containerized environments that replicate production security configurations.

The Continuous Integration (CI) pipeline enforces a strict workflow that requires all changes to undergo a merge request and a review process before being deployed to production. This security measure is implemented by ensuring that the SSH deployment key is available exclusively within the GitLab CI environment, preventing unauthorized direct pushes to the production branch.

3.3 Secret Management

Nginx certificates are stored encrypted in the repository. This allows us to test in an environment that closely matches production, while keeping sensitive data safe at rest.

All private SSH keys needed to connect to the server are kept only inside the CI system. This setup ensures that deployments can only happen through the CI, enforcing code review and reducing the risk of unauthorized access.

4 Evaluation

4.1 Containers Security Assessment

The web application and Nginx containers, respectively Figure 4 and 5, were evaluated using OpenSCAP with Chainguard GPOS STIG standards (General Purpose Operating System Security Technical Implementation Guide specifically designed for containerized environments, based on DoD security requirements derived from NIST 800-53 controls and delivered as XCCDF-formatted SCAP profiles for FedRAMP and FIPS compliance validation [2] [1]), achieving both a **88.89% compliance score** (86 passed, 5 failed out of 91 rules).

Status	**Count**	**Percentage**
Passed	86	94.5%
Failed	5	5.5%

Table 1: OpenSCAP Compliance Results

All five failed controls relate to **cryptographic requirements**, specifically NSA-approved cryptography and NIST FIPS-validated cryptography (both high severity), along with data confidentiality in transmission/reception and federal cryptographic module authentication (medium severity).

These failures are **architectural by design** and do not represent security vulnerabilities. The Chainguard distroless base image intentionally excludes OS-level cryptographic modules, with cryptographic services properly delegated to the host system and application-level crypto handled by the Java runtime. This separation of concerns follows modern containerization security best practices.

Control Type	**Implementation Layer**
OS Cryptography	Host System
Application Crypto	Java Runtime
Network Encryption	Container Runtime

Table 2: Security Control Delegation

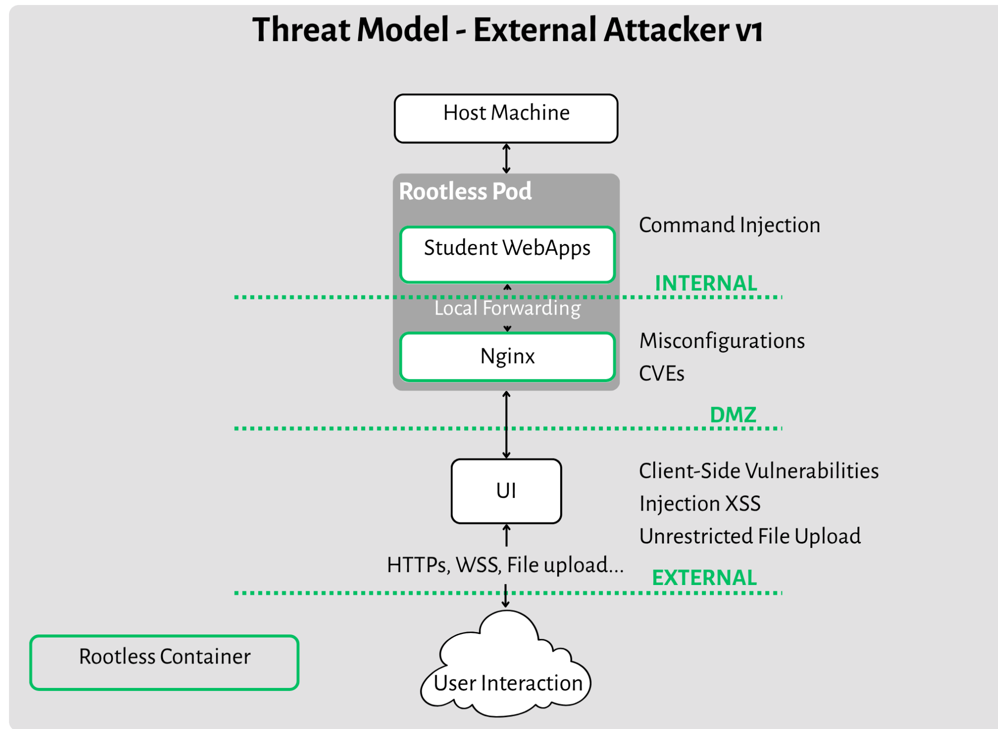

Figure 6: Threat model v1 from an external point of view after mitigation implementation

Key Enhancement: Multi-Layered OS Hardening and Container Security

1. **Containerization and Isolation:** Podman rootless containers with least-privilege principles, read-only filesystem for webapp, and targeted tmpfs mounts for nginx runtime directories to prevent unauthorized modifications.

2. **Secure Base Images:** Chainguard minimal distroless images with zero known CVEs to reduce attack surface by eliminating unnecessary packages, shell access, and potential vulnerability vectors.

3. **Network Isolation:** Pod architecture enforces no direct external access to webapp; all traffic passes through nginx reverse proxy acting as a security gateway.

4. **Server Hardening:** Removal of unused processes, port review, Fail2ban for SSH brute force protection, and Linux audit system with monitoring of critical files and process execution.

5. **Automated CI Pipeline:** Full automation of build, security validation, and deployment using documented Ansible playbooks with containerized local testing to ensure consistency and maintainability.

4.2 Bytecode Analysis & GraalVM Evaluation

4.2.1 Bytecode Analysis with GraalVM

Bytecode analysis works well with GraalVM Native Image compilation, providing static analysis capabilities. However, the limitations are dynamic behaviors that cannot be statically determined, requiring explicit monitoring and registration of reflection capabilities, JNI access points, and resource usage patterns during the build process.

4.2.2 Attack Surface Analysis

GraalVM Native Image compilation produced a **333.21MB native executable** from a **251.6MB JAR**. This represents significant deployment efficiency compared to the traditional **251.6MB JAR plus JRE** (typically 200–400MB additional overhead).

- We eliminated significant portions of the original codebase, with only 91.4% of types (20,843/22,793), 54.1% of fields (23,833/44,081), and 53.7% of methods (101,026/188,112) included in the final executable. Reflective capabilities were constrained to 4,216 types, 143 fields, and 1,218 methods, while JNI exposure was limited to 71 types, 69 fields, and 58 methods, with just 4 native libraries linked at build time.

- These results align well with published GraalVM Native Image benchmarks and documentation [4]. The official GraalVM build output documentation shows that typical native image compilation retains 72.5% of types (3,163/4,364), 50.3% of fields (3,801/7,553), and 45.5% of methods (15,183/33,405) in their HelloWorld example. Our retention rates of 91.4% types, 54.1% fields, and 53.7% methods are comparable, with slightly higher type retention likely due to our application's complexity and dependency structure. Similarly, their reflection registration (957 types, 81 fields, 480 methods) and JNI exposure (57 types, 55 fields, 52 methods) provide a reference point against our 4,216 reflection types and 71 JNI types, indicating our application requires significantly more reflective access due to its web framework dependencies.

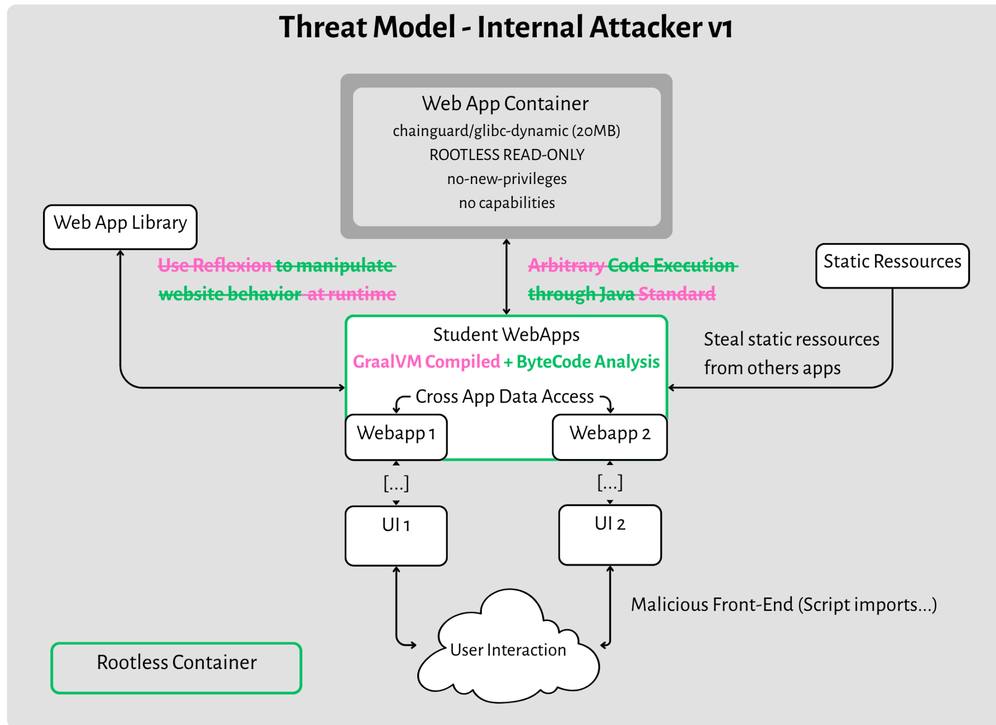

Figure 7: Threat model from a student point of view after mitigation implementation

Key Enhancement: Multi-Layered Security Through Bytecode Analysis

To harden our hosting environment, we implemented a **bytecode analysis framework** using ASM that provides three critical security enhancements:

1. **Unsafe Operation Detection:** Direct inspection of bytecode instructions to identify filesystem access, network operations, JNI calls, serialization mechanisms, and other security-critical operations that could compromise the hosting environment.
2. **Reflection Bypass Prevention:** Integration with GraalVM native compilation to whitelist all reflective calls, preventing attackers from using reflection to obfuscate malicious operations and bypass static analysis.
3. **Compile-Time Code Generation:** Elimination of runtime reflection dependencies through static analysis of class hierarchies and automated generation of compile-time instantiation code, achieving 100% accuracy in application loading while removing reflexion dependencies.

5 Discussion

Static and Runtime Analysis:
Our static analysis leverages GraalVM's closed-world assumption to identify security issues at compile time, providing a first layer of protection by catching known vulnerabilities before deployment. However, static analysis alone is insufficient for security. **Future work:** Integrating lightweight runtime analysis or anomaly detection could add a second layer, enabling the system to detect new or unexpected attacks during execution. The main challenge will be selecting or designing monitoring solutions that do not introduce significant performance overhead.

GraalVM Native Image Configuration:
Currently, generating configuration files for GraalVM native images is a manual process, requiring local compilation and limited path testing. This step takes approximately 3–5 minutes per build to produce the necessary reflection metadata, which hinders full automation of the deployment pipeline and introduces the risk of human error. **Future work:** Developing or adopting automated tools for reflection metadata generation would streamline deployment and reduce manual intervention.

Containerization Limitations and Improvements:

- **No seccomp profiles:** Neither the webapp nor the nginx container employs a seccomp profile, leaving a broad set of system calls exposed. **Future work:** Generating and enforcing custom seccomp profiles tailored to each container's needs would further reduce the kernel attack surface.

- **Network-based container communication:** The current setup uses network interfaces for communication between the webapp and nginx containers, increasing susceptibility to network-based attacks. **Future work:** Transitioning to Unix socket-based communication would enhance security by limiting the attack surface and reducing the risk of lateral movement.

Summary:
While the current approach establishes a strong security baseline, addressing these areas in future work will further enhance both the security and automation of the deployment process. The project repository is available here.

6 Timeline Expectations vs. Reality

The project followed the original proposal closely, with the timeline and major milestones proceeding as expected. While I adhered to the first and third milestones, I also dedicated substantial effort to exploring a wide range of tools and approaches, which enriched the overall outcome. The full project proposal is available here.

7 References

References

[1] Chainguard. *chainguard-dev/stigs - GPOS STIG Profile Repository*. General Purpose Operating System STIG profile for Wolfi-based container images. Chainguard. July 2024. URL: `https://github.com/chainguard-dev/stigs` (visited on 06/29/2025).

[2] Chainguard. *STIGs for Chainguard Containers*. NIST 800-53 derived STIG for containerized environments with XCCDF format. Chainguard. Apr. 2025. URL: `https://edu.chainguard.dev/chainguard/chainguard-images/features/image-stigs/` (visited on 06/29/2025).

[3] Cloud Security Alliance. *Ransomware in the Education Sector*. June 2025. URL: `https://cloudsecurityalliance.org/articles/ransomware-in-the-education-sector`.

[4] *Native Image Build Output*. Accessed: 2025-06-30. Oracle Corporation. 2025. URL: `https://www.graalvm.org/latest/reference-manual/native-image/overview/BuildOutput/`.

[5] UK Government. *Cyber security breaches survey 2025: education institutions findings*. Apr. 2025. URL: `https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2025/cyber-security-breaches-survey-2025-education-institutions-findings`.