Project Phase 2 – Security Analysis

As a Boilermaker pursuing academic excellence, we pledge to be honest and true in all that we do. Accountable together – We are Purdue.

(On group submissions, have each team member type their name).

Type or sign your names: Dylan Manning, Cecilia Jiang, Ava Lyall, Tianyu Zhang

Write today's date: 11/13/25

Assignment

This document provides a template for your system's security analysis. You are welcome to add content beyond what is listed; the template is designed to help ensure you cover all crucial components.

The terms used in this section are defined here.

Security requirements.

Here, list the *security* requirements of your system, aligned with the six security properties defined by the STRIDE article. These requirements may be unique to your system, depending on which features you implemented. It is possible that you will not have a requirement associated with every security property.

Confidentiality

- C1: Observers on the network cannot view or infer client–server interactions (TLS 1.2+ for all endpoints).
- C2: Model packages, metadata, and user data must not be readable in transit or at rest by unauthorized parties.
- C3: Sensitive models and their associated JavaScript programs must be stored in a restricted-access location (e.g., encrypted S3 bucket with IAM-based access).
- C4: Authentication tokens must never be exposed to clients other than the authenticated user (no logging of tokens, no reflection in UI).
- C5: Access to system health logs must not leak model names, user identifiers, or package metadata beyond what is minimally required.

Integrity

- I1: Uploaded model packages must not be modified without authorization; all stored artifacts must include integrity protection (e.g., checksums).
- I2: Ratings (net score, sub-scores, reproducibility, reviewedness, treescore) must be computed deterministically and not modifiable by clients.
- I3: The lineage graph must be derived from validated config metadata; tampering with lineage information must be prevented.
- I4: The “PackageConfusionAudit” must use validated historical download/search data to prevent forgery of popularity metrics.
- I5: Administrative actions (create/delete user, reset system state) must be logged immutably and protected against replay or injection.

Availability

- A1: The registry API must remain available even under high request volume (e.g., enumerate with millions of models must be paginated to avoid DoS).
- A2: A failed sensitive-model pre-download JS execution must block only that download, not the overall system operation.
- A3: The system must mitigate brute-force login attempts (rate limiting, lockout windows).
- A4: AWS components must fail gracefully—loss of S3, DynamoDB, or Lambda must produce clean error messages and not crash other services.
- A5: Health endpoints must respond even under partial system degradation to avoid cascading failures.

Authentication

- AU1: Only authenticated users may upload, search, ingest, or download packages.
- AU2: Username + secure password authentication must be required; passwords stored with salted, memory-hard hashing (Argon2/BCrypt/scrypt).
- AU3: Tokens must be unforgeable, bound to a user, expire after 1000 API calls or 10 hours, and be invalidated on logout/deletion.
- AU4: Sensitive-model downloads must record the downloader's authenticated identity for audit.
- AU5: Admin-only actions must require elevated privileges that cannot be spoofed by manipulating tokens.

Authorization

- AZ1: Only users with "upload" permissions can upload or ingest models.
- AZ2: Only users with "download" permissions can retrieve model packages or sub-assets.
- AZ3: Only administrators can register or delete other users.
- AZ4: Users can delete only their own accounts unless they are administrators.
- AZ5: Sensitive-model actions (uploading/deleting JS monitoring scripts, viewing history) must be restricted to authorized users.
- AZ6: JS monitoring programs must not allow privilege escalation (e.g., a user must not provide a script that grants themselves admin rights).

Nonrepudiation

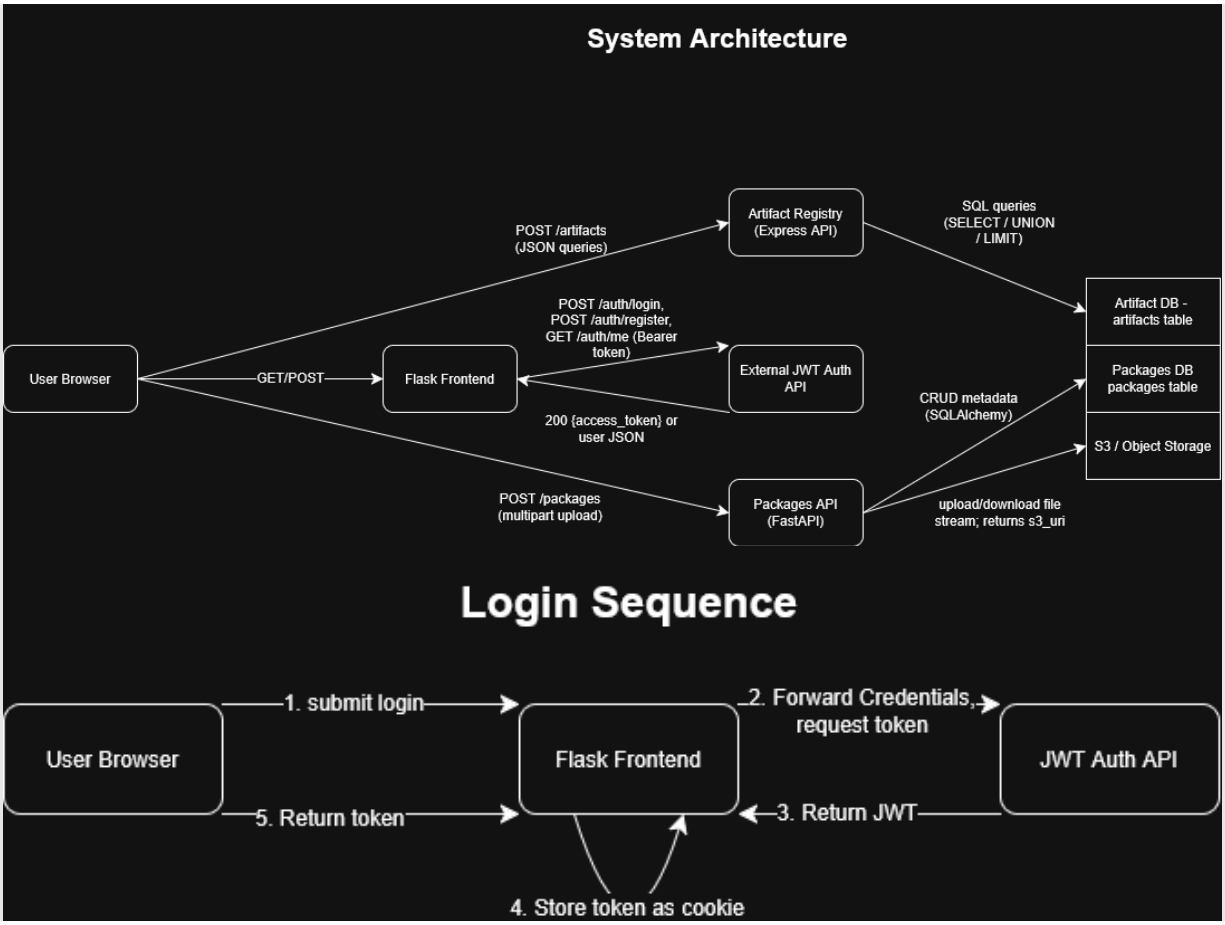
- NR1: All uploads, ingests, downloads, and administrative actions must be logged with timestamp, user identity, IP, and request identifiers.
- NR2: Sensitive-model downloads must capture downloader identity and the output of the pre-download JS program.
- NR3: Model ratings and lineage calculations must be traceable to the version of the algorithm used at the time of computation.
- NR4: Reset operations must leave an immutable audit trail to prevent users from claiming actions did not occur before the reset.

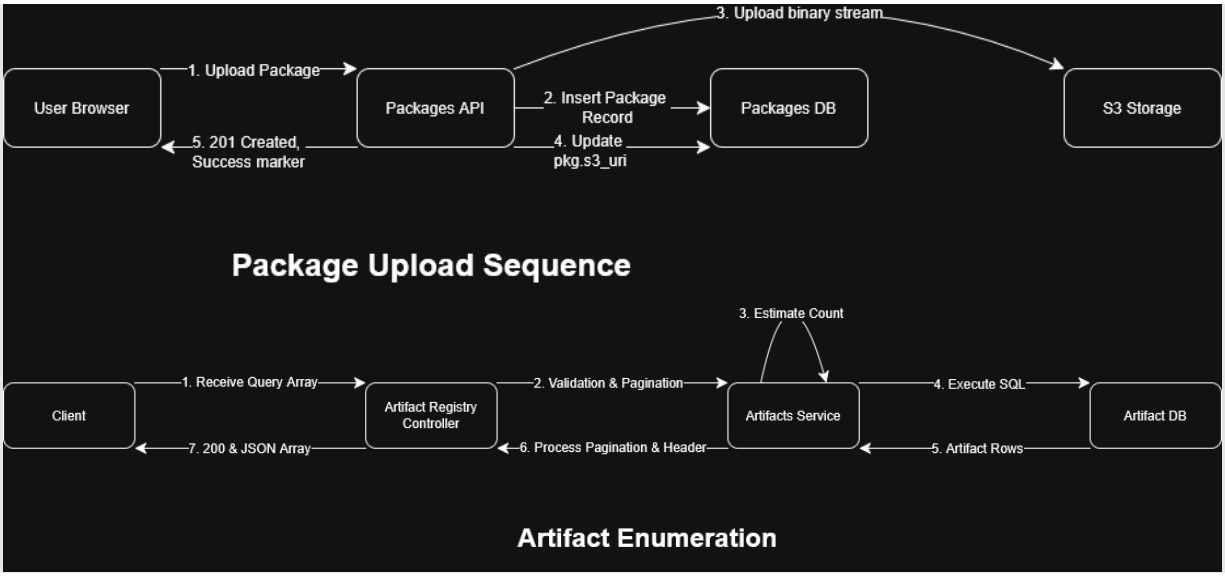
- NR5: Tokens must be uniquely tied to users to prevent plausible deniability of actions performed with them.

System model via DFDs

Present one or more data-flow diagrams of your deployed system. You can use some drawing tools such as https://app.diagrams.net/. A whiteboard picture is also fine, but use the correct symbols please – process, data sink, etc.

- You may provide multiple DFDs to capture different aspects of the system (e.g. one per feature or feature group).
- You may wish to indicate multiple trust boundaries, e.g. for different classes of users or for external components.
- Each diagram should indicate **at least** the following entities: data flow; data store; process; trust boundary. You may include interactors and multi-process if needed.





Threats

For each trust boundary indicated in the DFDs, describe the nature of the untrusted party involved (examples: "outsider threat [e.g. external hacker]" or "insider threat [e.g. ACME employee with valid credentials]" or "infrastructure provider threat [AWS]"), why the location of the trust boundary is appropriate, and a potential attack that could occur were this trust boundary not secured.

Trust boundary #1: User Browser => Flask Frontend
Untrusted party: External users on the public internet (could be benign users or malicious outsiders).
Rationale for this boundary: This boundary separates unauthenticated, untrusted client devices from the first backend entry point. It is necessary because all input originates here and cannot be assumed trustworthy. Attackers may manipulate requests, intercept traffic, or attempt unauthorized access.
Potential attack across this boundary: A rogue admin could delete audit logs, generate arbitrary tokens, change permissions, or alter sensitive lineage/rating computations. A stolen admin credential would yield full system takeover.

Trust boundary #2: Flask Frontend => JWT Auth API
Untrusted party: Potentially compromised frontend or attackers relaying forged requests through it.
Rationale for this boundary: Although the frontend is part of the system, it runs in a lower-trust environment than the authentication service. The Auth API must assume the frontend could send malformed or manipulated credential requests.
Potential attack across this boundary: An attacker could attempt credential-stuffing, token forgery, or replay attacks. Without proper authentication checks, the Auth API might issue valid tokens based on manipulated login requests.

Trust boundary #3: Flask Frontend => Package/Artifact APIs
Untrusted party: Authenticated but potentially malicious users or compromised accounts.

Rationale for this boundary: The frontend forwards privileged operations such as uploading packages, ingesting metadata, or listing sensitive artifacts. The APIs must enforce authorization independently because the frontend cannot be trusted to enforce permissions perfectly.
Potential attack across this boundary: A user could escalate privileges by submitting forged JWTs, attempt unauthorized model uploads, try to retrieve sensitive model assets, or inject malformed metadata to corrupt lineage graphs or ratings.

Trust boundary #4: Package API => S3 Storage
Untrusted party: Cloud storage infrastructure (AWS S3) and the network path to it.
Rationale for this boundary: The package service trusts its internal logic, but once data is transmitted to S3 it relies on AWS infrastructure, IAM policies, and bucket configurations. Misconfigurations or AWS compromise create risk.
Potential attack across this boundary: Without proper IAM scoping and encryption, an attacker could read or overwrite stored package binaries, inject malicious model files, or delete assets leading to data loss or inconsistent registry state.

Trust boundary #5: Artifact Service => Artifact Database
Untrusted party: Database infrastructure, OR authenticated insiders misusing database privileges.
Rationale for this boundary: Although internal, the DB sits behind a separate trust boundary because it stores authoritative ratings, lineage data, and metadata. Any unchecked input sent to the DB could compromise system integrity.
Potential attack across this boundary: An attacker could exploit SQL injection to modify ratings, falsify lineage, or tamper with popularity metrics. A malicious insider with DB credentials could alter records undetected without integrity checks.

Trust boundary #6: External Network => AWS Services
Untrusted party: The cloud provider (infrastructure provider threat), plus the public network between services.
Rationale for this boundary: Even though these components are part of the system, they operate under AWS's control. The system must trust AWS to maintain availability, confidentiality, and integrity, but it is still a separate trust domain.
Potential attack across this boundary: Misconfigured IAM roles or policy errors could allow attackers to enumerate storage buckets, delete records, or exfiltrate model packages. A compromised AWS account or leaked API key could lead to total system compromise.

Trust boundary #7: Admin Users => Administrative Control Plane
Untrusted party: Insider threat (malicious administrator or compromised admin credentials).
Rationale for this boundary: Admin interfaces allow resetting system state, deleting accounts, and modifying metadata. These privileges exceed those of normal users and represent a higher-risk trust domain.
Potential attack across this boundary: A rogue admin could delete audit logs, generate arbitrary tokens, change permissions, or alter sensitive lineage/rating computations. A stolen admin credential would yield full system takeover.

AI-Assisted Threat Modeling and Feedback

After completing your DFD(s), consult a Large Language Model (LLM) for a preliminary security analysis. The goal is to generate a broad set of security considerations and best practices for your group to critically evaluate.

1. **Request Specific Analysis:** Ask the LLM for a component-by-component security analysis. Key requests should include:
    - Specific advice on securely configuring the services you are using (e.g., "What are the security best practices for configuring an AWS S3 bucket that stores user files?" or "How should we securely configure our AWS RDS instance?").
    - General feedback on potential design weaknesses in your data flows or trust boundaries.
2. **Sub-Group Analysis:** Divide your team into two sub-groups. Each sub-group will independently query an LLM using the group's DFD. After gathering feedback:
    - As a full group, consolidate and compare the results.
    - Discuss similarities and differences in the LLM's advice.
    - Critically evaluate which recommendations are most relevant and actionable for your project.
3. **Summarize Findings:** In the area below, summarize this process. You **must** detail the specific LLM analysis or recommendations that you found most helpful or insightful. Discuss how this feedback informed your team's final security analysis and mitigation strategies.

After both sub-groups consulted an LLM using our completed DFD, we met as a full team to consolidate and evaluate the results. Several of the LLM's recommendations were especially relevant and helped refine our final threat model and mitigation strategies.

One of the most consistently helpful insights came from the LLM's guidance on securely configuring our AWS S3 bucket. Both sub-groups received nearly identical warnings about misconfiguration risks. One LLM response stated, "Ensure that the S3 bucket has 'Block Public Access' fully enabled, requires TLS for all requests, and uses bucket-level IAM policies instead of ACLs. Enable server-side encryption (SSE-S3 or SSE-KMS) and turn on object versioning to protect against accidental or malicious deletions." This aligned with several threats we had identified but expanded our thinking about what misconfigurations attackers might exploit. As a result, we strengthened our mitigations to include S3 versioning, access logs, and stricter IAM scoping.

We also found the LLM's analysis of our Artifact Database extremely useful. One excerpt our group discussed was "A database connected directly to internal services should assume that the upstream API could be compromised. Enforce least-privilege IAM roles, restrict access to a private subnet, and require parameterized queries to eliminate injection risks. Consider enabling point-in-time recovery and CloudWatch alerts for unusual read/write patterns." This helped us re-evaluate the database trust boundary and incorporate additional controls like anomaly monitoring and stronger assumptions about insider threats within internal services.

Another valuable area of feedback addressed the authentication and API communication flow in our design. One LLM response highlighted that "Frontend applications should never be trusted to enforce authorization logic. Each backend API must independently validate JWT signatures, token expiry, and user roles. Implement rate limiting at the API gateway level to prevent brute-force or credential-stuffing attempts." This recommendation directly influenced our evaluation of threats across the User to Frontend and Frontend to Auth API boundaries. We originally focused on input validation, but after this guidance, we added rate limiting and token-replay protection as explicit mitigations.

The LLM also pointed out a design weakness none of us had discussed before, which is the possibility of metadata corruption affecting lineage and rating calculations. One particularly insightful passage said, "If model metadata or lineage submissions are not validated, an attacker could inject malformed or adversarial metadata that misrepresents model origins, manipulates popularity metrics, or breaks downstream integrity checks." This led us to add new lineage-validation controls and sanity checks in the Artifact Service, which were not in our original model.

When we compared the outputs from both sub-groups, we noticed several overlaps, such as encryption-at-rest, IAM least privilege, and centralized validation, but some recommendations appeared in only one set of outputs. For example, one LLM strongly emphasized logging and traceability, stating, "Enable immutable audit logs using CloudTrail with write-once S3 storage. This is critical for responding to insider threats or compromised service credentials." The other LLM focused more on abuse prevention, with advice like: "Treat authenticated users as untrusted when calling internal APIs. Avoid assuming that a valid token implies safe behavior." Reviewing both sets helped us triangulate which risks were the most foundational versus which were tied to more specific threat scenarios.

Overall, the LLM-assisted threat modeling significantly strengthened our security analysis. It helped surface misconfigurations we had not previously considered, validated several of our original threat boundaries, and provided concrete best practices that we were able to incorporate directly into our mitigation strategies. The process of comparing the two independent LLM outputs also helped highlight common high-priority issues as well as unique insights one sub-group may have otherwise overlooked.

Analysis of Threats and Mitigations

Fill out the following table for each STRIDE property (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege). For each STRIDE category, you must list specific potential threats, brainstorm multiple mitigation strategies, and then label each threat as either "Should Fix" or "Won't Fix". You must justify any "Won't Fix" decisions by explaining why the risk is low-priority, implausible, or out of scope. For every "Should Fix" threat, you must identify a single, specific mitigation strategy you plan to implement.

Note: It is acceptable if you are unable to implement *all* mitigation strategies you planned by the deadline for *this* document. However, for the final project submission, you will be required to have implemented *all* mitigation strategies for every threat you labeled as "Should Fix" in this document.

STRIDE property: Spoofing
Relevant system security properties: Authentication
Analysis of components:

- Diagram+Component: Login sequence, component 1
  - Risk 1: Attackers may attempt to impersonate a legitimate user by forging or stealing an auth token.
    - Possible Mitigations: Use signed JWTs, enforce token expiration, validate tokens server-side before all sensitive operations.
    - How do these address the risk: They ensure that even if a token is stolen or modified, invalid signatures or expired timestamps prevent unauthorized access.
    - Suggestions for additional mitigations, if needed: IP-based session throttling; device-based token binding.
    - **Decision: Should Fix**
    - Justification and Plan: Implement server-side JWT verification and strict expiration checks in the ingestion, search, and download endpoints.
  - Risk 2: An attacker could submit a model upload request using a spoofed client identity.
    - Possible Mitigations: Require login before upload; check user identity against request metadata.
    - How do these address the risk: Ensures that only authenticated users can perform actions that mutate system state.
    - Suggestions for additional mitigations, if needed: Add rate limiting for repeated authentication failures.
    - **Decision: Should Fix**
    - Justification and Plan: Require valid authentication tokens on all upload endpoints.

STRIDE property: Tampering
Relevant system security properties: Integrity
Analysis of components:

- Diagram+Component: System Architecture, S3/Object Storage
  - Risk 1: A malicious user could upload modified or corrupt model files to poison the registry.
    - Possible Mitigations: Verify file hash/size on upload; validate model structure using schema checks.
    - How do these address the risk: Prevents malformed or intentionally broken files from being stored.
    - Suggestions for additional mitigations, if needed: Store integrity metadata (SHA256) for each model.
    - **Decision: Should Fix**
    - Justification and Plan: Implement hash verification as part of the ingestion pipeline.

- Risk 2: An attacker may modify metric results by injecting malicious values.
  - Possible Mitigations: Ensure metrics are computed server-side only; sanitize any input that flows into computations.
  - How do these address the risk: Prevents client-provided data from influencing trust scores.
  - Suggestions for additional mitigations, if needed: Add strict type validation for all metric fields.
  - **Decision: Should Fix**
  - Justification and Plan: Restrict metric calculation to controlled backend code.

STRIDE property: Repudiation
Relevant system security properties: Non-repudiation
Analysis of components:

- Diagram+Component: Package Upload Sequence, Component 2
  - Risk 1: Users may deny uploading a harmful or broken model.
    - Possible Mitigations: Keep timestamped audit logs of uploads and operations.
    - How do these address the risk: Provides a verifiable record of associating actions with users.
    - Suggestions for additional mitigations, if needed: Add IP logging for admin use only.
    - **Decision: Should Fix**
    - Justification and Plan: Implement simple logging for all mutations (upload, delete, reset).
  - Risk 2: Admin may reset registry without trace.
    - Possible Mitigations: Log reset events, including admin ID and timestamp.
    - How do these address the risk: Ensures traceability of high impact events.
    - Suggestions for additional mitigations, if needed: Send reset confirmation to a secured admin email.
    - **Decision: Won't Fix**
    - Justification and Plan: Email notification is out of scope since no real user is involved.

STRIDE property: Information disclosure
Relevant system security properties: Confidentiality
Analysis of components:

- Diagram+Component: Package Upload Sequence, Component 2
  - Risk 1: Model files or user tokens may be intercepted during transmission
    - Possible Mitigations: Use HTTPS for all endpoints.
    - How do these address the risk: Encrypts traffic, preventing network observers from reading sensitive data.
    - Suggestions for additional mitigations, if needed: Disable outdated TLS versions.
    - **Decision: Should Fix**
    - Justification and Plan: Deploy the system behind HTTPS using provided cloud environment or local certificates.
  - Risk 2: Logs may accidentally reveal model content or user identities.
    - Possible Mitigations: Avoid logging model files, redact sensitive fields.
    - How do these address the risk: Limits exposure if logs are accessed by unauthorized users.
    - Suggestions for additional mitigations, if needed: Access control for logs.
    - **Decision: Won't Fix**
    - Justification and Plan: Given the course's conclusion in less than two weeks, implementing fine-grained log control is not feasible due to the current workload.

STRIDE property: Denial of service

- *NB: Remember the ReDoS demonstration in class?*

Relevant system security properties: Availability
Analysis of components:

- Diagram+Component: System Architecture, POST/artifacts(JSON queries)
  - Risk 1: ReDoS via expensive regex search queries.
    - Possible Mitigations: Limit regex length and complexity, enforce timeouts.
    - How do these address the risk: Prevents expensive regex evaluation from blocking the server.
    - Suggestions for additional mitigations, if needed: Precompile safe regex patterns.
    - **Decision: Should Fix**
    - Justification and Plan: Implement regex length limits and evaluation timeout.
  - Risk 2: User repeatedly uploads large files to exhaust storage or bandwidth.
    - Possible Mitigations: Limit file size and enforce per user rate limits.
    - How do these address the risk: Stops spam uploads from overwhelming storage.
    - Suggestions for additional mitigations, if needed: Quota system per user.

        - **Decision: Should Fix**
        - Justification and Plan: Set upload size limit (e.g., 200MB) and restrict repeated uploads.

STRIDE property: Elevation of privilege
- *NB: If you implemented the "JSProgram" feature, tread carefully.*

Relevant system security properties: Authorization

Analysis of components:
- Diagram+Component: Login Sequence, Component 4
    - Risk 1: Regular users may access admin only reset endpoint.
        - Possible Mitigations: Role based access control and server-side token role verification.
        - How do these address the risk: Ensures only admins can execute high impact operations.
        - Suggestions for additional mitigations, if needed: Session based admin security prompts.
        - **Decision: Should Fix**
        - Justification and Plan: Implement role check before performing reset.
    - Risk 2: Attackers may bypass safety checks by invoking internal audit logic manually.
        - Possible Mitigations: Ensure backend enforces audit before downloading.
        - How do these address the risk: Prevents user-controlled code from influencing privileged operations.
        - Suggestions for additional mitigations, if needed: Checksum verification for audit binary.
        - **Decision: Won't Fix**
        - Justification and Plan: Due to the project's imminent deadline, we are limiting scope to a backend controlled flow.

Risks resulting from component interactions

The STRIDE framework (per Microsoft) advises you to divide and conquer – analyze each component in turn. You have now done that.

Are there any instances in your system where a risk emerges from the interaction of multiple components? If you can, identify one case and describe it with reference to your DFDs. Did you find it during your STRIDE process or only just now? (1 paragraph)

In our enumerate and search API implementation, we encountered a security-relevant issue that emerged from the interaction between PostgreSQL's DISTINCT clause and ORDER BY functionality. When implementing the POST /artifact/byRegEx endpoint, we needed to remove duplicate results while maintaining sorted output. PostgreSQL requires that any ORDER BY columns must appear in the SELECT list when using DISTINCT, but our initial query only selected the columns needed for the API response. This caused queries to fail with error code 42P10. While this appears to be a functionality issue, it creates a Denial-of-Service vulnerability where malicious users could craft regex patterns knowing the query would fail, repeatedly triggering database errors and consuming server resources handling exceptions. This vulnerability only emerged from the interaction between two valid SQL features and wasn't visible when analyzing each component in isolation. We discovered this during integration testing, not during our initial STRIDE analysis of individual components.

Are there instances in your system where the requirements of one component (e.g., security, correctness, performance) may negatively affect the security requirements of another component or the system? If you can, identify one case and describe it. (1 paragraph)

Our database connection pooling implementation creates a tension with our authentication middleware security requirements. The connection pool is designed for performance maintaining persistent connections and reusing them across multiple requests to minimize connection overhead. However, our authentication middleware needs to validate user credentials and permissions on every request. The security requirement of the auth middleware conflicts with the performance requirement of connection pooling. If not carefully implemented, this could allow privilege escalation where a connection established by an admin user gets reused for a non-admin request, or vice versa. Our mitigation is to ensure that user context is never stored at the connection level, and all authorization checks happen at the application layer before query execution, not at the database connection level. This adds latency to every query but is necessary for security.

Root cause analysis

Presumably (1) you did not intentionally create any security vulnerabilities, yet (2) found some through this process. Choose two interesting vulnerabilities. Describe why they happened.

Example 1:

- Succinctly describe the vulnerability and mitigation (1-2 sentences)
- How did it get through your design and review process? (2-3 sentences)

Succinct description**:** During database migration setup, we used query<T extends any> instead of query<T extends QueryResultRow> to fix a TypeScript compilation error, which eliminated type safety protections against SQL injection vulnerabilities.
How it got through design and review**:** The TypeScript error appeared during migration testing under time pressure approaching Deliverable #1, so we focused on "making it compile" rather than security implications. Our STRIDE analysis focused on runtime security (input validation) but didn't consider compile-time type safety as part of defense-in-depth.

Additionally, the error occurred in a migration script rather than main application code, so we didn't recognize that the fix pattern would be copied to other database queries, spreading the vulnerability.

Example 2:

- Succinctly describe the vulnerability and mitigation
- How did it get through your design and review process?

Succinct description**:** Our regex search query failed with PostgreSQL error 42P10 when using DISTINCT with ORDER BY. The fix exposed internal database fields in API responses, creating an information disclosure vulnerability that leaked artifact ingestion patterns.

How it got through design and review**:** We focused solely on making the query work rather than validating the fix against our OpenAPI specification. Our testing validated "does it return results" but not "does it return only specified fields," and we lacked integration tests comparing response structure against the OpenAPI schema. The fix was made by backend developers who weren't involved in writing security requirements, creating knowledge silos where SQL optimization and API security compliance were handled separately.