03 — Sécurité Blockchain (Trail of Bits)

Scanners de vulnérabilités blockchain (Solana, Substrate, TON, Cosmos, Cairo, Algorand).

□□ Index — 03 — Sécurité Blockchain (Trail of Bits)

03 — Sécurité Blockchain (Trail of Bits)

> Index des skills regroupés dans ce livre. Clique sur un skill pour ouvrir sa fiche.

11 skills documentés.

Skill	En une phrase
algorand-vulnerability-scanner	Scanne les smart contracts Algorand (écrits en TEAL ou PyTeal) pour repérer 11 types de failles classiques sur cette chaîne, dont…
audit-prep-assistant	T'aide à préparer ton codebase pour un audit de sécurité externe en suivant la checklist Trail of Bits : objectifs clairs,…
cairo-vulnerability-scanner	Scanne les smart contracts Cairo (le langage des dApps StarkNet, le "L2" de validité sur Ethereum) pour repérer 6 types de…
code-maturity-assessor	Évalue le "niveau de maturité" d'un codebase blockchain selon une grille Trail of Bits à 9 catégories (arithmétique, monitoring,…
cosmos-vulnerability-scanner	Scanne les modules d'une blockchain Cosmos SDK (en Go) ou les smart contracts CosmWasm (en Rust) pour repérer 9 types de failles…
guidelines-advisor	Conseille un projet de smart contracts (surtout Solidity) selon les "Development Guidelines" Trail of Bits : génère la doc,…
secure-workflow-guide	Pilote ton développement de smart contracts via un workflow sécurité Trail of Bits en 5 étapes (Slither, features spéciales,…

Skill	En une phrase
solana-vulnerability-scanner	Scanne automatiquement le code d'un programme Solana pour repérer 6 types de failles de sécurité critiques propres à cette...
substrate-vulnerability-scanner	Scanne le code d'une blockchain construite avec Substrate (la techno derrière Polkadot et ses "parachains") pour repérer 7 types...
token-integration-analyzer	Analyse soit le contrat d'un token (ERC20/ERC721) pour vérifier sa conformité, soit un protocole qui intègre des tokens externes...
ton-vulnerability-scanner	Scanne les smart contracts TON (The Open Network, la blockchain de Telegram) écrits en FunC pour repérer 3 types de failles...

substrate-vulnerability-scanner

substrate-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne le code d'une blockchain construite avec Substrate (la techno derrière Polkadot et ses "parachains") pour repérer 7 types de failles critiques qui peuvent faire crasher un nœud, ouvrir des attaques de spam ou bypasser l'admin.

Quand l'utiliser

- Quand un projet utilise Substrate ou FRAME (le framework pour fabriquer une blockchain "à la Polkadot") et que tu veux faire un check sécurité.
- Avant de lancer une parachain Polkadot/Kusama sur le réseau principal.
- Quand un dev a écrit un "pallet" custom (un module qui ajoute des fonctionnalités à la chaîne) et que tu veux le faire relire.
- Pour vérifier que les calculs de "poids" (le coût d'une transaction) sont corrects — un mauvais poids ouvre des attaques de spam.
- Si tu auditeras des fonctions privilégiées (qui ne devraient être appelables que par root/admin).

Comment l'invoquer

- **Slash command** : `/substrate-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my Substrate pallet" / "check FRAME runtime" / "scan Polkadot parachain"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

Substrate est le framework Rust de Parity pour construire des blockchains personnalisées (notamment les parachains Polkadot). Le code est organisé en "pallets" (modules) qui sont assemblés dans un runtime. Les bugs Substrate sont souvent catastrophiques : un panic dans un pallet peut faire planter tous les nœuds du réseau, et une mauvaise vérification d'origine peut donner les droits admin à n'importe qui.

Ce skill cherche 7 patterns critiques : 1) **Arithmetic Overflow** (utiliser `+`, `-`, `*` directement au lieu de `checked_*` — les calculs débordent silencieusement en mode release), 2) **Don't Panic** (un `unwrap()` qui plante = nœud HS = blockchain HS), 3) **Weights & Fees mal calibrés** (ouvre des attaques DoS), 4) **Verify First, Write Last** (écrire en storage avant de valider — sur d'anciennes versions, l'écriture persiste même si la validation échoue), 5) **Unsigned Transaction Validation** (transactions sans signataire mal protégées contre le replay), 6) **Bad Randomness** (utiliser un random prédictible/manipulable), 7) **Bad Origin** (utiliser `ensure_signed` au lieu de `ensure_root` sur une fonction admin — n'importe qui peut l'appeler).

Workflow : tu pointes ton dossier `pallets/`, le skill scanne, te liste les findings avec sévérité et te propose les fix Rust.

Pour aller plus loin

Pour les détails techniques (patterns exacts, code Rust avant/après, intégration `cargo-fuzz` et `try-runtime`), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/substrate-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `substrate-vulnerability-scanner`

- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/substrate-vulnerability-scanner/SKILL.md`

cosmos-vulnerability-scanner

cosmos-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne les modules d'une blockchain Cosmos SDK (en Go) ou les smart contracts CosmWasm (en Rust) pour repérer 9 types de failles critiques qui peuvent halter la chaîne, casser le consensus ou faire perdre des fonds.

Quand l'utiliser

- Quand tu touches à une blockchain de l'écosystème Cosmos (Cosmos Hub, Osmosis, Injective, etc.) ou à un contrat CosmWasm.
- Avant de pousser un module custom (`x/monmodule/`) en prod.
- Pour vérifier qu'un contrat CosmWasm valide bien les "denoms" (= les noms de tokens — accepter un token bidon est une attaque classique).
- Quand tu intègres l'IBC (Inter-Blockchain Communication, le pont entre chaînes Cosmos) — des erreurs IBC peuvent vider des pools.
- Pour enquêter sur un crash de chaîne ("chain halt") après un upgrade.

Comment l'invoquer

- **Slash command** : `/cosmos-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my Cosmos module" / "check CosmWasm contract" / "scan IBC handler"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

Cosmos est un écosystème de blockchains interopérables, où chaque chaîne est faite de modules Go (`x/banking`, `x/staking`...) ou héberge des contrats CosmWasm écrits en Rust. La grande particularité : si un module produit un résultat différent sur deux nœuds (= "non-déterminisme"), la chaîne s'arrête. Un simple `time.Now()` ou un map Go non trié peut tout faire planter.

Ce skill cherche 9 patterns : 1) **Missing Denom Validation** (accepter n'importe quel token), 2) **Insufficient Authorization** (oublier de vérifier qui envoie le message), 3) **Missing Balance Check**, 4) **Improper Reply Handling** (mal gérer les réponses de sous-messages), 5) **Missing Reply ID Check**, 6) **Improper IBC Packet Validation** (paquets cross-chain non validés — fuites de fonds), 7) **Unvalidated Execute Message**, 8) **Integer Overflow**, 9) **Reentrancy via Submessages** (modifier l'état avant un appel externe — réentrance possible).

Il cible aussi les méthodes ABCI (BeginBlocker, EndBlocker, CheckTx, DeliverTx) qui sont consensus-critiques, et utilise CodeQL si dispo pour traquer le non-déterminisme.

Pour aller plus loin

Pour les détails techniques (exemples Go et Rust, patterns IBC, integration CodeQL), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/cosmos-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `cosmos-vulnerability-scanner`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/cosmos-vulnerability-scanner/SKILL.md`

token-integration-analyzer

token-integration-analyzer

> *Catalogue généré le 2026-05-11*

En une phrase

Analyse soit le contrat d'un token (ERC20/ERC721) pour vérifier sa conformité, soit un protocole qui intègre des tokens externes pour vérifier qu'il gère bien les ~24 "tokens bizarres" qui existent (USDT sans return value, fee-on-transfer, rebasing, etc.).

Quand l'utiliser

- Quand tu construis un token (ERC20 ou NFT ERC721) et que tu veux t'assurer qu'il respecte le standard.
- Quand tu construis un protocole DeFi qui **accepte des tokens arbitraires** (un DEX, un lending market, un yield aggregator) — c'est là que ça devient critique : un token bizarre peut casser ta logique.
- Pour analyser les "weird ERC20 patterns" : USDT qui ne renvoie pas de boolean sur `transfer`, tokens à frais (PAXG), tokens rebasing (Ampleforth), tokens pausables (BNB), tokens upgradeables (USDC), tokens avec hooks (ERC777 → réentrance), etc.
- Pour faire une analyse on-chain : distribution des holders, supply, présence sur exchanges, risques de flash mint.
- Quand tu hésites sur la nécessité d'utiliser `SafeERC20` ou un wrapper défensif.

Comment l'invoquer

- **Slash command** : `/token-integration-analyzer`

- **Phrases déclencheurs (texte)** : "analyze token integration" / "check ERC20 conformity" / "audit weird token handling"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le grand piège du DeFi : tous les "ERC20" ne se comportent pas comme le standard le dit. USDT ne retourne pas de booléen sur `transfer` (les protocoles naïfs cassent). PAXG prend des frais à chaque transfert (tu envoies 100, le destinataire reçoit 98). Ampleforth modifie les balances sans émettre d'event. ERC777 a des hooks qui permettent la réentrance. Si ton protocole ne sait pas gérer ces cas, il perd des fonds.

Ce skill, basé sur la "Token Integration Checklist" + "Weird ERC20 Database" de Trail of Bits, fait deux choses selon le contexte. **Si tu écris un token** : il vérifie la conformité ERC20/ERC721 (return values, decimals, métadonnées, race conditions sur approve), liste les privilèges du owner (mint, pause, blacklist), analyse la complexité du contrat. **Si ton protocole intègre des tokens externes** : il check les 10 catégories d'assessment dont les fameux ~24 "weird patterns" — fee-on-transfer, rebasing, missing return values, hooks de réentrance, balances modifiées hors transfer, low/high decimals, code injection via le nom du token, etc.

Pour Solidity, il s'appuie sur `slither-check-erc`, `slither --print human-summary` et `slither-prop`. Pour les contrats déjà déployés, il peut aussi faire de l'analyse on-chain (supply, distribution, exchange listings).

Pour aller plus loin

Pour la liste complète des 24+ weird ERC20 patterns avec exemples et mitigations, consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/token-integration-analyzer/SKILL.md` et la ressource `resources/ASSESSMENT_CATEGORIES.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `token-integration-analyzer`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/token-integration-analyzer/SKILL.md`

code-maturity-assessor

code-maturity-assessor

> *Catalogue généré le 2026-05-11*

En une phrase

Évalue le "niveau de maturité" d'un codebase blockchain selon une grille Trail of Bits à 9 catégories (arithmétique, monitoring, contrôles d'accès, complexité, décentralisation, doc, MEV, low-level, tests) et te livre une note `Missing / Weak / Moderate / Satisfactory / Strong` par catégorie avec recommandations.

Quand l'utiliser

- Pour faire un **état des lieux objectif** d'un projet blockchain — pas un bug hunt, mais une note globale.
- Quand un investisseur, un partenaire ou toi-même veut savoir si un protocole est "prod-ready" ou encore en alpha.
- Avant un audit, pour identifier les chantiers structurels (= au-delà des bugs : est-ce que la gouvernance est trop centralisée ? Est-ce que la doc existe ?).
- Quand tu hésites entre 2 projets/forks et que tu veux comparer leur sérieux technique.
- Pour bâtir une roadmap d'amélioration priorisée.

Comment l'invoquer

- **Slash command** : `/code-maturity-assessor`
- **Phrases déclencheurs (texte)** : "assess code maturity" / "maturity scorecard" / "evaluate codebase quality"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Trail of Bits a formalisé une grille de "Code Maturity Evaluation v0.1.0" qui regarde un projet sous 9 angles, et c'est ce que ce skill applique de façon systématique. Contrairement aux scanners (qui cherchent des bugs précis), ici on évalue des **pratiques** : "est-ce qu'il y a des events bien définis pour le monitoring ?", "est-ce que les rôles admin sont bien séparés ?", "est-ce que la doc inline existe ?", etc.

Les 9 catégories : 1) **Arithmétique** (protection contre les overflows, gestion de la précision), 2) **Auditing** (events, monitoring, incident response), 3) **Authentification / Access Controls** (gestion des privilèges, multisig), 4) **Complexity Management** (taille des fonctions, profondeur d'héritage), 5) **Decentralization** (risques de centralisation, timelocks, opt-out utilisateur), 6) **Documentation** (specs, NatSpec, glossaire), 7) **Transaction Ordering Risks / MEV**, 8) **Low-Level Code** (assembleur, optimisations risquées), 9) **Testing** (coverage, fuzzing, tests d'invariants).

Workflow en 3 phases : **Discovery** (le skill scanne ton projet), **Analysis** (il vérifie chaque catégorie, te pose des questions sur les processus qu'il ne voit pas dans le code), **Report** (une scorecard avec ratings + roadmap priorisée).

Pour aller plus loin

Pour la grille complète avec les seuils de notation détaillés, consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/code-maturity-assessor/SKILL.md` et la ressource `resources/ASSESSMENT_CRITERIA.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `code-maturity-assessor`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/code-maturity-assessor/SKILL.md`

guidelines-advisor

guidelines-advisor

> *Catalogue généré le 2026-05-11*

En une phrase

Conseille un projet de smart contracts (surtout Solidity) selon les "Development Guidelines" Trail of Bits : génère la doc, revoit l'architecture, audite les patterns d'upgrade et proxy, évalue les tests et les dépendances.

Quand l'utiliser

- En **début de projet** pour cadrer l'architecture avant d'avoir trop de dette technique.
- Quand tu hérites d'un codebase blockchain et que tu veux une vue d'ensemble structurée.
- Pour générer la documentation manquante : description en français clair, diagrammes d'architecture, NatSpec (le format de commentaires standardisé Solidity).
- Quand le projet utilise des **proxies upgradeables** (un pattern complexe et dangereux où le contrat peut être modifié après déploiement) et que tu veux faire vérifier la conformité.
- Pour challenger les choix on-chain vs off-chain (= "qu'est-ce qui doit vraiment vivre sur la blockchain et qu'est-ce qu'on peut faire ailleurs ?").

Comment l'invoquer

- **Slash command** : `/guidelines-advisor`
- **Phrases déclencheurs (texte)** : "review my smart contract architecture" / "check best practices" / "audit upgradeability"
- **Auto-invocation** : Sur demande explicite (souvent au début d'un projet ou lors d'une refonte).

Description détaillée

Trail of Bits maintient un guide "Building Secure Contracts — Development Guidelines" qui condense des années d'audits en bonnes pratiques. Ce skill applique ce guide à ton projet de façon interactive et te livre des recommandations actionnables.

Il couvre 11 zones d'analyse : 1) **Documentation & Specifications** (descriptions plain-English, diagrammes, NatSpec), 2) **On-Chain vs Off-Chain Computation** (quoi mettre où), 3) **Upgradeability** (faut-il vraiment de l'upgradeable ? migration vs upgrade, séparation données/logique), 4) **Delegatecall Proxy Pattern** (cohérence du storage layout, init, function shadowing — sources classiques de bugs critiques), 5) **Function Composition** (taille, clarté, modularité), 6) **Inheritance** (profondeur, diamond problem), 7) **Events Logging**, 8) **Common Pitfalls**, 9) **Dependencies**, 10) **Testing Coverage** et 11) techniques de test avancées.

Workflow en 5 phases : Discovery → Documentation Generation (génère ce qui manque) → Architecture Analysis → Implementation Review → Recommandations priorisées. Pour Solidity, il s'appuie sur les "Slither printers" (les générateurs de diagrammes intégrés à Slither, l'analyseur statique Trail of Bits).

C'est un skill **stratégique** plutôt qu'un scanner : il ne te dit pas "ligne 42, bug X", il te dit "votre architecture a tel risque structurel, voici comment y remédier".

Pour aller plus loin

Pour la liste détaillée des critères et les checks par zone, consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/guidelines-advisor/SKILL.md` et la ressource `resources/ASSESSMENT_AREAS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `guidelines-advisor`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/guidelines-advisor/SKILL.md`

secure-workflow-guide

secure-workflow-guide

> *Catalogue généré le 2026-05-11*

En une phrase

Pilote ton développement de smart contracts via un workflow sécurité Trail of Bits en 5 étapes (Slither, features spéciales, diagrammes, propriétés à fuzzer, revue manuelle) — à faire à chaque check-in ou avant déploiement.

Quand l'utiliser

- À **chaque check-in important** sur un projet de smart contracts (donc régulier, pas seulement avant l'audit).
- Avant un déploiement en mainnet.
- Quand tu veux une routine de sécurité qui couvre à la fois les outils auto (Slither, slither-prop, Echidna) et la revue manuelle des angles que les outils ratent.
- Pour structurer ta sécurité au-delà du simple "j'ai lancé Slither une fois".
- Quand le projet a des features spéciales (upgrades, tokens ERC, IBC...) et que tu veux les checks dédiés.

Comment l'invoquer

- **Slash command** : `/secure-workflow-guide`
- **Phrases déclencheurs (texte)** : "run secure development workflow" / "security check-in" / "pre-deployment review"
- **Auto-invocation** : Sur demande explicite (idéalement à chaque check-in).

Description détaillée

Trail of Bits a formalisé un workflow en 5 étapes pour maintenir un niveau de sécurité élevé tout au long du développement (et pas juste à la fin). Ce skill l'opérationnalise et l'adapte à ton projet (= il ne lance que ce qui s'applique).

Étape 1 — Check for Known Security Issues : exécute **Slither** (l'analyseur statique star de TOB, avec 70+ détecteurs intégrés pour Solidity), parse les findings par sévérité, trie les faux positifs avec toi. **Étape 2 — Check Special Features** : si tu as de l'upgradeabilité → `slither-check-upgradeability` (17 détecteurs de risques d'upgrade). Si c'est un token → `slither-check-erc` (conformité ERC) + redirection vers `token-integration-analyzer`. **Étape 3 — Visual Security Inspection** : génère 3 diagrammes (héritage, function summary, qui peut écrire dans quel storage) pour repérer visuellement les problèmes. **Étape 4 — Document Security Properties** : la phase la plus importante — t'aide à écrire les invariants (= "telle chose ne doit jamais arriver"), puis configure **Echidna** (fuzzer property-based) ou **Manticore** (exécution symbolique) pour les tester. **Étape 5 — Manual Review** : analyse les angles que les outils ne voient pas (privacy, front-running, MEV, cryptographie faible, hypothèses sur les oracles/flash loans).

C'est un workflow **récurrent** (à relancer souvent), contrairement à `audit-prep-assistant` qui est ponctuel.

Pour aller plus loin

Pour les commandes exactes de chaque étape et les explications détaillées, consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/secure-workflow-guide/SKILL.md` et la ressource `resources/WORKFLOW_STEPS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `secure-workflow-guide`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/secure-workflow-guide/SKILL.md`

solana-vulnerability-scanner

solana-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne automatiquement le code d'un programme Solana pour repérer 6 types de failles de sécurité critiques propres à cette blockchain (validation des comptes, signataires, adresses dérivées, appels inter-programmes, etc.).

Quand l'utiliser

- Avant de déployer un programme Solana (en Rust natif ou avec le framework Anchor) sur le réseau principal.
- Quand tu veux faire auditer un projet Solana ou préparer son audit externe.
- Quand tu intègres un programme qui appelle d'autres programmes ("CPI", c'est-à-dire un programme qui en invoque un autre) et que tu veux vérifier que c'est fait proprement.
- Pour valider la gestion des PDA (Program Derived Addresses, des adresses "calculées" à partir d'une formule et utilisées comme comptes spéciaux).
- Quand un dev t'envoie son code Solana et que tu veux faire un premier filtre rapide.

Comment l'invoquer

- **Slash command** : `/solana-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my Solana program" / "check this Anchor contract" / "scan Solana CPI"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

Solana est une blockchain rapide où les programmes (l'équivalent des smart contracts) sont écrits en Rust, parfois avec un framework appelé Anchor. Le modèle Solana est très particulier : chaque transaction passe une liste de "comptes" au programme, et c'est au programme de vérifier que ces comptes sont les bons. Beaucoup de hacks viennent d'un oubli de vérification.

Ce skill cherche 6 catégories de bugs classiques : 1) **Arbitrary CPI** (le programme appelle un autre programme sans vérifier que c'est le bon — un attaquant peut substituer un programme malveillant), 2) **PDA mal validée** (l'adresse dérivée n'est pas calculée canoniquement, donc usurpable), 3) **Owner check manquant** (on lit les données d'un compte sans vérifier qui le possède), 4) **Signer check manquant** (on autorise une action sans vérifier que la bonne personne a signé), 5) **Sysvar usurpé** (utilisation d'anciennes fonctions non sécurisées pour lire les variables système), 6) **Mauvaise introspection des instructions**.

Le workflow : tu lui donnes ton code Rust/Anchor, il fait des recherches `ripgrep` ciblées, te liste les findings avec gravité (CRITICAL/HIGH/MEDIUM), pointe les lignes concernées et propose un patch.

Pour aller plus loin

Pour les détails techniques (patterns exacts, exemples de code vulnérable vs corrigé, tests Anchor pour reproduire les attaques), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/solana-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `solana-vulnerability-scanner`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/solana-vulnerability-scanner/SKILL.md`

ton-vulnerability-scanner

ton-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne les smart contracts TON (The Open Network, la blockchain de Telegram) écrits en FunC pour repérer 3 types de failles critiques liées à la validation de l'expéditeur, aux dépassements arithmétiques et à la gestion du gas.

Quand l'utiliser

- Quand tu auditeras un contrat TON ou Jetton (le standard de token sur TON).
- Avant de déployer un contrat FunC en prod sur le mainnet TON.
- Quand le projet implique un wallet TON ou un système de notifications de transfert (un piège classique : croire qu'un message vient du vrai contrat Jetton alors qu'il vient d'un faux).
- Pour vérifier que les opérations privilégiées (mint, burn, withdraw) valident bien le sender.
- Quand tu transfères du TON entre contrats et que tu veux t'assurer que le gas est correctement réservé pour éviter que la transaction échoue à mi-chemin.

Comment l'invoquer

- **Slash command** : `/ton-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my TON contract" / "check FunC contract" / "scan Jetton implementation"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

TON est la blockchain associée à Telegram, avec une architecture très différente d'Ethereum (modèle asynchrone à base de messages). Les contrats sont en FunC, un langage bas niveau. Les bugs TON les plus courants sont liés à la nature "message-passing" : un contrat reçoit un message et doit vérifier qui l'envoie, sinon n'importe qui peut déclencher des actions privilégiées.

Ce skill cherche 3 patterns critiques : 1) **Missing Sender Check** (le contrat ne vérifie pas l'adresse de l'expéditeur — quelqu'un peut appeler une fonction admin), 2) **Integer Overflow** (l'arithmétique FunC déborde silencieusement si non vérifiée — perte ou création de tokens), 3) **Improper Gas Handling** (mauvaise réservation de gas lors d'un `send_raw_message` — la transaction échoue à mi-parcours et laisse des états incohérents).

Il regarde aussi les pièges Jetton : un faux contrat Jetton peut envoyer un message `transfer_notification` pour piéger un contrat naïf. Le workflow : tu pointes le dossier `contracts/` avec tes `.fc`, le skill analyse et te liste les findings avec recommandations.

Pour aller plus loin

Pour les détails techniques (patterns FunC, exemples Jetton, recommandations de gas), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/ton-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `ton-vulnerability-scanner`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/ton-vulnerability-scanner/SKILL.md`

cairo-vulnerability-scanner

cairo-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne les smart contracts Cairo (le langage des dApps StarkNet, le "L2" de validité sur Ethereum) pour repérer 6 types de failles critiques liées à l'arithmétique felt252, au stockage, au contrôle d'accès et aux ponts L1/L2.

Quand l'utiliser

- Quand tu auditeras un contrat StarkNet écrit en Cairo (le langage spécifique de cet écosystème).
- Avant de mettre en prod un projet sur StarkNet (DeFi, NFT, etc.).
- Quand le projet inclut un pont L1-L2 (un mécanisme qui fait transiter des fonds entre Ethereum et StarkNet) — c'est souvent là que se trouvent les plus gros bugs.
- Pour valider les fonctions qui acceptent des messages venant d'Ethereum (`#[l1_handler]`).
- Si tu veux passer un coup de "Caracal" (l'analyseur statique Trail of Bits pour Cairo) avant l'audit externe.

Comment l'invoquer

- **Slash command** : `/cairo-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my StarkNet contract" / "check Cairo contract" / "scan L1-L2 bridge"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

Cairo est le langage des smart contracts sur StarkNet, un "rollup de validité" qui exécute du code hors-Ethereum puis prouve cryptographiquement le résultat. Cairo a un type natif bizarre, le `felt252` (un entier sur 252 bits, presque comme un `uint256` mais pas tout à fait), qui crée des bugs d'arithmétique inattendus. Cairo 1.0 a changé pas mal de choses — beaucoup de codes existants ont des reliquats Cairo 0 dangereux.

Ce skill cherche 6 patterns critiques : 1) **Unchecked Arithmetic** (overflow/underflow sur `felt252`), 2) **Storage Collision** (deux variables qui se mappent au même slot de stockage — l'une écrase l'autre), 3) **Missing Access Control** (pas de vérification du caller sur des fonctions sensibles), 4) **Improper Felt252 Boundaries** (oublier que `felt252` n'est pas exactement un uint256), 5) **Unvalidated Contract Address** (utiliser une adresse fournie par l'utilisateur sans la valider), 6) **Missing Caller Validation** (oublier `get_caller_address()` sur une fonction admin).

Il peut aussi exécuter **Caracal**, l'outil officiel Trail of Bits (`caracal detect src/`), pour compléter avec des règles automatisées.

Pour aller plus loin

Pour les détails techniques (exemples Cairo 1.0, patterns L1 handlers, intégration Caracal et Starknet Foundry), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/cairo-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `cairo-vulnerability-scanner`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/cairo-vulnerability-scanner/SKILL.md`

algorand-vulnerability-scanner

algorand-vulnerability-scanner

> *Catalogue généré le 2026-05-11*

En une phrase

Scanne les smart contracts Algorand (écrits en TEAL ou PyTeal) pour repérer 11 types de failles classiques sur cette chaîne, dont la plus emblématique : l'attaque par "rekeying" (donner l'autorité d'un compte à un attaquant).

Quand l'utiliser

- Quand tu auditeras un projet Algorand (DeFi, NFT, application stateful).
- Avant de pousser une "logic signature" (smart contract sans état) en prod.
- Pour vérifier qu'un programme TEAL bloque bien la "rekeying attack" (= modifier secrètement la clé d'autorité d'un compte pour en prendre le contrôle).
- Quand le projet utilise des transactions groupées atomiques ("atomic groups") — l'ordre des transactions peut être manipulé.
- Pour tester rapidement avec **Tealer** (l'analyseur statique Trail of Bits dédié à Algorand).

Comment l'invoquer

- **Slash command** : `/algorand-vulnerability-scanner`
- **Phrases déclencheurs (texte)** : "audit my Algorand contract" / "check PyTeal contract" / "scan TEAL approval program"
- **Auto-invocation** : Sur demande explicite (généralement lors d'un audit).

Description détaillée

Algorand est une blockchain "pure proof-of-stake" avec un modèle de transaction très particulier : chaque transaction porte beaucoup de champs (RekeyTo, CloseRemainderTo, AssetCloseTo...) que le contrat doit vérifier explicitement. Si tu oublies de vérifier `RekeyTo`, un attaquant peut détourner ton compte. Les contrats sont écrits en TEAL (assembleur) ou plus souvent en PyTeal (DSL Python qui compile en TEAL).

Ce skill couvre 11 patterns : 1) **Rekeying Vulnerability** (CRITICAL — oubli du check RekeyTo), 2) **Missing Transaction Verification** (pas de check GroupSize/GroupIndex), 3) **Group Transaction Manipulation**, 4) **Asset Clawback Risk**, 5) **Application State Manipulation**, 6) **Asset Opt-In Missing**, 7) **Minimum Balance Violation**, 8) **Close Remainder To Check** (oubli qui peut vider un compte), 9) **Application Clear State** (peut être abusé pour bypass), 10) **Atomic Transaction Ordering** (supposer un ordre sans le valider), 11) **Logic Signature Reuse** (sigs réutilisables = replay attack).

Le skill cherche les fichiers `.teal` et `.py` (PyTeal), peut lancer **Tealer** (`pip3 install tealer; tealer contract.teal --detect all`) et te livre un rapport hiérarchisé.

Pour aller plus loin

Pour les détails techniques (patterns TEAL/PyTeal complets, exemples de vulnérabilités exploitables, intégration Tealer et Beaker), consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/algorand-vulnerability-scanner/SKILL.md` et la ressource `resources/VULNERABILITY_PATTERNS.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `algorand-vulnerability-scanner`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/algorand-vulnerability-scanner/SKILL.md`

audit-prep-assistant

audit-prep-assistant

> *Catalogue généré le 2026-05-11*

En une phrase

T'aide à préparer ton codebase pour un audit de sécurité externe en suivant la checklist Trail of Bits : objectifs clairs, analyse statique, nettoyage de code mort, tests, documentation — pour que les auditeurs travaillent vite et bien.

Quand l'utiliser

- 1 à 2 semaines **avant** un audit de sécurité externe d'un projet blockchain.
- Quand tu veux maximiser la valeur d'un audit (= ne pas payer des experts pour qu'ils découvrent des bugs triviaux que Slither aurait trouvé en 5 minutes).
- Pour donner aux auditeurs un point d'entrée propre : commit gelé, build qui marche, liste des fichiers in-scope.
- Quand tu veux écrire les "user stories" et flowcharts qui expliquent ce que ton protocole est censé faire (les auditeurs trouvent plus de bugs quand ils comprennent l'intention).
- Pour faire un dernier sweep de "low-hanging fruits" et de code mort avant de lancer le chrono d'audit.

Comment l'invoquer

- **Slash command** : `/audit-prep-assistant`
- **Phrases déclencheurs (texte)** : "prep for audit" / "get ready for security review" / "audit checklist"
- **Auto-invocation** : Sur demande explicite (généralement avant un audit planifié).

Description détaillée

Un audit de sécurité coûte cher (souvent 20-50k+ pour un protocole DeFi). Si les auditeurs passent 3 jours à comprendre comment builder ton projet ou à trier des findings Slither que tu aurais pu fixer toi-même, c'est de l'argent perdu. Trail of Bits a publié une checklist de préparation, et ce skill l'opérationnalise.

Le processus en plusieurs étapes : 1) **Set Review Goals** (le skill te pose des questions pour cadrer : quel niveau de sécurité vises-tu, quelles zones t'inquiètent le plus, quel scénario catastrophe veux-tu éviter), 2) **Resolve Easy Issues** (lance Slither pour Solidity, dylint pour Rust, golangci-lint pour Go, et t'aide à trier/fixer), 3) **Augmente la coverage de tests** et identifie le code non testé, 4) **Supprime le code mort** (fonctions/libs/features inutilisées), 5) **Rends le code accessible** (liste des fichiers in/out scope, instructions de build, branche dédiée, version freezée), 6) **Génère de la documentation** (flowcharts, user stories, commentaires inline).

C'est un skill **transversal** (pas spécifique à une chaîne) — il s'adapte à Solidity, Rust, Go, Cairo, etc.

Pour aller plus loin

Pour la checklist complète Trail of Bits et les commandes outil par outil, consulter le SKILL.md original à `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/audit-prep-assistant/SKILL.md`.

Source

- **Plugin** : `trailofbits/building-secure-contracts`
- **Nom interne** : `audit-prep-assistant`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/building-secure-contracts/1.0.1/skills/audit-prep-assistant/SKILL.md`