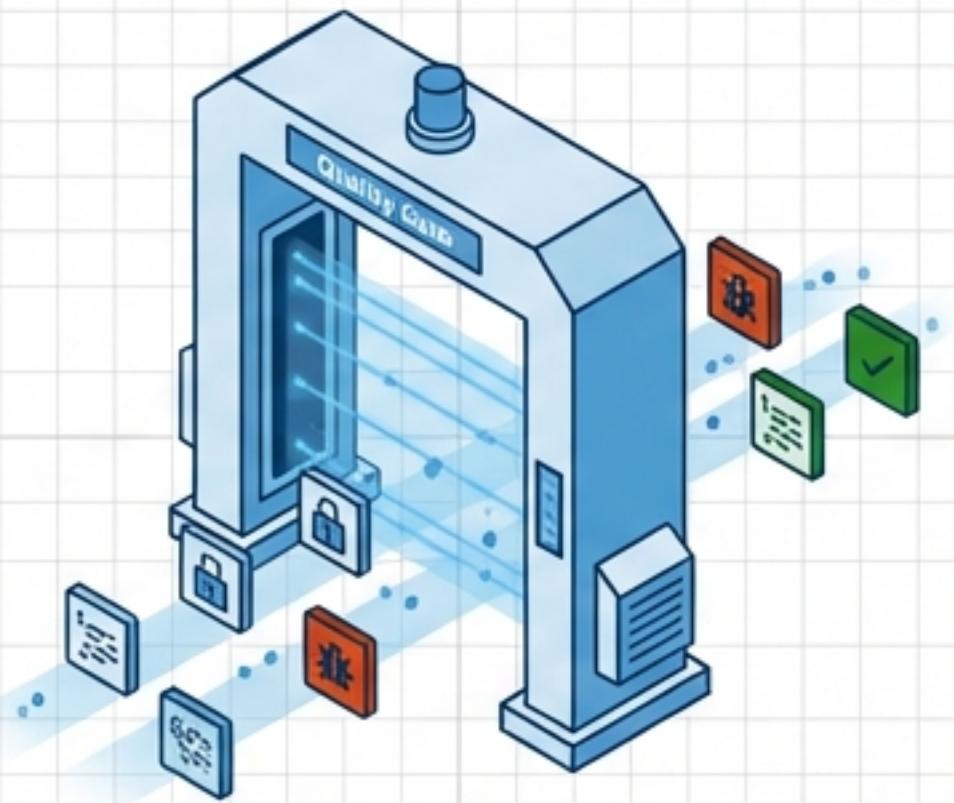


Azure DevOps Factory Blueprint: Security Addendum

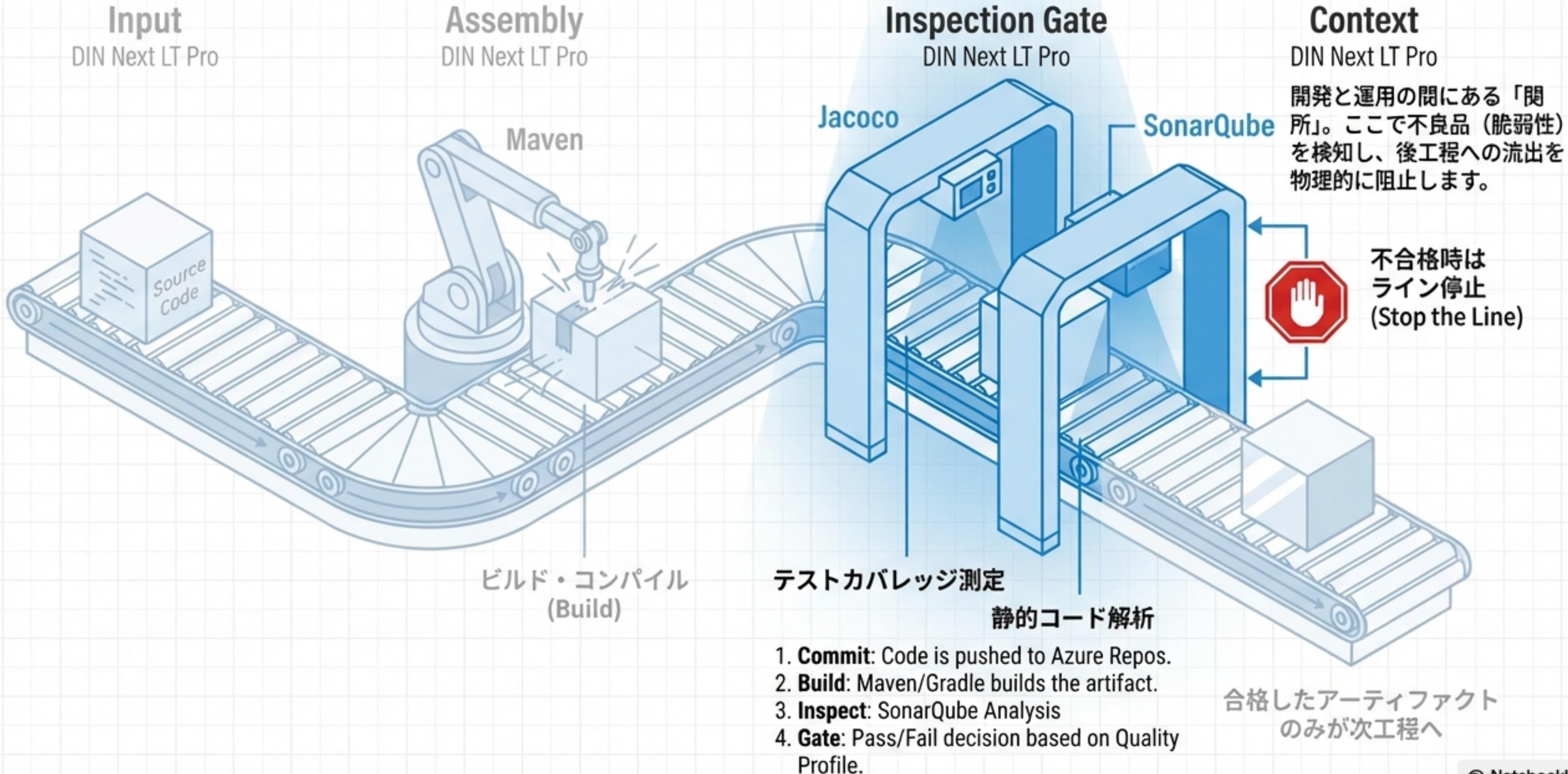
SonarQubeによる自動品質ゲートとSAST戦略



Theme: "The Invisible Inspector" (見えざる検査官)**

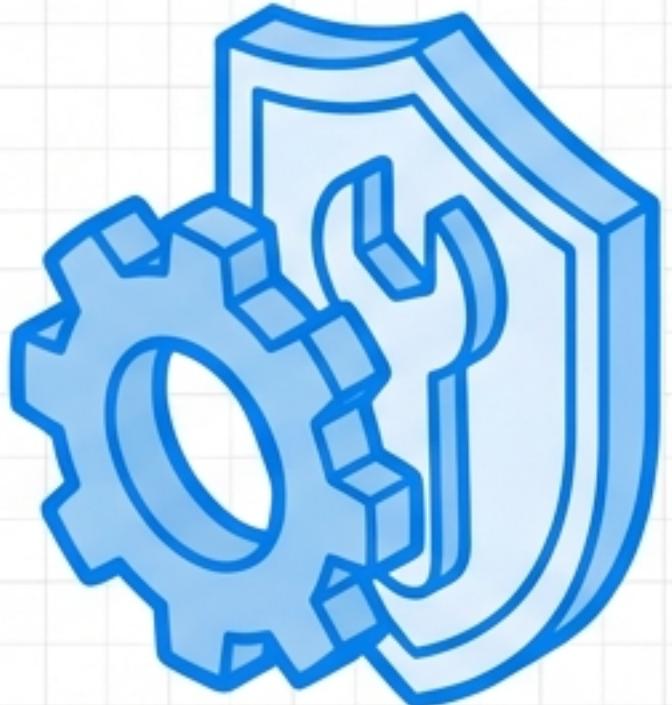
Summary: 本資料では、Factory Blueprintにおける「自動検査ゲート」の詳細仕様を定義します。静的解析（SAST）の中核となるSonarQubeのデータフロー解析（ティント解析）のメカニズムと、開発プロセスへの適用方法について解説します。

The Position of the Inspection Gate (検査ゲートの配置)



Three Pillars of Code Quality (コード品質を支える3つの柱)

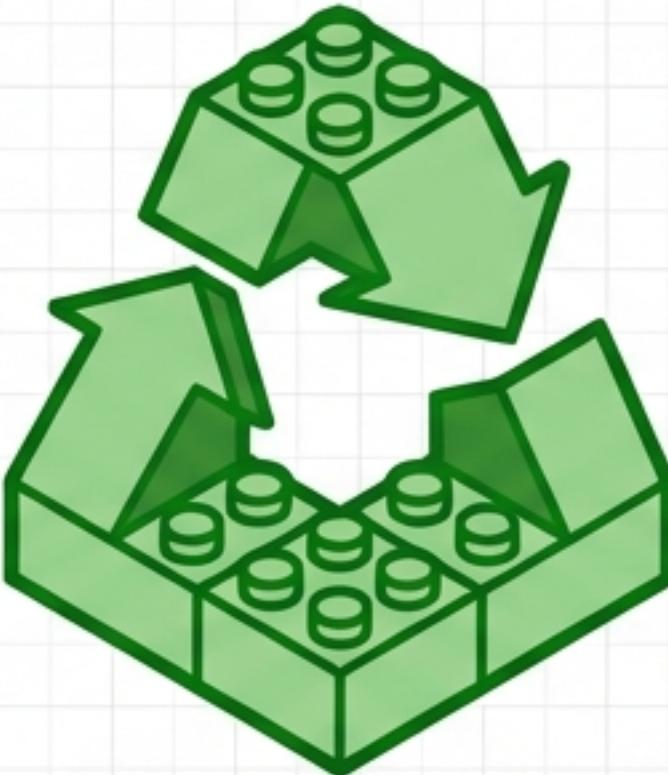
1. Reliability (信頼性 - Bug)



Will it crash? (アプリケーションがクラッシュしないか?)

Examples: Null Pointer Dereference, Resource leaks.

2. Maintainability (保守性 - Code Smell)



Can we update it easily? (将来の改修が容易か?)

Examples: Duplicated code, overly complex logic (Cyclomatic Complexity).

3. Security (安全性 - Vulnerability)

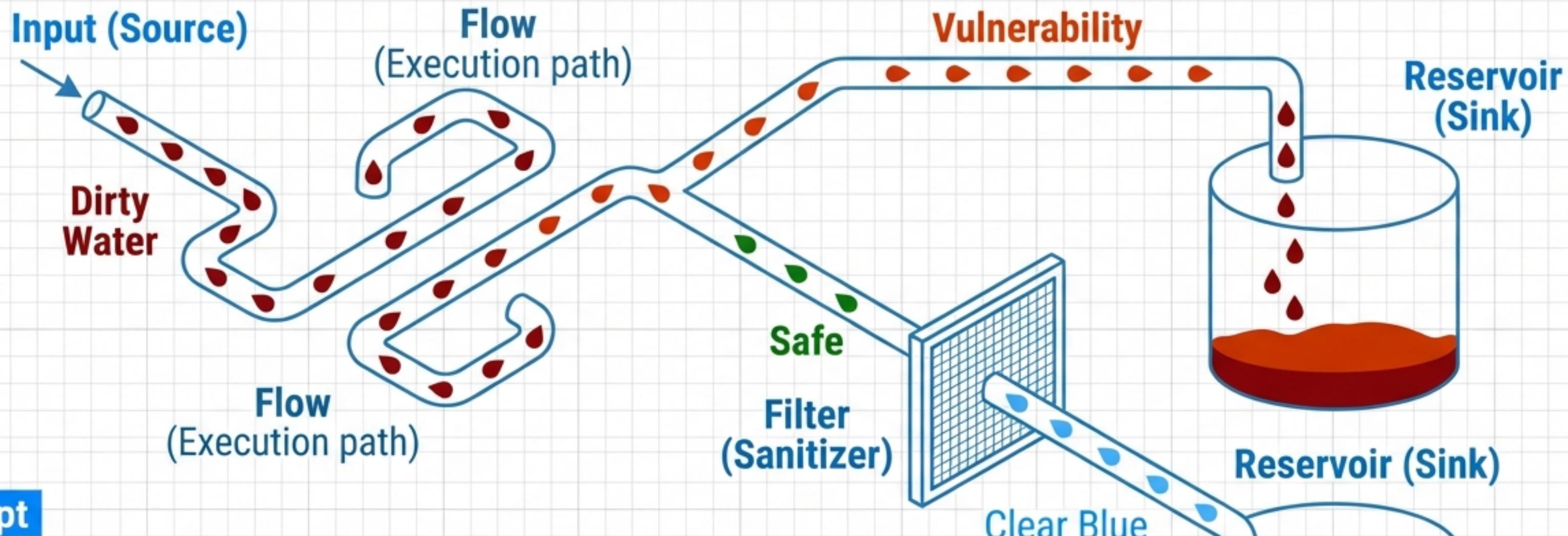


Can it be hacked? (攻撃されるリスクはないか?)

Focus: This is the domain of SAST (Static Application Security Testing).

Examples: Injection attacks, broken authentication.

Mechanism: What is Taint Analysis? (ティント解析のメカニズム)



Concept

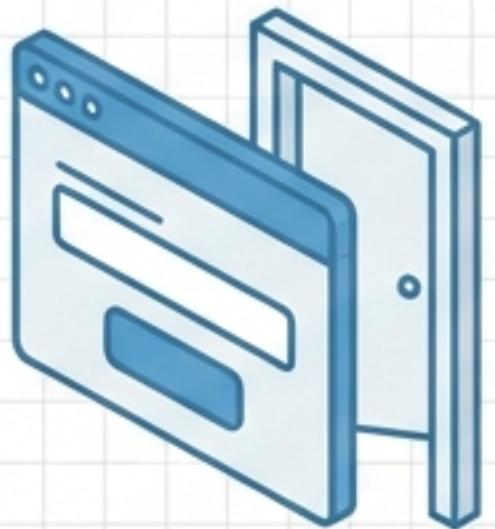
ユーザーからの入力データ (Source) が、無害化 (Sanitizer) されずに危険な箇所 (Sink) に到達するかを追跡する技術です。

Why it's needed

単純なテキスト検索 (Grep) では発見できない、ファイルやメソッドを跨いだデータの流れを解析します。

The Anatomy of a Vulnerability (脆弱性の構造 : Source, Flow, Sink)

1. Source (入口)

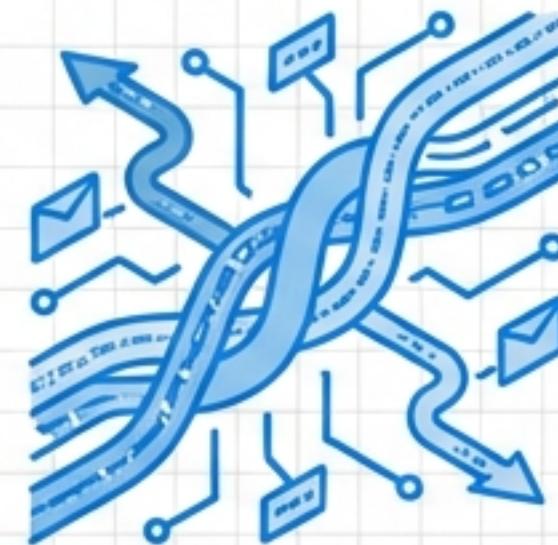


SonarQube Blue (#4E9BCD)

外部からの入力点。攻撃者が悪意
あるデータを注入できる場所。

Examples: HttpServletRequest.getParameter(), API headers, Form inputs.

2. Flow (伝播パス)



Azure Blue (#007FFF)

データがアプリケーション内でコ
ピー、変換、受け渡しされる経路。

Capability: SonarQube performs Cross-file analysis (ファイル間解析) to track data even when it moves between different classes.

3. Sink (出口/危険箇所)

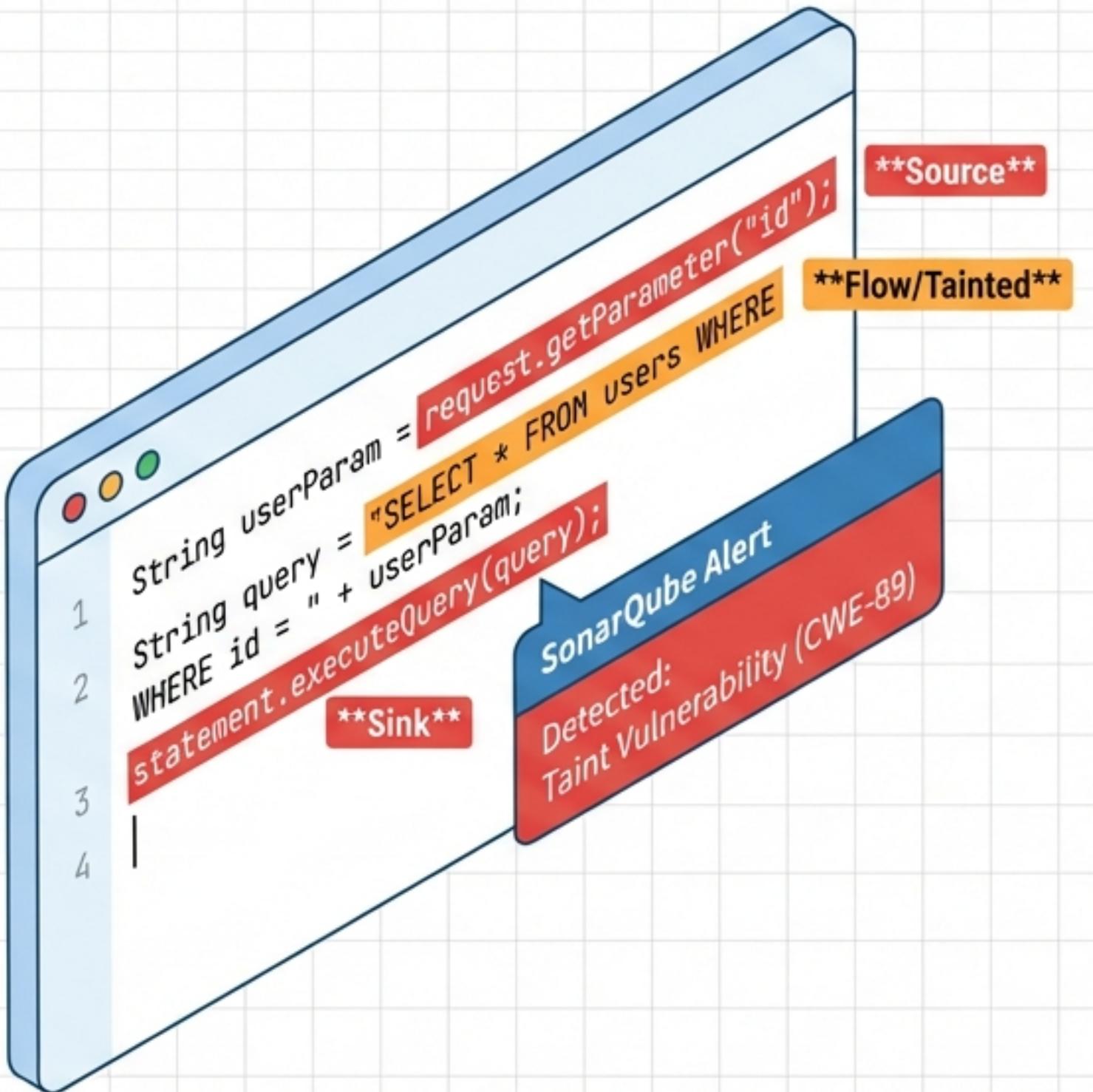


Warning Red (#D83B01)

データが実行される場所。ここで
攻撃が成立する。

Examples: executeQuery() (SQL),
Runtime.exec() (Command), HTML output
(XSS).

Risk Example: SQL Injection (具体例：SQLインジェクションのリスク)



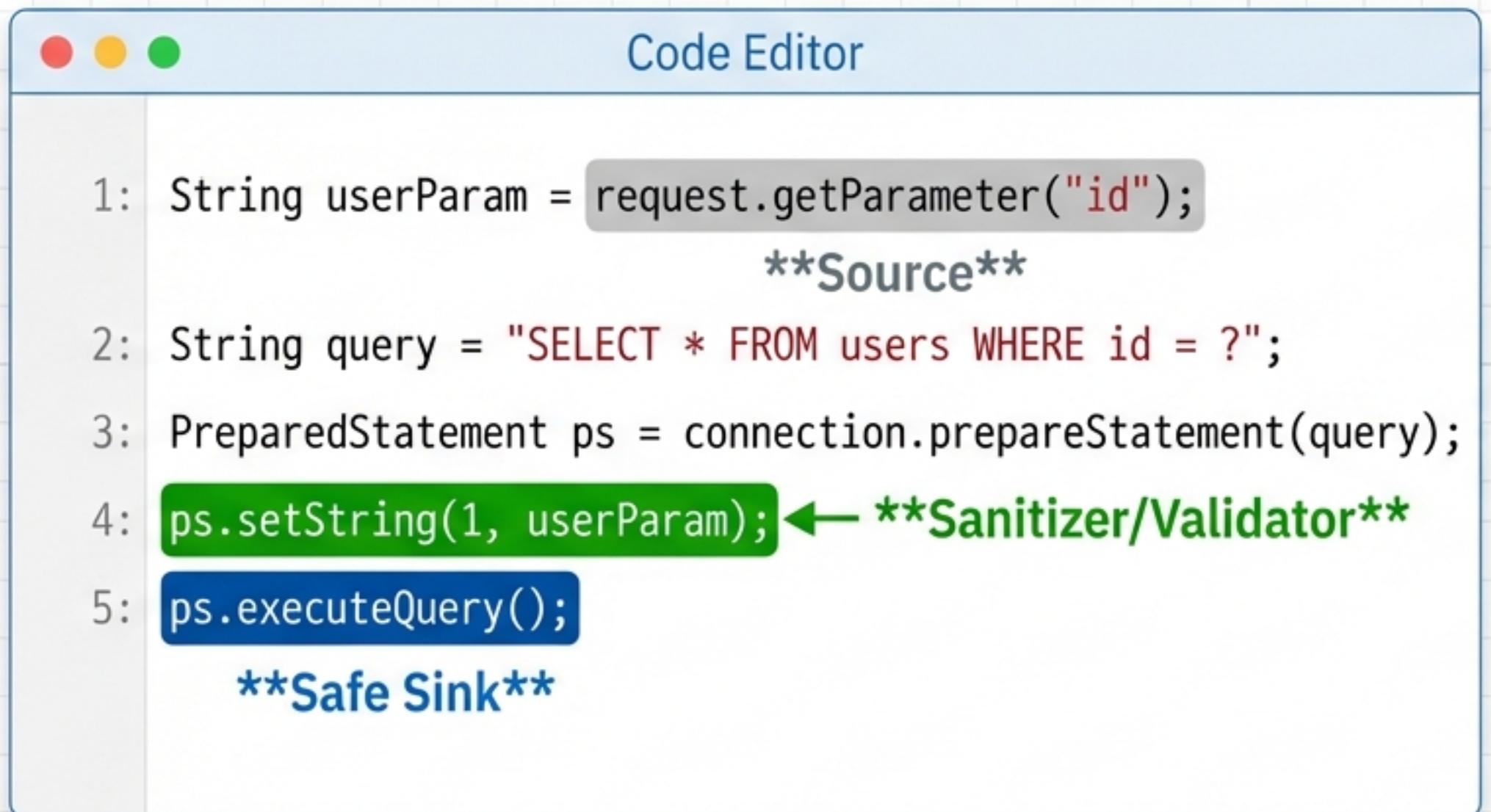
Diagnosis

外部からの入力 (userParam) が検証されずに、直接データベースへの命令文 (query) に結合されています。

Impact

攻撃者が任意のSQLコマンドを実行し、データの盗難や改ざんが可能になります。

Remedy: Sanitization & Validation (対処例：サニタイズと無害化)



The code editor window shows the following Java code:

```
1: String userParam = request.getParameter("id");
   **Source**
2: String query = "SELECT * FROM users WHERE id = ?";
3: PreparedStatement ps = connection.prepareStatement(query);
4: ps.setString(1, userParam); ← **Sanitizer/Validator**
5: ps.executeQuery();
   **Safe Sink**
```

The code demonstrates a fix for SQL injection. It reads a parameter from a request, stores it in a variable, and then uses it in a query string. The fix involves using a PreparedStatement and setting the parameter value using the setString method, which is highlighted with a green background and labeled as a 'Sanitizer/Validator'. The resulting code is considered a 'Safe Sink'.

The Fix

プレースホルダ (?) を使用した「プリペアトドステートメント」を利用。

SonarQube's Logic

解析エンジンは setString() メソッドがデータを安全に処理（エスケープ）することを認識しています。SourceとSinkの間のパスが「切斷」されたと判断し、警告を解除します。

Vulnerabilities vs. Security Hotspots (脆弱性とセキュリティ・ホットスポット)

Vulnerability (脆弱性)



Definition:

攻撃が可能であることが確実な欠陥。

Action: Fix Immediately. (即時修正)

Role: Stop the Line.

SonarQubeは単にエラーを指摘するだけでなく、開発者に「安全なコードの書き方」を学習させるツールです。

Security Hotspot (セキュリティ・ホットスポット)



Definition:

セキュリティ上「敏感」なコード（暗号化処理、権限確認など）。必ずしも危険ではないが、人間による確認が必要。

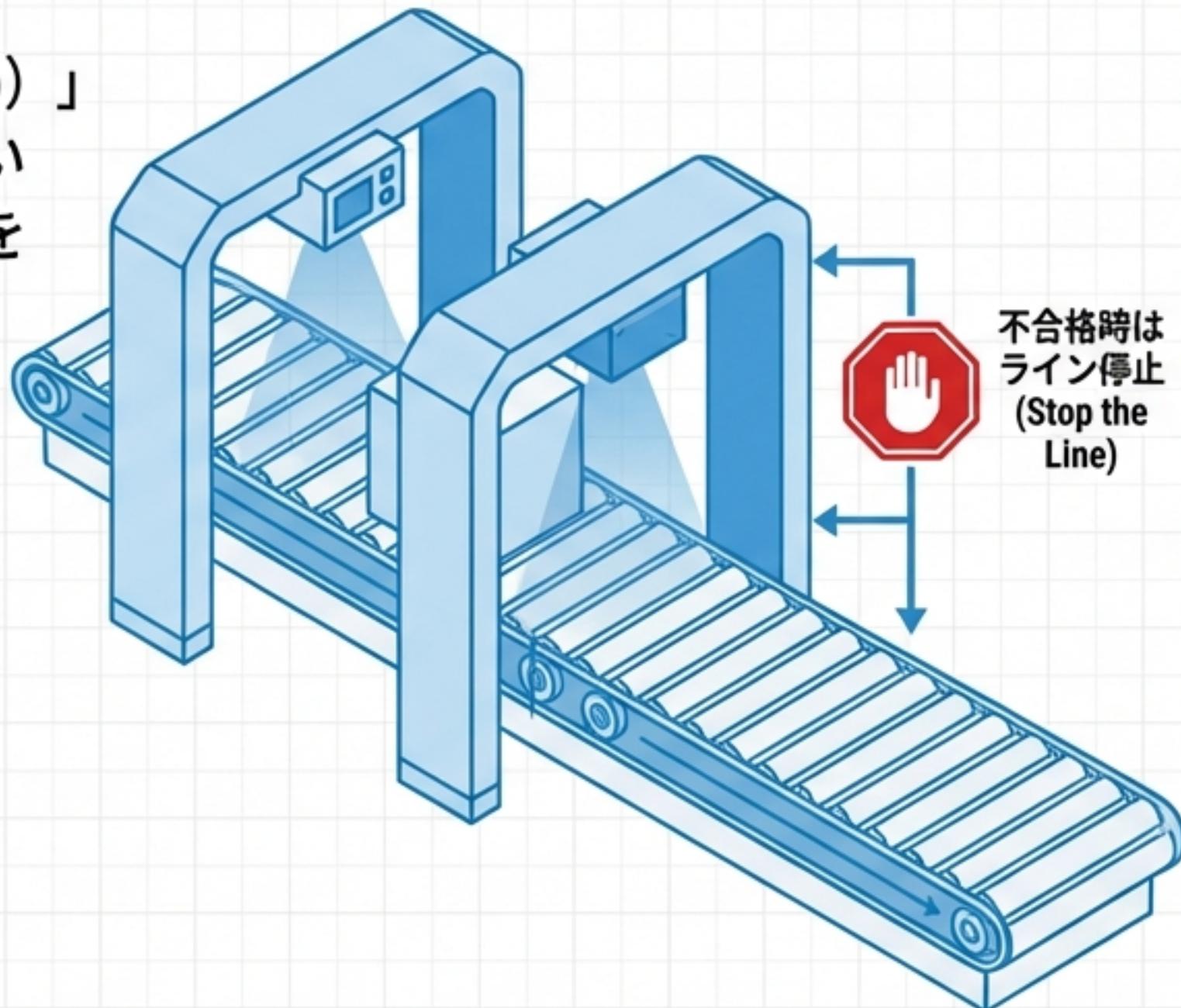
Action: Review. (開発者によるレビュー)

Value: 教育的效果。開発者のセキュリティ意識 (Security IQ) を向上させる。

Enforcement: The 'Stop the Line' Policy (品質ゲートによる強制力)

The Rule:

「品質ゲート (Quality Gate)」を設定し、基準を満たさないコードのパイプライン通過を阻止します。



Policy Card

Recommended Threshold (推奨設定)

New Code (新規コード):

- Critical / Major Vulnerabilities: 0 (ゼロであること)
- Security Hotspots Reviewed: 100%

Outcome: 脆弱性が含まれるアーティファクトは生成されません (Registryに登録されない)。これにより、「既知の脆弱性」を本番環境に持ち込むリスクを完全に排除します。