Python Secure Code Risk Analyzer

IDEO-LAB

JANUARY 30TH 2026

AGENDA GENERAL

- 1) Vision produit
- 2) Plan de projet (phases)
- 3) Analyse Fonctionnelle (AF)
- 4) Roadmap “règles MVP” (haut impact / faciles)
- 5) Livrables attendus (concrets)

1 VISION PRODUIT

1) Vision produit

Objectif

Un outil qui analyse un projet Python (ou un fichier), détecte les **risques sécurité**, **qualité**, **conformité**, et produit un rapport :

- **Compréhensible** (pédagogique)
- **Actionnable** (patchs/solutions proposées)
- **Fiable** (faible taux de faux positifs)
- **Industrializable** (CI/CD, Git hooks, dashboard)

Public cible

- Devs Python (junior → senior)
- Lead dev / architectes
- DevSecOps / sécurité applicative
- Audits internes / conformité

Positionnement

Plus didactique qu'un "linter", plus orienté *risque* qu'un simple scanner :

- "**Pourquoi c'est dangereux**"
- "**Comment c'est exploité**" (exemples contrôlés)
- "**Comment corriger proprement**" (fix recommandé + alternatives)
- "**Quel niveau de criticité**" + priorisation

2 PLAN DE PROJET

AGENDA PLAN DE PROJET

- Phase 0 — Cadrage (fondations)
- Phase 1 — MVP SAST (statique AST)
- Phase 2 — Analyse avancée (flux / taint)
- Phase 3 — Écosystème (dépendances / config)
- Phase 4 — UX & Éducation
- Phase 5 — Industrialisation

Phase 3 — Écosystème (dépendances / config)

1. Analyse dépendances (requirements, poetry, pipenv)
2. Détection libs vulnérables (DB CVE)
3. Analyse config (Django/Flask/FastAPI, settings, headers, CORS, DEBUG)

Phase 4 — UX & Éducation

1. Rapports HTML interactifs (tri, filtres, diff entre scans)
2. "Learning mode" : mini cours par finding
3. "Fix mode" : PR patch (optionnel) / suggestions

Phase 5 — Industrialisation

1. Intégrations CI (GitHub Actions/GitLab CI)
2. Baselines + suppression contrôlée (suppressions signées)
3. Historique / scoring / alerting

2) Plan de projet (phases)

Phase 0 — Cadrage (fondations)

1. Définir le **périmètre sécurité** (OWASP-like adapté Python)
2. Définir un **format findings** standardisé (JSON + Markdown + HTML)
3. Définir une **taxonomie** (catégories, sévérité, confiance, CWE)

Phase 1 — MVP SAST (statique AST)

1. Parser Python (AST)
2. Premier set de règles "haut rendement" :
   - injections (SQL / shell / template)
   - secrets hardcodés
   - crypto faible
   - désérialisation dangereuse
   - SSRF / path traversal
3. Rapport local (CLI) + export JSON

Phase 2 — Analyse avancée (flux / taint)

1. Propagation "données non fiables" → "sinks dangereux"
2. Réduction faux positifs via contexte
3. Auto-suggestions plus intelligentes (patch guidé)

3 ANALYSE FONCTIONNELLE AF

AGENDA ANALYSE FONCTIONNELLE

- 3.1 Acteurs
- 3.2 Cas d’utilisation (Use Cases)
- 3.3 Périmètre fonctionnel (ce que l’outil analyse)
- 3.4 Sorties (Deliverables du système)
- 3.5 Exigences non fonctionnelles
- 3.6 Moteur de règles (fonctionnellement)
- 3.7 Paramétrage / profils

3) Analyse Fonctionnelle (AF)

3.1 Acteurs

- **Développeur** : lance une analyse locale, corrige.
- **Lead/Reviewer** : consulte rapport, impose qualité, valide exceptions.
- **CI/CD** : bloque/autorise merge selon seuils.
- **Admin Sécurité** : définit politiques, règles, niveaux d'exigence.

3.2 Cas d'utilisation (Use Cases)

UC1 — Scanner un projet

Entrée : chemin dossier projet

Sortie : findings + score + rapport HTML/JSON

Options :

- niveaux (fast/standard/deep)
- ciblage (fichiers, modules)
- profils (web/django/api/cli/data)

UC2 — Scanner un fichier / snippet

Entrée : fichier unique

Sortie : findings ciblés + correction proposée

UC3 — Générer un rapport pédagogique

Sortie :

- explication
- impact
- preuve (snippet + localisation)
- "comment corriger"
- liens doc (CWE/OWASP/Doc Python) *(optionnel)*

UC4 — Mode CI "gating"

Entrée : findings JSON

Règle : fail si "critical>=1" ou "high>=N"

Sortie : exit code + résumé court

UC5 — Baseline / Diff

Comparer scan A vs scan B :

- nouvelles failles
- failles résolues
- régression de score

UC6 — Suppressions contrôlées

Permettre de marquer une finding comme :

- faux positif
- risque accepté (avec justification + durée + owner)

3.3 Périmètre fonctionnel (ce que l'outil analyse)

A) Code Python (AST / patterns / taint)

- appels dangereux : `eval`, `exec`, `pickle`, `subprocess`, `os.system`, etc.
- injections : SQL / shell / template
- validation input manquante / types inattendus
- path traversal / accès fichiers
- SSRF (requêtes vers URL contrôlée)
- crypto faible / mauvais hasard / mauvais TLS
- auth : comparaisons non constantes, tokens stockés en clair
- logs : fuites (PII, secrets)

B) Frameworks & web

- **Django** : `DEBUG`, `ALLOWED_HOSTS`, `CSRF`, `SECURE_*`, `SECRET_KEY`, sessions, headers
- **Flask/FastAPI** : CORS, debug, dépendances, validation schémas
- Contrôles HTTP : headers sécurité, cookies, redirections

C) Dépendances & supply chain

- libs à risques / versions connues vulnérables
- typosquatting (optionnel)
- dépendances non pinées / hash manquants

D) Secrets

- clés API, tokens, passwords
- patterns + entropie + contexte
- exclusion de faux positifs (fixtures/tests/README)

3.4 Sorties (Deliverables du système)

1) Findings normalisés

Chaque finding contient :

- `id`, `rule_id`, `category`, `severity`, `confidence`
- fichier, ligne, colonne, extrait
- “trace” (si taint)
- explication pédagogique (texte court + long)
- recommandation (fix)
- ref CWE + tags

2) Score et priorisation

- score global (0–100)
- score par catégorie
- top 10 risques (impact x probabilité)
- “quick wins” (fix rapides)

3) Rapports

- **Console** : résumé
- **JSON** : machine-readable
- **HTML** : lecture “auditeur” + navigation + filtres
- (option) **SARIF** : intégration plateformes

3.5 Exigences non fonctionnelles

Performance

- mode fast (<X sec sur projet moyen)
- cache d'analyse incrémentale (hash fichiers)

Qualité

- réduction faux positifs
- niveau de confiance affiché
- règles testées (jeu de samples)

Sécurité de l'outil

- ne jamais exécuter le code scanné
- sandbox si analyse "dynamique" future

Extensibilité

- moteur de règles "plug-in"
- règles en Python déclaratives (pattern + metadata)

3.6 Moteur de règles (fonctionnellement)

Types de règles

1. **Pattern AST** (simple et rapide)
2. **Dataflow / taint** (source → propagation → sink)
3. **Config rules** (settings, YAML, env, docker)
4. **Dependency rules** (version audit)

Cycle d'une règle

- match → enrichissement contexte → scoring → suggestion fix → rendu pédagogique

---

3.7 Paramétrage / profils

- `--profile web|django|api|cli|data`
- `--severity-threshold`
- `--exclude tests,migrations,venv`
- `--baseline baseline.json`
- `--format json,html,sarif`

4 ROADMAP REGLES MVP

4) Roadmap “règles MVP” (haut impact / faciles)

Pour démarrer très fort dès la V1 :

- `eval/exec` sur input
- `pickle` /désérialisation non sûre
- `subprocess` avec `shell=True` et concat
- SQL concat (même simple heuristique)
- secrets dans code
- `yaml.load` non safe
- `random` pour tokens / sécurité
- `ssl` /TLS vérification désactivée
- path traversal (join avec input non normalisé)
- DEBUG activé (framework)

---

5) Livrables attendus (concrets)

- CLI `py-risk-analyzer`
- Rapport HTML interactif
- JSON findings
- Bibliothèque “rules pack”
- Jeu de tests (projets vulnérables / safe)
- Guide “comment corriger” (base knowledge)