# 04 — Audit de code (Trail of Bits)

Outils d'audit de code Trail of Bits (Semgrep, differential review, sharp edges, etc.).

# □□ Index — 04 — Audit de code (Trail of Bits)

# 04 — Audit de code (Trail of Bits)

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

**12 skills documentés.**

| Skill | En une phrase |
|---|---|
| **ask-questions-if-underspecified** | Force Claude à te poser des questions de clarification avant de se lancer dans le code, quand ta demande peut être interprétée de… |
| **audit-context-building** | Avant de chercher des bugs, force Claude à lire le code ligne par ligne et à construire une compréhension profonde et structurée… |
| **burpsuite-project-parser** | Permet de fouiller en ligne de commande dans les fichiers de projet de Burp Suite (un outil de test d'intrusion web très répandu… |
| **constant-time-analysis** | Détecte les bouts de code cryptographique qui mettent un peu plus (ou un peu moins) de temps à s'exécuter selon la valeur d'un… |
| **differential-review** | Compare deux versions d'un code (avant/après une modification) pour repérer ce qui a changé et identifier les risques de bugs ou… |
| **dwarf-expert** | Apporte une expertise pointue sur DWARF — un format technique caché dans les programmes compilés qui permet aux outils de debug… |
| **entry-point-analyzer** | Cartographie toutes les « portes d'entrée » d'un smart contract — c'est-à-dire les fonctions que n'importe qui peut appeler… |

| Skill | En une phrase |
| --- | --- |
| **interpreting-culture-index** | Interprète les résultats du test Culture Index — un sondage RH qui mesure des traits comportementaux (autonomie, sociabilité,… |
| **semgrep-rule-creator** | Aide à écrire des règles personnalisées pour Semgrep — un outil qui scanne le code à la recherche de motifs vulnérables — afin de… |
| **sharp-edges** | Repère les API ou les configurations qui ressemblent à des « pièges à doigts » : techniquement utilisables, mais conçues d'une… |
| **spec-to-code-compliance** | Compare un document de spécification (le « cahier des charges » d'un projet, type whitepaper) au code réellement écrit, pour… |
| **variant-analysis** | Quand on a trouvé un bug à un endroit, ce skill cherche systématiquement le même type de bug ailleurs dans le code — parce qu'une… |

# entry-point-analyzer

# entry-point-analyzer

> *Catalogue généré le 2026-05-11*

## En une phrase

Cartographie toutes les « portes d'entrée » d'un smart contract — c'est-à-dire les fonctions que n'importe qui peut appeler depuis l'extérieur pour modifier l'état du contrat — afin de savoir où concentrer un audit de sécurité.

## Quand l'utiliser

- Au tout début d'un audit de smart contract (Solidity, Vyper, Solana, Move, TON, CosmWasm…), pour dresser la liste des points exposés.
- Pour comprendre quelles fonctions sont accessibles publiquement et lesquelles sont réservées à l'admin.
- Quand tu veux vérifier le « contrôle d'accès » — autrement dit, qui a le droit de faire quoi dans le contrat.
- Avant de chercher des vulnérabilités spécifiques : on commence toujours par savoir où l'attaquant peut frapper.

## Comment l'invoquer

- **Slash command** : `/entry-point-analyzer` (ou `/entry-points`).
- **Phrases déclencheurs (texte)** : "find entry points", "external functions", "audit flows", "access control", "privileged operations".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Un smart contract, c'est un programme qui vit sur la blockchain. Il expose des fonctions, comme des boutons qu'on peut presser depuis l'extérieur. Certains boutons sont réservés à l'admin (par exemple : retirer de l'argent du contrat), d'autres sont accessibles à tout le monde (par exemple : transférer ses propres tokens). Le skill `entry-point-analyzer` dresse la liste exhaustive de ces boutons et indique qui a le droit de les actionner.

Il ne s'intéresse qu'aux fonctions qui changent l'état du contrat (celles qui peuvent déplacer de l'argent, modifier des balances, changer des permissions). Les fonctions purement « lecture seule » (`view`, `pure`) sont exclues parce qu'elles ne peuvent pas, à elles seules, faire perdre des fonds. Pour les contrats en Solidity, le skill peut s'appuyer sur Slither, un outil d'analyse statique très utilisé, qui produit automatiquement le tableau des entry points.

En sortie, tu obtiens un rapport Markdown structuré : la liste des fonctions exposées, leur niveau d'accès (public, admin, restreint à un rôle, accessible seulement par un autre contrat), et les « modifiers » de sécurité appliqués (les garde-fous comme `onlyOwner`). C'est la base de travail à partir de laquelle tout audit sérieux démarre.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/entry-point-analyzer`
- **Nom interne** : `entry-point-analyzer`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/entry-point-analyzer/1.0.0/skills/entry-point-analyzer/SKILL.md`

# variant-analysis

# variant-analysis

> *Catalogue généré le 2026-05-11*

# En une phrase

Quand on a trouvé un bug à un endroit, ce skill cherche systématiquement le même type de bug ailleurs dans le code — parce qu'une erreur faite une fois est souvent reproduite plusieurs fois.

# Quand l'utiliser

- Tu viens de découvrir une faille à un endroit et tu veux savoir si elle est répétée ailleurs.
- Tu veux construire une règle automatique (avec Semgrep ou CodeQL, deux outils qui scannent le code) pour repérer un motif vulnérable.
- Après un audit qui a trouvé un problème : tu veux remonter à la racine et débusquer tous les cousins.
- Quand un correctif vient d'être publié pour une bibliothèque et que tu veux vérifier si ton code contient le même schéma vulnérable.

# Comment l'invoquer

- **Slash command** : `/variant-analysis` (ou `/variants`).
- **Phrases déclencheurs (texte)** : "find similar bugs", "hunt variants", "build a CodeQL query", "search for this pattern".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Le principe de base de ce skill, c'est qu'un bug est rarement seul. Quand un développeur fait une erreur dans une partie du code (par exemple : oublier de vérifier qu'une valeur n'est pas négative), il a probablement fait la même erreur à plusieurs autres endroits, par copier-coller ou simplement par habitude. Plutôt que d'attendre que ces bugs frères soient découverts un par un, on les chasse tous d'un coup.

Le skill fonctionne en cinq étapes : 1) bien comprendre la cause racine du bug initial (pas juste son symptôme), 2) écrire un motif de recherche qui retrouve exactement le bug d'origine, 3) repérer les éléments qu'on peut généraliser (le nom de la variable n'a pas d'importance, mais la structure de l'appel oui), 4) élargir le motif petit à petit en vérifiant qu'on ne génère pas trop de faux positifs (alertes injustifiées), 5) trier les résultats par exploitabilité.

En entrée, on lui donne un bug connu. En sortie, on obtient une liste de candidats potentiels dans tout le projet, classés par niveau de confiance (élevé / moyen / faible), avec leur fichier et leur ligne. Il choisit son outil selon le besoin : `ripgrep` pour une recherche textuelle rapide, Semgrep pour des motifs syntaxiques, ou CodeQL pour suivre des flux de données complexes à travers le code.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/variant-analysis`
- **Nom interne** : `variant-analysis`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/variant-analysis/1.0.0/skills/variant-analysis/SKILL.md`

# ask-questions-if-underspecified

# ask-questions-if-underspecified

> *Catalogue généré le 2026-05-11*

## En une phrase

Force Claude à te poser des questions de clarification avant de se lancer dans le code, quand ta demande peut être interprétée de plusieurs façons — pour éviter de coder pendant deux heures dans la mauvaise direction.

## Quand l'utiliser

- Quand tu demandes quelque chose de vague (« améliore l'affichage », « ajoute une page ») et que tu veux que Claude vérifie qu'il a bien compris.
- Avant un gros chantier où une mauvaise compréhension coûterait cher.
- Quand plusieurs interprétations sont possibles et que tu préfères choisir explicitement.
- Quand tu veux établir ensemble les critères de « c'est fini » avant de coder.

## Comment l'invoquer

- **Slash command** : `/ask-questions-if-underspecified`.

- **Phrases déclencheurs (texte)** : "clarify requirements", "ask before implementing", "are you sure you understood".
- **Auto-invocation** : Activé automatiquement dès que la demande est ambiguë, ou sur demande explicite.

# Description détaillée

C'est un skill méta : il ne fait pas de code, il discipline la manière dont Claude réagit à une demande. L'idée part d'un constat simple : quand un dev (humain ou IA) se lance sans avoir compris le besoin, il finit souvent par livrer quelque chose qui ne correspond pas, qu'il faut tout refaire. Plutôt que de deviner, autant prendre cinq minutes pour s'aligner.

Concrètement, dès que Claude reçoit une demande, il vérifie six choses : l'objectif (qu'est-ce qui doit changer ?), la définition de « terminé » (à quoi je reconnais que c'est fait ?), le périmètre (quels fichiers, quels composants ?), les contraintes (perf, style, dépendances), l'environnement (quelle version, quel OS ?), et la sécurité (faut-il une procédure de rollback ?). Si une de ces dimensions est floue, il s'arrête et te pose 1 à 5 questions courtes, sous forme numérotée avec des options à choisir.

Mieux : il propose toujours une option par défaut (marquée « recommandée ») et accepte une réponse compacte type `defaults` ou `1a 2b 3a`. Tu n'as donc pas besoin de rédiger un cahier des charges, juste de cocher des cases. Une fois les réponses obtenues, Claude reformule ta demande en deux phrases pour confirmer, puis commence. Ça évite la moitié des « ah non, ce n'était pas ce que je voulais ».

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/ask-questions-if-underspecified`
- **Nom interne** : `ask-questions-if-underspecified`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/ask-questions-if-underspecified/1.0.1/skills/ask-questions-if-underspecified/SKILL.md`

# spec-to-code-compliance

# spec-to-code-compliance

> *Catalogue généré le 2026-05-11*

# En une phrase

Compare un document de spécification (le « cahier des charges » d'un projet, type whitepaper) au code réellement écrit, pour vérifier que le code fait exactement ce que la doc promet — ni plus, ni moins.

# Quand l'utiliser

- Tu audites un smart contract et tu disposes du whitepaper (le document fondateur d'un projet blockchain qui décrit comment il est censé fonctionner).
- Tu veux trouver les « trous » entre ce qui est promis sur le papier et ce qui est codé.
- Tu cherches du code qui fait des choses non documentées (potentiellement un backdoor ou une fonctionnalité oubliée).
- Tu veux valider qu'une implémentation respecte un protocole standard.

# Comment l'invoquer

- **Slash command** : `/spec-to-code-compliance` (ou `/spec-compliance`).
- **Phrases déclencheurs (texte)** : "does this code match the spec?", "what's missing from the implementation?", "compliance check", "compare to whitepaper".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Dans le monde de la blockchain et des protocoles décentralisés, un projet commence presque toujours par un « whitepaper » : un document qui explique en détail ce que le système est censé faire (les règles, les calculs, les garanties de sécurité, les invariants à respecter). Puis des développeurs traduisent ce document en code. Le problème, c'est que la traduction n'est jamais parfaite : il y a des oublis, des interprétations, des raccourcis. Ce skill traque ces écarts.

Il fonctionne en plusieurs phases. D'abord, il identifie tous les documents qui font office de spécification (whitepaper PDF, Markdown, notes de design). Ensuite, il extrait de manière exhaustive chaque exigence ou affirmation de la doc. Puis il cherche dans le code la portion qui implémente cette exigence. Pour chaque correspondance, il attribue un type (match complet, match partiel, divergence, exigence non implémentée) et un niveau de confiance entre 0 et 1. Il signale aussi le sens inverse : du code qui existe mais qui ne correspond à aucune ligne de la spec — c'est souvent là que se cachent les vulnérabilités les plus retorses.

La règle d'or du skill : ne jamais inventer. Chaque conclusion doit citer une preuve précise (section du document, numéro de ligne du code). Pas de supposition, pas de « probablement ». Quand quelque chose est ambigu, c'est classé comme tel plutôt que tranché à la légère.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/spec-to-code-compliance`
- **Nom interne** : `spec-to-code-compliance`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/spec-to-code-compliance/1.1.0/skills/spec-to-code-compliance/SKILL.md`

# semgrep-rule-creator

# semgrep-rule-creator

> *Catalogue généré le 2026-05-11*

# En une phrase

Aide à écrire des règles personnalisées pour Semgrep — un outil qui scanne le code à la recherche de motifs vulnérables — afin de détecter automatiquement un type de bug spécifique à ton projet.

# Quand l'utiliser

- Tu as identifié un motif vulnérable récurrent et tu veux qu'un scan automatique le détecte à chaque commit.
- Tu veux imposer une convention de code à ton équipe (par exemple : « interdiction d'utiliser telle fonction dangereuse »).
- Tu veux suivre la trace d'une donnée « salie » par l'utilisateur jusqu'à un point sensible (mode « taint », qui suit le flux des données).
- Tu écris une règle Semgrep et tu veux la tester proprement avec des cas vulnérables ET des cas sûrs.

# Comment l'invoquer

- **Slash command** : `/semgrep-rule-creator` (ou `/semgrep-rule`).
- **Phrases déclencheurs (texte)** : "write a Semgrep rule", "build a custom static analysis detection", "detect this pattern".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Semgrep est un outil qui parcourt ton code et signale tous les endroits où un certain motif apparaît. Par exemple, on peut lui demander de signaler tous les appels à `eval()` (une fonction réputée dangereuse) ou toutes les requêtes SQL construites par concaténation. Le skill `semgrep-rule-creator` t'aide à écrire ces règles correctement, parce qu'une règle mal écrite produit soit trop d'alertes inutiles (faux positifs qui érodent la confiance), soit pas assez (faux négatifs qui laissent passer des vrais bugs).

Le skill insiste sur la méthode : commencer par un motif large pour bien capturer le bug initial, puis l'affiner jusqu'à ce qu'il ne déclenche que sur le vrai cas dangereux. Il rappelle aussi qu'une règle non testée ne sert à rien : il faut toujours fournir des exemples de code vulnérable (qui doivent déclencher l'alerte) et des exemples sûrs (qui ne doivent pas la déclencher). Il décourage les patterns trop larges (qui matchent tout et n'importe quoi) et trop spécifiques (qui ratent les variantes).

Pour les cas où on veut suivre la propagation d'une donnée — par exemple, une entrée utilisateur qui finit dans une commande système — il guide vers le mode « taint » de Semgrep, qui modélise les sources (où entre la donnée) et les sinks (où elle ne devrait jamais arriver sans nettoyage). En sortie : une règle YAML prête à intégrer dans ton pipeline d'intégration continue (CI).

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/semgrep-rule-creator`
- **Nom interne** : `semgrep-rule-creator`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/semgrep-rule-creator/1.1.0/skills/semgrep-rule-creator/SKILL.md`

# burpsuite-project-parser

# burpsuite-project-parser

> *Catalogue généré le 2026-05-11*

## En une phrase

Permet de fouiller en ligne de commande dans les fichiers de projet de Burp Suite (un outil de test d'intrusion web très répandu chez les pentesters), pour retrouver des requêtes HTTP, des en-têtes ou des réponses suspectes.

## Quand l'utiliser

- Tu as réalisé un audit de sécurité d'un site web avec Burp Suite et tu veux extraire des trouvailles précises de ton fichier `.burp`.
- Tu veux chercher un motif spécifique (mot-clé, regex) à travers des milliers de requêtes capturées.
- Tu prépares un rapport et tu dois retrouver toutes les réponses qui contiennent un certain en-tête (par exemple : `Set-Cookie` sans `Secure`).
- Tu veux automatiser l'exploration du trafic HTTP sans cliquer manuellement dans l'interface graphique de Burp.

## Comment l'invoquer

- **Slash command** : `/burpsuite-project-parser` (ou `/burp-search`).
- **Phrases déclencheurs (texte)** : "search Burp project", "analyze .burp file", "extract findings from Burp", "dump proxy history".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Burp Suite est l'outil de référence pour auditer la sécurité des applications web : il se positionne comme un proxy entre ton navigateur et le serveur, et capture toutes les requêtes et réponses qui passent. À la fin d'une session, tu te retrouves avec un fichier `.burp` qui peut peser plusieurs giga-octets et contenir des dizaines de milliers d'échanges. Impossible à explorer à la main.

Ce skill ne lit pas le fichier directement (le format est propriétaire), mais s'appuie sur Burp Suite Professional et une extension officielle appelée `burpsuite-project-file-parser`. Une fois cette extension installée, le skill fournit un script wrapper qui te permet de lancer des recherches ciblées depuis le terminal — par exemple : « donne-moi toutes les réponses dont le code de statut est 500 », ou « trouve-moi toutes les requêtes qui contiennent le mot `password` dans leur corps ».

Le skill insiste fortement sur les filtres : récupérer l'historique complet d'un projet Burp peut renvoyer plusieurs giga-octets de texte qui saturent la console. Il préconise donc de toujours commencer par les en-têtes (petits, rapides à scanner) et de ne charger les corps de réponses qu'après avoir identifié les URLs intéressantes. Idéal pour extraire automatiquement des éléments de preuve à coller dans un rapport d'audit.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/burpsuite-project-parser`
- **Nom interne** : `burpsuite-project-parser`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/burpsuite-project-parser/1.0.0/skills/SKILL.md`

# differential-review

# differential-review

> *Catalogue généré le 2026-05-11*

## En une phrase

Compare deux versions d'un code (avant/après une modification) pour repérer ce qui a changé et identifier les risques de bugs ou de failles introduits par la modif.

## Quand l'utiliser

- Avant de fusionner une « pull request » (proposition de modification de code) sur ton dépôt.
- Quand un collègue ou Claude a touché à du code sensible (connexion, paiements, gestion de mots de passe).
- Pour vérifier qu'un « refactor » (réorganisation du code sans changer ses fonctionnalités) n'a pas cassé une protection au passage.
- Avant un déploiement en production sur un projet sérieux.
- Quand tu veux un rapport écrit qui justifie pourquoi tu valides (ou pas) un changement.

## Comment l'invoquer

- **Slash command** : `/differential-review`
- **Phrases déclencheurs (texte)** : "review this PR", "differential review", "check my diff", "pre-landing review".
- **Auto-invocation** : Sur demande explicite, ou suggérée automatiquement avant un merge.

# Description détaillée

Ce skill intervient au moment où une modification de code est sur le point d'être intégrée (un merge, un déploiement, une release). Il prend en entrée un « diff » — c'est-à-dire la liste des lignes ajoutées, supprimées ou modifiées — et le compare au reste du projet pour comprendre l'impact réel du changement.

Il ne se contente pas de lire les nouvelles lignes : il regarde aussi l'historique Git (qui a touché à ces fichiers avant, quand, pourquoi), calcule le « blast radius » (autrement dit : combien d'autres parties du code dépendent de ce que tu changes), vérifie si des tests automatiques couvrent les nouvelles lignes, et adapte sa profondeur d'analyse à la taille du projet (petit / moyen / gros).

En sortie, tu obtiens un rapport en Markdown qui liste les problèmes de sécurité potentiels, classés par niveau de risque (élevé / moyen / faible), avec pour chacun une preuve concrète (numéro de ligne, scénario d'attaque imaginé) et un niveau de confiance. Il sait notamment repérer les régressions de sécurité — par exemple, une vérification d'identité qu'on aurait silencieusement retirée en refactorisant.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/differential-review`
- **Nom interne** : `differential-review`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/differential-review/1.0.0/skills/differential-review/SKILL.md`

# audit-context-building

# audit-context-building

> *Catalogue généré le 2026-05-11*

# En une phrase

Avant de chercher des bugs, force Claude à lire le code ligne par ligne et à construire une compréhension profonde et structurée du projet — parce qu'on ne trouve pas de failles dans un code qu'on n'a pas vraiment compris.

# Quand l'utiliser

- Au démarrage d'un audit de sécurité sérieux, avant toute chasse aux vulnérabilités.
- Pour préparer une revue d'architecture ou un threat modeling (exercice qui imagine comment des attaquants pourraient s'en prendre au système).
- Quand tu veux éviter les conclusions hâtives basées sur « j'ai compris en gros ».
- Quand le code te paraît compliqué et que tu sens qu'une compréhension superficielle te ferait rater des choses.

# Comment l'invoquer

- **Slash command** : `/audit-context-building` (ou `/audit-context`).
- **Phrases déclencheurs (texte)** : "deep context", "audit prep", "build understanding before bug hunting", "line-by-line analysis".
- **Auto-invocation** : Sur demande explicite, au tout début d'une mission d'audit.

# Description détaillée

C'est un skill de discipline mentale, pas un outil. Il configure la manière dont Claude « pense » pendant la phase préparatoire d'un audit. Plutôt que de survoler le code à la recherche de patterns suspects (ce qui est tentant mais qui produit des hallucinations et des faux positifs), Claude bascule en mode « ultra-granulaire » : il lit fonction par fonction, bloc par bloc, parfois ligne par ligne.

Pour chaque morceau de code, il applique trois techniques. Le « First Principles » consiste à se demander ce que le code fait vraiment, sans accepter les apparences. Les « 5 Pourquoi » consistent à creuser une décision de design jusqu'à toucher la raison profonde. Les « 5 Comment » servent à explorer toutes les manières dont une logique peut être déclenchée ou contournée. À chaque étape, Claude note explicitement ses hypothèses, les invariants qu'il croit identifier, les acteurs en jeu (utilisateurs, admin, oracles), et il met à jour son modèle mental dès qu'une nouvelle preuve contredit ce qu'il pensait.

L'objectif n'est pas de trouver des bugs (ça vient après), mais de construire une carte mentale stable et explicite du système. Une fois cette carte établie, les autres skills d'audit (chasse aux vulnérabilités, analyse de variantes, contrôle de conformité à la spec) peuvent travailler sur des bases solides plutôt que sur des suppositions. Lent en apparence, mais beaucoup plus efficace au final.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/audit-context-building`
- **Nom interne** : `audit-context-building`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/audit-context-building/1.1.0/skills/audit-context-building/SKILL.md`

# dwarf-expert

# dwarf-expert

> *Catalogue généré le 2026-05-11*

# En une phrase

Apporte une expertise pointue sur DWARF — un format technique caché dans les programmes compilés qui permet aux outils de debug (comme `gdb`) de savoir à quelle ligne de code source correspond chaque instruction machine.

# Quand l'utiliser

- Tu travailles sur un programme compilé en C, C++, Rust ou Go et tu veux comprendre comment ses informations de debug sont stockées.
- Tu utilises des outils comme `dwarfdump` ou `readelf` pour examiner un binaire (un fichier exécutable).
- Tu écris ou tu relis du code qui lit/produit des données DWARF (par exemple : un débogueur, un profileur, un outil de couverture de code).
- Tu veux valider qu'un binaire contient bien des infos de debug exploitables (`llvm-dwarfdump --verify`).

# Comment l'invoquer

- **Slash command** : `/dwarf-expert`.
- **Phrases déclencheurs (texte)** : "DWARF debug info", "parse DWARF", "dwarfdump", "readelf", "debug symbols".
- **Auto-invocation** : Sur demande explicite, ou quand tu mentionnes DWARF dans une question.

# Description détaillée

Quand on compile un programme (en C, C++, Rust…), le code source lisible est transformé en code machine illisible. Mais pour pouvoir déboguer ce programme plus tard, le compilateur joint un dictionnaire à part qui dit : « cette suite d'instructions machine correspond à la ligne 42 du fichier `main.c`, la variable `x` est stockée dans le registre EAX à ce moment-là, etc. ». Ce dictionnaire suit un standard appelé DWARF (versions 3, 4 et 5). C'est lui qui permet à `gdb` ou à un profileur de te montrer du code source plutôt que de l'assembleur.

Ce skill apporte la connaissance technique pour manipuler ces données : comment elles sont structurées, comment les extraire avec les outils standards (`dwarfdump`, `readelf`, `llvm-dwarfdump`), comment les générer correctement, et comment écrire du code qui les analyse à l'aide de bibliothèques comme `libdwarf`, `pyelftools` ou `gimli` (côté Rust).

Il est utile dans plusieurs scénarios concrets : développer un outil de couverture de code, analyser un binaire suspect dans un contexte de reverse engineering, vérifier qu'un binaire de production est correctement compilé avec ses infos de debug, ou comparer la qualité du debug entre deux versions d'un compilateur. C'est un skill très spécialisé, qui ne sert que si tu touches à des binaires compilés natifs (pas du JavaScript, pas du Python).

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/dwarf-expert`
- **Nom interne** : `dwarf-expert`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/dwarf-expert/1.0.0/skills/dwarf-expert/SKILL.md`

# constant-time-analysis

# constant-time-analysis

> *Catalogue généré le 2026-05-11*

## En une phrase

Détecte les bouts de code cryptographique qui mettent un peu plus (ou un peu moins) de temps à s'exécuter selon la valeur d'un secret — ce qui peut permettre à un attaquant de deviner ce secret juste en mesurant les temps de réponse.

## Quand l'utiliser

- Tu écris ou tu relis du code qui manipule des clés cryptographiques, des mots de passe, des tokens d'authentification.
- Tu utilises des opérations sensibles comme la division `/` ou le modulo `%` sur des valeurs secrètes.
- Tu touches à une fonction `sign`, `verify`, `encrypt`, `decrypt` ou `derive_key`.
- Tu as entendu parler de « timing attack » ou « side-channel » et tu veux savoir si ton code est concerné.

## Comment l'invoquer

- **Slash command** : `/constant-time-analysis` (ou `/ct-check`).
- **Phrases déclencheurs (texte)** : "constant-time", "timing attack", "side-channel", "KyberSlash".
- **Auto-invocation** : Sur demande explicite, ou quand tu écris du code crypto.

# Description détaillée

Imagine un cadenas digital qui dit « bipe court » quand le premier chiffre est faux et « bipe long » quand il est juste. Tu n'as plus qu'à essayer de 0 à 9 sur le premier chiffre, retenir lequel a fait le bip long, puis passer au deuxième. En quelques secondes, tu ouvres un cadenas censé être inviolable. C'est exactement le principe des « attaques par canal auxiliaire par mesure de temps » (timing side-channel) : un attaquant ne voit pas le secret, mais il voit combien de temps le programme met à répondre, et ça lui suffit parfois à le déduire.

Pour qu'un code cryptographique soit sûr, il doit prendre **exactement le même temps** quel que soit le secret manipulé — on dit qu'il est « à temps constant ». Concrètement, ça veut dire éviter les branchements `if (secret == x)` qui prennent un chemin différent selon la valeur, et éviter certaines opérations comme la division qui peuvent être plus rapides sur certains nombres que sur d'autres. Le skill analyse le code (en C, C++, Go, Rust, Java, Python, JavaScript et plein d'autres langages) et signale tous les endroits qui ne respectent pas cette contrainte.

Il fournit un script `ct_analyzer` qui produit un rapport détaillé : chaque opération suspecte, sa ligne dans le code, et pourquoi elle est risquée. Il peut sortir en JSON pour intégration dans un pipeline CI. Particulièrement utile sur les opérations sensibles modernes (signatures Ed25519, encapsulation post-quantique type Kyber, etc.). À noter : ce skill n'est pas pertinent pour du code « métier » classique — il vise uniquement le code qui manipule des secrets cryptographiques.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/constant-time-analysis`
- **Nom interne** : `constant-time-analysis`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/constant-time-analysis/0.1.0/skills/constant-time-analysis/SKILL.md`

# interpreting-culture-index

# interpreting-culture-index

> *Catalogue généré le 2026-05-11*

## En une phrase

Interprète les résultats du test Culture Index — un sondage RH qui mesure des traits comportementaux (autonomie, sociabilité, patience…) — pour aider à recruter, manager, prévenir l'épuisement professionnel ou composer des équipes équilibrées.

## Quand l'utiliser

- Tu as fait passer un test Culture Index à quelqu'un (toi, un candidat, un collègue) et tu veux comprendre les résultats.
- Tu prépares un recrutement et tu veux définir le profil idéal pour un poste.
- Tu manages une équipe et tu veux comprendre pourquoi deux personnes ne se synchronisent pas.
- Tu veux détecter un risque de burnout en comparant qui est quelqu'un naturellement (graphe Survey) et ce qu'il s'efforce d'être au travail (graphe Job).
- Tu veux composer une équipe avec un bon mix « gaz / frein / colle » (les profils qui accélèrent, ceux qui prudent, ceux qui soudent).

## Comment l'invoquer

- **Slash command** : `/interpreting-culture-index`.
- **Phrases déclencheurs (texte)** : "Culture Index", "behavioral profile", "team composition", "burnout risk", "hiring profile".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Culture Index (CI) est un sondage comportemental utilisé par beaucoup d'entreprises pour mieux comprendre comment leurs employés fonctionnent au quotidien. Il mesure six dimensions (autonomie, sociabilité, patience, conformité, logique, ingéniosité) et produit deux graphes : un graphe « Survey » qui décrit la personnalité naturelle de la personne (câblée à l'adolescence, ne change quasiment plus) et un graphe « Job » qui décrit la manière dont elle se comporte au travail (modifiable, mais demande de l'énergie).

Ce skill est l'expert qui sait lire ces graphes correctement, sans tomber dans les pièges classiques. Premier piège évité : ne jamais comparer des valeurs brutes entre deux personnes (« Dan a 8 en autonomie, Jim a 5, donc Dan est plus autonome »). Ce qui compte, c'est la distance entre le point du trait et la flèche rouge qui marque la moyenne. Deuxième piège évité : interpréter un écart entre Survey et Job comme une bonne ou une mauvaise chose. Un écart durable (3-6 mois) est un signal de burnout potentiel, parce que la personne dépense de l'énergie à faire semblant d'être quelqu'un qu'elle n'est pas.

En entrée, le skill accepte soit un JSON déjà extrait d'un rapport CI, soit un PDF brut qu'il peut analyser via un script OpenCV pour reconnaître les graphes. Il sait faire de l'interprétation individuelle, de la prédiction de traits à partir d'un transcript d'entretien, de l'analyse d'équipe, du coaching de manager, et même de la médiation de conflit en se basant sur les profils respectifs.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/culture-index`
- **Nom interne** : `interpreting-culture-index`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/culture-index/1.1.0/skills/interpreting-culture-index/SKILL.md`

# sharp-edges

# sharp-edges

> *Catalogue généré le 2026-05-11*

# En une phrase

Repère les API ou les configurations qui ressemblent à des « pièges à doigts » : techniquement utilisables, mais conçues d'une manière où l'usage normal mène facilement à une faille de sécurité.

# Quand l'utiliser

- Tu conçois une bibliothèque ou une API publique et tu veux qu'elle soit « sûre par défaut ».
- Tu audites une configuration où une option mal réglée peut tout faire sauter.
- Tu utilises ou tu évalues une bibliothèque crypto et tu te demandes si son usage « naturel » est sécurisé.
- Tu veux comprendre pourquoi des libraires connues (type JWT) ont mauvaise réputation en matière de pièges.

# Comment l'invoquer

- **Slash command** : `/sharp-edges`.
- **Phrases déclencheurs (texte)** : "footgun", "misuse-resistant", "secure defaults", "API usability", "dangerous configuration".
- **Auto-invocation** : Sur demande explicite.

# Description détaillée

Il existe deux philosophies pour concevoir une API. La première : « on offre toute la flexibilité, c'est au développeur de bien l'utiliser ». La deuxième : « on rend le chemin sécurisé tellement évident qu'on ne peut pas vraiment se tromper » — c'est le « pit of success ». Ce skill vérifie si une API tombe dans la première catégorie (auquel cas elle produira des bugs en masse) ou dans la deuxième.

Concrètement, il chasse plusieurs types de « pièges ». Les pièges de choix d'algorithme : laisser le développeur choisir entre RSA, HMAC, ou pire `none` (le célèbre piège JWT où un attaquant peut désactiver la vérification de signature). Les valeurs par défaut dangereuses : un timeout par défaut de 0 qui signifie « accepte tout », un mode debug activé par défaut. Les paramètres ambigus : `lifetime=0` veut-il dire « immédiat » ou « infini » ? Personne ne sait. Les API qui demandent au développeur de se rappeler de règles spéciales (« n'oublie pas d'appeler `verify()` après `decode()` ») au lieu de les imposer automatiquement.

Le skill rejette aussi les excuses classiques : « c'est documenté » (les devs ne lisent pas la doc sous pression), « les utilisateurs avancés ont besoin de flexibilité » (90 % des usages « avancés » sont du copier-coller), « c'est la responsabilité du dev » (non, c'est le designer qui a mis le piège). Il propose à la place des principes : choix sécurisé par défaut, primitives dangereuses cachées derrière une couche haut niveau, configurations dangereuses rejetées à la validation.

# Pour aller plus loin

Pour les détails techniques, exemples et patterns spécifiques, voir le SKILL.md original.

# Source

- **Plugin** : `trailofbits/sharp-edges`
- **Nom interne** : `sharp-edges`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/sharp-edges/1.0.0/skills/sharp-edges/SKILL.md`