TEAM 9 – THREAT MODELING REPORT

1. System Overview

Our system is a trustworthy package management system designed for secure, scalable, and efficient operations. While similar to other course projects in terms of using TypeScript, AWS, and serverless architecture, our project has distinct implementations and design choices:

1. **Frontend**:
    - React-based application styled with TailwindCSS for fast and responsive UI development.
    - Integrated with backend APIs to:
        - Upload and rate packages.
        - Perform regex-based package searches.
        - View package metadata, ratings, and cost calculations.
    - Includes role-based access controls (RBAC) for admin-specific features like resetting the registry.
    - Debugging features allow developers to directly interact with endpoints for testing (e.g., /last_run endpoint).
2. **Backend**:
    - Built on AWS Lambda for serverless scalability.
    - Deployed via API Gateway for secure and efficient routing of requests.
    - Core API endpoints:
        - authenticate: Handles token-based user authentication.
        - createPackage and updateSinglePackage: Allow secure creation and updating of packages.
        - ratePackage: Calculates package trustworthiness metrics.
        - regex: Enables complex search queries using regex patterns.
        - resetRegistry: Resets the system state (admin-only feature).
    - Functions written in TypeScript, leveraging its static typing to reduce runtime errors.
3. **Storage**:
    - Packages are stored securely in AWS S3, with metadata maintained in DynamoDB.
    - Dependency management:
        - Metadata is validated against the npm registry for accuracy.

    - Circular dependency checks are implemented to prevent ingestion errors.

4. **CI/CD Integration**:
    - Automated pipelines using GitHub Actions and AWS CodePipeline for:
        - Continuous testing of API endpoints.
        - Deployment to AWS Lambda and S3.
    - Logs and monitoring via CloudWatch, enabling real-time debugging.

---

2. Key Design Decisions

- **Security by Design**:
    - Role-based access (admin vs. user) is enforced at the API Gateway level.
    - JWTs are used for authentication, ensuring tokens are tamper-proof and time-bound.
    - Strict CORS policies are applied in production to restrict access to trusted origins.
- **Developer-Focused Debugging**:
    - The /last_run endpoint was created to simplify backend testing.
    - Dynamic logs in CloudWatch are used to analyze API performance and errors.
- **Scalable Storage**:
    - DynamoDB is used for fast metadata retrieval.
    - S3 ensures secure and scalable storage for package files.

---

3. STRIDE Analysis (Intro)

To ensure security and reliability, a STRIDE analysis was conducted across key trust boundaries in our system:

- **Frontend ↔ API Gateway**:
    - Protecting API endpoints from unauthorized access and malicious inputs.
- **API Gateway ↔ Lambda**:
    - Securing request payloads to prevent tampering and injection attacks.
- **Lambda ↔ S3**:
    - Ensuring stored packages and metadata are accessed only by authorized services.
- **Deployment Pipelines (GitHub ↔ CodePipeline)**:
    - Preventing supply chain attacks and ensuring secure deployments.

Each boundary was evaluated using STRIDE categories, and specific mitigations were implemented to address identified risks.

What Makes Our System Unique

- **Custom Solutions**:
  - Circular dependency detection and resolution during package ingestion.
  - Dynamic regex evaluation with built-in input validation to handle complex queries.
- **Advanced Debugging**:
  - Development-specific endpoints (e.g., /last_run) for real-time testing.
- **Optimized Performance**:
  - Use of AWS X-Ray for tracking API latency and pinpointing performance bottlenecks.
  - Lambda concurrency limits to ensure stable handling of high traffic.

4. STRIDE Analysis in Detail

4.1 Frontend ↔ API Gateway

- **Trust Boundary**: Communication between the frontend and the backend's API Gateway.
- **Threats and Mitigations**:
  1. **Spoofing**:
     - **Threat**: Attackers could forge tokens to access restricted API endpoints.
     - **Mitigation**:
       - Use JSON Web Tokens (JWTs) with HMAC SHA256 to ensure secure token signing.
       - Expire tokens after short durations and provide a refresh mechanism.
       - Use HTTPS to prevent token interception in transit.
  2. **Tampering**:
     - **Threat**: Malicious actors could manipulate API requests or inject invalid payloads.
     - **Mitigation**:
       - Validate payloads using JSON schemas before processing.
       - Reject invalid or unauthorized fields with meaningful error responses.

3. **Denial of Service (DoS)**:
    - **Threat**: High-volume requests to APIs (e.g., /regex) could overload the backend.
    - **Mitigation**:
        - Enable rate-limiting and throttling in API Gateway.
        - Monitor incoming traffic with AWS WAF and trigger alerts for unusual patterns.

4.2 API Gateway ↔ Lambda

- **Trust Boundary**: Routing and processing of requests between API Gateway and Lambda functions.
- **Threats and Mitigations**:
    1. **Tampering**:
        - **Threat**: Malformed payloads could exploit vulnerabilities in Lambda functions.
        - **Mitigation**:
            - Enforce strict schema validation for all inputs.
            - Use parameterized queries to avoid injection attacks when interacting with DynamoDB.
    2. **Information Disclosure**:
        - **Threat**: Sensitive data (e.g., package metadata or logs) could be exposed through API responses.
        - **Mitigation**:
            - Limit API responses to only necessary fields, avoiding sensitive data exposure.
            - Log sensitive information only in secure systems like CloudWatch with restricted access.
    3. **Elevation of Privileges**:
        - **Threat**: Unauthorized users may attempt to access admin-only APIs (e.g., /resetRegistry).
        - **Mitigation**:
            - Validate JWT roles for all sensitive APIs, ensuring only admin users can access them.
            - Audit all admin-level actions in CloudWatch.

4.3 Lambda ↔ S3

- **Trust Boundary**: Access and management of package data between Lambda functions and S3 buckets.
- **Threats and Mitigations**:
    1. **Tampering**:
        - **Threat**: Unauthorized modification of stored packages or metadata.
        - **Mitigation**:
            - Use S3 server-side encryption (SSE-S3 or SSE-KMS) to secure stored data.
            - Apply bucket policies to restrict access to specific Lambda roles.
    2. **Information Disclosure**:
        - **Threat**: Exposing package metadata or S3 bucket names through misconfigured APIs.
        - **Mitigation**:
            - Mask S3 bucket names in API responses and only provide necessary metadata.
            - Encrypt package metadata stored in DynamoDB.
    3. **Denial of Service (DoS)**:
        - **Threat**: Overloading S3 with repeated requests or large payloads.
        - **Mitigation**:
            - Set request rate limits on Lambda functions interacting with S3.
            - Use S3 versioning to prevent accidental overwrites or excessive file storage.

4.4 Deployment Pipelines (GitHub ↔ CodePipeline)

- **Trust Boundary**: Code and configuration files moving through CI/CD pipelines.
- **Threats and Mitigations**:
    1. **Spoofing**:
        - **Threat**: Unauthorized users could push code to repositories or trigger pipeline runs.
        - **Mitigation**:
            - Require signed commits for all contributions.
            - Use webhook secrets to authenticate GitHub to CodePipeline interactions.
    2. **Tampering**:

- **Threat**: Malicious actors could inject vulnerabilities via compromised code.
- **Mitigation**:
  - Conduct automated dependency scans and vulnerability checks during CI builds.
  - Require peer code reviews before merging to the main branch.

3. **Information Disclosure**:
   - **Threat**: Exposing sensitive data like access tokens in pipeline logs.
   - **Mitigation**:
     - Mask sensitive values in CI/CD pipelines using environment variables.
     - Restrict pipeline log access to authorized users only.

4.5 Deployment Pipelines

- **Trust Boundary**: Code and configuration files moving through CI/CD pipelines.
- **Threats and Mitigations**:

1. **Spoofing**:
   - **Threat**: Unauthorized users could push code to repositories or trigger pipeline runs.
   - **Mitigation**:
     - Require signed commits for all contributions.
     - Use webhook secrets to authenticate GitHub to CodePipeline interactions.
   - **How this addresses the threat**: Signed commits ensure code changes are authenticated and traceable, while webhook secrets prevent unauthorized access to the pipeline.
2. **Tampering**:
   - **Threat**: Malicious actors could inject vulnerabilities via compromised code.
   - **Mitigation**:
     - Conduct automated dependency scans and vulnerability checks during CI builds.
     - Require peer code reviews before merging to the main branch.
   - **How this addresses the threat**: Dependency scans identify and mitigate vulnerabilities, and peer reviews ensure code quality and safety.
3. **Information Disclosure**:
   - **Threat**: Exposing sensitive data like access tokens in pipeline logs.

  - **Mitigation**:
    - Mask sensitive values in CI/CD pipelines using environment variables.
    - Restrict pipeline log access to authorized users only.
  - **How this addresses the threat**: Prevents sensitive information leakage by securing logs and limiting access.

4. **Repudiation**:
   - **Threat**: Developers or contributors could deny submitting code changes or triggering builds.
   - **Mitigation**:
     - Require signed commits for all contributions to tie changes to specific individuals.
     - Maintain detailed logs of all pipeline activities using AWS CloudWatch.
   - **How this addresses the threat**: Ensures every action is traceable, reducing the ability to deny responsibility.

4.6 Summary of STRIDE Mitigations

Trust boundary	Threat	Mitigation
Frontend ↔ API Gateway	Spoofing	JWTs, HTTPS, and short token lifespans.
	Tampering	Input validation via JSON schemas.
	DoS	Rate-limiting and AWS WAF.
API Gateway ↔ Lambda	Tampering	Schema validation and parameterized queries.
	Information Disclosure	Limit sensitive data in responses, secure logging.
	Elevation of Privileges	Validate JWT roles, audit admin actions.
Lambda ↔ S3	Tampering	S3 encryption, restricted bucket policies.
	Information Disclosure	Mask bucket names, encrypt DynamoDB metadata
	DoS	Set request rate limits, enable S3 versioning.
Deployment Pipelines	Spoofing	Signed commits, webhook secrets.
	Tampering	Signed commits, webhook secrets.
	Information Disclosure	Mask sensitive pipeline logs, restrict log access.

Repudation	repudation	Signed commits, detailed logs in CloudWatch.

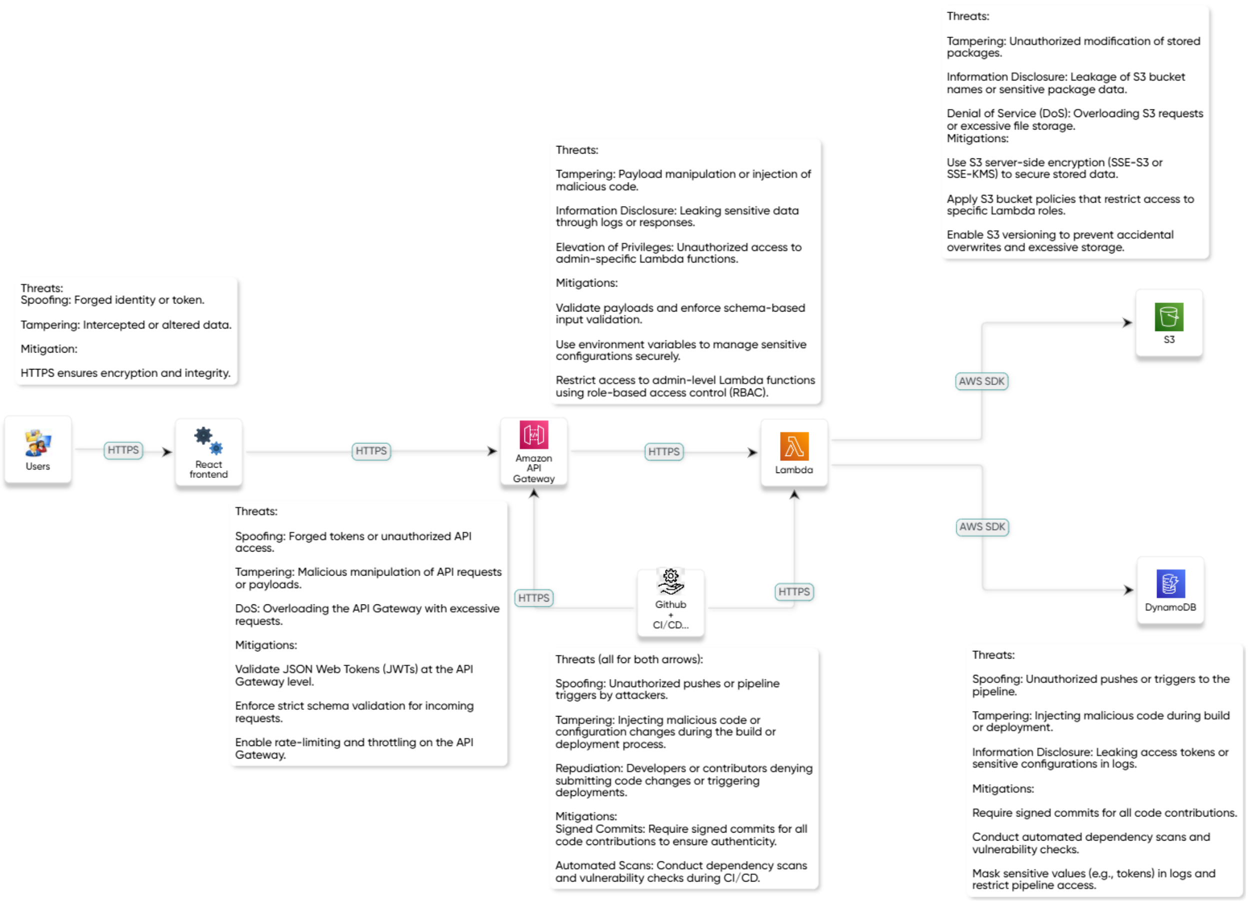
5. Identified Vulnerabilities and Mitigations

Through our detailed STRIDE analysis, we identified the following critical vulnerabilities in our system and implemented effective mitigations to address them:

Vulnerability	Threat category	Mitigation
Weak token validation logic	Spoofing	Standardized JWT validation across APIs; enforced HTTPS.
Misconfigured CORS policies	Information Disclosure	Restricted CORS policies to trusted frontend origins
Missing or incomplete dependency metadata	Tampering	Fallback to npm registry for missing metadata; added circular dependency checks.
Over-permissioned S3 bucket access	Elevation of Privileges	Applied least-privilege bucket policies and role-based access controls.
Malformed regex input causing crashes	Denial of Service (DoS)	Validated regex input; added timeouts for regex evaluations.
Unauthorized API access to admin endpoints	Elevation of Privileges	Enforced RBAC through JWT roles; logged all admin actions in CloudWatch.
Exposure of sensitive data in CI/CD logs	Information Disclosure	Masked sensitive pipeline values and restricted access to deployment logs.
Public exposure of API vulnerabilities	Reputation	Implemented proactive vulnerability scans and regular security audits; published responsible disclosure policies to handle security reports effectively.

These vulnerabilities highlight areas where proactive measures were taken to secure the system while maintaining functionality and performance.

6. Visual Threat Model Diagram



Threats:
Spoofing: Forged identity or token.

Tampering: Intercepted or altered data.

Mitigation:

HTTPS ensures encryption and integrity.

Threats:

Spoofing: Forged tokens or unauthorized API access.

Tampering: Malicious manipulation of API requests or payloads.

DoS: Overloading the API Gateway with excessive requests.

Mitigations:

Validate JSON Web Tokens (JWTs) at the API Gateway level.

Enforce strict schema validation for incoming requests.

Enable rate-limiting and throttling on the API Gateway.

Threats:

Tampering: Payload manipulation or injection of malicious code.

Information Disclosure: Leaking sensitive data through logs or responses.

Elevation of Privileges: Unauthorized access to admin-specific Lambda functions.

Mitigations:

Validate payloads and enforce schema-based input validation.

Use environment variables to manage sensitive configurations securely.

Restrict access to admin-level Lambda functions using role-based access control (RBAC).

Threats:

Tampering: Unauthorized modification of stored packages.

Information Disclosure: Leakage of S3 bucket names or sensitive package data.

Denial of Service (DoS): Overloading S3 requests or excessive file storage.
Mitigations:

Use S3 server-side encryption (SSE-S3 or SSE-KMS) to secure stored data.

Apply S3 bucket policies that restrict access to specific Lambda roles.

Enable S3 versioning to prevent accidental overwrites and excessive storage.

Threats (all for both arrows):

Spoofing: Unauthorized pushes or pipeline triggers by attackers.

Tampering: Injecting malicious code or configuration changes during the build or deployment process.

Repudiation: Developers or contributors denying submitting code changes or triggering deployments.

Mitigations:
Signed Commits: Require signed commits for all code contributions to ensure authenticity.

Automated Scans: Conduct dependency scans and vulnerability checks during CI/CD.

Threats:

Spoofing: Unauthorized pushes or triggers to the pipeline.

Tampering: Injecting malicious code during build or deployment.

Information Disclosure: Leaking access tokens or sensitive configurations in logs.

Mitigations:

Require signed commits for all code contributions.

Conduct automated dependency scans and vulnerability checks.

Mask sensitive values (e.g., tokens) in logs and restrict pipeline access.

Components:

- **Frontend: A React application styled with TailwindCSS that provides the user interface and interacts with the API Gateway via HTTPS for secure communication.**
- **API Gateway: Serves as the secure entry point for all API requests, validating and routing them to the appropriate Lambda functions.**
- **Lambda Functions: Handles backend logic for package management, including operations like retrieving metadata, accessing package files, and calculating costs.**
- **S3 and DynamoDB: S3 is used for secure package file storage, while DynamoDB stores metadata related to packages.**
- **GitHub and CodePipeline: Implements CI/CD for automated testing, building, and deployment to AWS Lambda and API Gateway.**

Trust Boundaries:

- **Frontend ↔ API Gateway: Secures API requests and prevents unauthorized access using HTTPS and JWT validation.**
- **API Gateway ↔ Lambda: Enforces input validation and schema checks to secure backend interactions.**

- **Lambda ↔ S3/DynamoDB: Protects data access with encryption (SSE-S3 or SSE-KMS) and strict IAM policies to limit unauthorized access.**
- **CI/CD Pipeline ↔ Deployment Targets: Mitigates risks of supply chain attacks by requiring signed commits, conducting dependency scans, and masking sensitive data in logs.**

---

7. Reflection and Future Considerations

Reflection

This threat modeling exercise provided critical insights into the potential risks within our package management system. By systematically applying STRIDE analysis to key trust boundaries, we were able to:

- Identify and mitigate security risks effectively.
- Enhance the system's reliability and maintainability.
- Ensure compliance with best practices for secure software development.

Our team's focus on **security by design**, such as enforcing role-based access control and validating all input, has significantly reduced the attack surface. Debugging tools like /last_run and comprehensive CloudWatch logs have improved both development efficiency and system monitoring.

Future Considerations

While our system currently adheres to robust security principles, future work could include:

1. **Advanced Logging**:
   - Use AWS CloudTrail to track detailed activity logs across all AWS services.
2. **Enhanced Rate-Limiting**:
   - Implement adaptive rate-limiting to adjust thresholds based on traffic patterns.
3. **Comprehensive Automated Testing**:
   - Integrate fuzz testing for endpoints like /regex to uncover edge-case vulnerabilities.
4. **Monitoring and Alerts**:
   - Set up AWS SNS for real-time alerts when anomalies are detected in API Gateway or Lambda metrics.
5. **Improved Token Handling**:

    - Use OAuth 2.0 for improved token-based authentication with granular permissions.

By addressing these future considerations, we can further enhance the security, scalability, and user experience of our system.