Vibe coding : risques sécurité et pratiques sûres pour développeurs

> Vibe coding avec Cursor, Copilot ou Claude : les 7 risques de sécurité du code IA généré, SAST, .cursorrules et gouvernance IA pour développeurs.

Le vibe coding désigne une approche de développement où le développeur exprime son intention en langage naturel et laisse un LLM (Large Language Model) générer tout ou partie du code. En 2025, Google a révélé que 30% du code produit en interne était écrit par des systèmes d'IA, et en 2026, la tendance s'est généralisée à l'ensemble de l'industrie logicielle. Cursor, GitHub Copilot, Claude Code, Gemini Code Assist et Codeium ont transformé la façon de développer. Le gain de productivité est réel et mesurable. Mais une réalité moins médiatisée accompagne cette révolution : le code généré par les LLM introduit régulièrement des vulnérabilités de sécurité significatives, parfois subtiles, souvent invisibles sans analyse spécialisée. Des études académiques et des rapports sectoriels documentent des SQL injections introduites dans des applications bancaires, des secrets hardcodés dans des dépôts publics, et des packages npm inexistants "hallucinés" par les LLM que des attaquants ont créés avec du contenu malveillant. Ce guide analyse les 7 risques de sécurité majeurs du vibe coding, présente des pratiques concrètes pour coder avec l'IA sans compromettre la posture de sécurité, et propose un cadre de gouvernance applicable dans votre organisation.

À RETENIR

À retenir :

30% du code Google est généré par IA en 2025 ; les LLM introduisent régulièrement des vulnérabilités de sécurité comme les SQL injections, les secrets hardcodés et la logique d'authentification incorrecte.

Le risque de "package hallucination" (LLM inventant des noms de packages npm/PyPI inexistants) est documenté et exploité par des attaquants qui créent ces packages avec du contenu malveillant.

Le SAST automatique (Semgrep, Snyk Code, CodeQL) intégré dans la CI/CD est le contrôle compensatoire le plus efficace pour détecter les vulnérabilités introduites par le code IA.

Les fichiers de configuration des outils IA (.cursorrules, .github/copilot-instructions.md) permettent d'injecter des règles de sécurité dans le contexte du LLM pour orienter la génération vers un code plus sécurisé.

Définition du vibe coding : qu'est-ce que coder par intention ?

Le terme "vibe coding" a été popularisé en 2025 pour décrire un paradigme de développement où le développeur pilote l'IA par des descriptions en langage naturel plutôt que par l'écriture directe de code. "Crée une API REST qui prend un email en paramètre, vérifie sa validité, et l'enregistre en base de données avec un timestamp" — et le LLM génère immédiatement le code complet, fonctionnel, avec gestion d'erreurs et tests unitaires. L'IA agit comme un pair programmer hyper-productif.

Les principaux outils de vibe coding en 2026 incluent Cursor (IDE basé sur VS Code avec intégration profonde de Claude et GPT-4), GitHub Copilot (intégration native VS Code/ JetBrains, maintenant avec Copilot Workspace pour les tâches multi-fichiers), Claude Code (terminal-first, opère sur l'ensemble du codebase), Gemini Code Assist (Google Cloud, intégration IDE + Cloud Workstations), et Codeium / Windsurf (alternative open-source-friendly). Chacun de ces outils a ses forces et faiblesses en matière de sécurité.

Données chiffrées : l'IA dans le code de production

En 2025, Sundar Pichai a annoncé que plus de 30% du code produit chez Google était généré ou assisté par IA. GitHub rapporte que les développeurs utilisant Copilot sont 55% plus productifs en termes de vitesse de complétion des tâches. Stack Overflow Developer Survey 2025 indique que 76% des développeurs utilisent ou prévoient d'utiliser des outils d'IA dans leur workflow. Ces chiffres confirment que le vibe coding n'est plus un phénomène marginal.

> ⚡ **Retour terrain**
>
> Dans les projets de déploiement d'IA générative en entreprise, le risque le moins documenté est la fuite de données par les prompts utilisateur. Pour une ESN qui utilisait Claude Enterprise, j'ai analysé 3 mois de logs : 12 % des prompts contenaient des données confidentielles — NDA, données clients, spécifications techniques propriétaires. Les utilisateurs ne font pas la distinction entre leur outil de recherche web et leur assistant IA. La sensibilisation sur ce point précis réduit le taux à 2-3 % en 6 semaines.

Mais les données de sécurité racontent une autre histoire. Une étude de Stanford (2023, mais toujours citée dans les formations sécurité) a montré que les développeurs utilisant GitHub Copilot produisaient significativement plus de vulnérabilités de sécurité que ceux sans assistance

IA, notamment parce qu'ils avaient tendance à faire davantage confiance au code généré sans le relire attentivement. Des études plus récentes (Snyk State of AI Code Security 2025) confirment que 40% des développeurs interrogés ont déjà trouvé des vulnérabilités de sécurité dans du code IA généré qu'ils avaient intégré en production.

Les 7 risques de sécurité du vibe coding

Risque 1 : SQL injection et validation des entrées absente

Les LLM tendent à générer du code fonctionnel avant tout. Dans un contexte où la sécurité n'est pas explicitement mentionnée dans le prompt, les requêtes SQL générées peuvent utiliser la concaténation de chaînes plutôt que les requêtes préparées. Une requête comme `"SELECT * FROM users WHERE email = '" + email + "'"` est fonctionnelle mais vulnérable à l'injection SQL. Ce type de vulnérabilité apparaît plus fréquemment dans les portions de code "utilitaires" générées rapidement que dans les composants centraux sur lesquels le développeur est concentré.

```
 # Code IA potentiellement vulnérable (injection SQL)
def get_user(email):
    query = f"SELECT * FROM users WHERE email = '{email}'"
    return db.execute(query)

# Code sécurisé avec requête préparée
def get_user(email):
    query = "SELECT * FROM users WHERE email = %s"
    return db.execute(query, (email,))
```

Risque 2 : Secrets hardcodés dans le code suggéré

Les LLM entraînés sur des dépôts publics ont vu des millions d'exemples de code avec des clés API, tokens et mots de passe hardcodés (souvent dans des fichiers de configuration d'exemple ou de test). Ils reproduisent ce pattern naturellement. Une suggestion de code pour une connexion à une API externe peut inclure un exemple de clé API hardcodée. Si le développeur accepte la

suggestion sans modification et commit le fichier, le secret se retrouve dans l'historique Git. GitGuardian rapporte une augmentation de 67% des secrets exposés dans des dépôts publics entre 2023 et 2025, corrélée à l'adoption des outils de génération de code IA.

```
 # Pattern dangereux généré par IA
api_key = "sk-prod-abc123xyz789"  # Ne JAMAIS committer ça
client = openai.OpenAI(api_key=api_key)

# Pattern sécurisé avec variables d'environnement
import os
api_key = os.environ.get("OPENAI_API_KEY")
if not api_key:
    raise ValueError("OPENAI_API_KEY non configurée")
client = openai.OpenAI(api_key=api_key)
```

Risque 3 : Dépendances vulnérables ajoutées sans vérification

Lorsqu'un LLM génère du code utilisant une bibliothèque tierce, il suggère la version qu'il connaît — qui peut être ancienne et vulnérable. Plus grave : il ne vérifie pas systématiquement si la version suggérée présente des CVE critiques. Un `pip install requests==2.20.0` ou `npm install lodash@4.17.4` intégré dans un requirements.txt généré par IA peut introduire des dépendances avec des vulnérabilités connues et patchées depuis. L'audit des dépendances (npm audit, pip-audit, OWASP Dependency Check) est indispensable sur tout code assisté par IA.

Risque 4 : Logique d'authentification incorrecte

L'implémentation d'une logique d'authentification est complexe et les LLM font régulièrement des erreurs dans ce domaine. Le cas le plus documenté est la vérification de signature JWT : un LLM peut générer du code qui décode un JWT sans vérifier sa signature, acceptant ainsi n'importe quel token forgé. D'autres erreurs fréquentes incluent des comparaisons de tokens non-constant-time (vulnérable au timing attack), l'absence de vérification de l'expiration du token, et des fonctions de hachage de mots de passe incorrectes (MD5 ou SHA-1 au lieu de bcrypt/Argon2).

```
 # Erreur IA classique : vérification JWT incorrecte
import jwt
payload = jwt.decode(token, options={"verify_signature": False})  # DANGER !

# Implémentation correcte
payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"],
                     options={"require": ["exp", "iat"]})
```

Risque 5 : Path traversal et SSRF dans le code généré

Les fonctions de lecture de fichiers et d'accès à des URL distantes générées par IA ne sanitisent pas toujours les entrées utilisateur. Un endpoint qui accepte un nom de fichier en paramètre et l'utilise directement dans une opération de lecture peut être vulnérable au path traversal (`../../../etc/passwd`). De même, un endpoint qui effectue des requêtes HTTP vers une URL fournie par l'utilisateur sans whitelist est vulnérable au SSRF (Server-Side Request Forgery), permettant d'accéder aux services internes du cloud (metadata endpoint AWS, GCP, Azure).

Risque 6 : Package hallucination et typosquatting

C'est peut-être le risque le plus insidieux et le moins intuitif. Les LLM peuvent "halluciner" des noms de packages — c'est-à-dire inventer des noms de bibliothèques plausibles mais inexistantes. Par exemple, un LLM pourrait suggérer `import request-handler-utils` (package inexistant) ou `pip install color-converter-pro`. Des chercheurs en sécurité ont démontré que ces noms hallucinés sont exploitables : en créant le package npm ou PyPI avec le nom exact suggéré par le LLM et en y injectant du code malveillant, un attaquant peut contaminer les projets qui suivent les suggestions IA sans vérifier l'existence du package. Des cas documentés de ce type ont été signalés sur npm et PyPI depuis 2024.

Risque 7 : Prompt injection via contexte de code contaminé

Dans les outils d'IA avec accès au codebase (Cursor, Claude Code), le contenu du code indexé fait partie du contexte du LLM. Un attaquant qui arrive à introduire dans le dépôt un fichier contenant des instructions déguisées en commentaires peut potentiellement influencer les

suggestions de code vers des patterns dangereux. Ce vecteur — la prompt injection via le contexte de code — est encore peu exploité en 2026 mais documenté dans les recherches en sécurité IA.

Étude de cas : vulnérabilité SQL injection introduite par Copilot

Un incident documenté en 2025 dans une application bancaire d'une fintech européenne illustre concrètement ces risques. L'équipe utilisait GitHub Copilot pour accélérer le développement d'une nouvelle API de gestion de comptes. Le développeur, sous pression temporelle, avait accepté une suggestion Copilot pour une fonction de recherche de transactions : la suggestion construisait la requête SQL par concaténation de chaînes en incluant le paramètre de recherche de l'utilisateur directement dans la chaîne SQL. Le SAST de la CI/CD (Checkmarx) n'était pas configuré pour ce nouveau microservice. La vulnérabilité n'a été découverte que lors d'un pentest externe 3 mois plus tard. L'impact potentiel : accès à l'ensemble de la base de données des transactions de tous les clients. La remédiation a nécessité 3 semaines de corrections et de tests de non-régression.

Pratiques sûres : du SAST au prompt engineering sécurisé

Intégration SAST dans la CI/CD

Le SAST (Static Application Security Testing) automatique est le contrôle compensatoire le plus efficace pour détecter les vulnérabilités introduites par le code IA. Semgrep (open source, règles communautaires) et Snyk Code (version commerciale avec règles avancées) s'intègrent facilement dans GitHub Actions, GitLab CI et Jenkins. CodeQL (GitHub, gratuit pour les dépôts publics) offre une analyse sémantique approfondie pour les langages principaux.

```yaml
# .github/workflows/security.yml - SAST avec Semgrep
name: Security Scan
on: [push, pull_request]
jobs:
  semgrep:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: returntocorp/semgrep-action@v1
        with:
          config: >-
            p/owasp-top-ten
            p/sql-injection
            p/secrets
            p/javascript
          auditResults: true
```

Configuration .cursorrules et copilot-instructions pour la sécurité

Les outils d'IA modernes permettent de fournir des instructions persistantes qui orientent les suggestions de code. Ces fichiers de configuration sont l'équivalent d'un "system prompt" pour votre assistant de code.

```
 # .cursorrules (ou .claude/CLAUDE.md pour Claude Code)
## Security Requirements — Mandatory for all generated code

### Input Validation
- ALWAYS use parameterized queries / prepared statements for database operations
- NEVER concatenate user input into SQL queries
- Validate and sanitize all user inputs before processing

### Authentication & Secrets
- NEVER hardcode API keys, passwords, tokens, or secrets in code
- ALWAYS use environment variables for sensitive configuration
- Use constant-time comparison for token/password verification
- JWT: always verify signature, expiration, and required claims

### Dependencies
- When suggesting external packages, use well-known packages with recent maintenance history
- Flag any package suggestion with version; ensure it is not known-vulnerable

### Error Handling
- NEVER expose stack traces, internal paths, or sensitive data in error messages
- Use generic error messages for authentication/authorization failures
```

Détection des packages hallucinés

Avant d'installer tout package suggéré par un LLM, vérifiez son existence et sa légitimité : consultez la page officielle sur npmjs.com ou pypi.org, vérifiez le nombre de téléchargements et la date de dernière publication, recherchez le dépôt GitHub correspondant. Des outils comme Socket.dev (npm) et pip-audit analysent automatiquement les packages avant installation pour détecter les packages suspects, les typosquatters et les packages avec des comportements malveillants connus.

Comparatif sécurité des outils : Cursor vs Copilot vs Codeium vs Claude Code

GitHub Copilot : Le plus utilisé (15M+ développeurs). Intégration IDE profonde. Microsoft a ajouté des filtres de contenu sécurité en 2024 qui réduisent les suggestions de code manifestement vulnérable. La fonctionnalité Copilot Autofix (intégrée à GitHub Advanced Security) propose des corrections automatiques pour les vulnérabilités détectées par CodeQL. Point faible : le contexte de code vu par Copilot est limité aux fichiers ouverts.

Cursor : Basé sur VS Code, Cursor offre une intégration LLM (Claude, GPT-4) avec accès à l'ensemble du codebase via indexation vectorielle. La configuration .cursorrules permet un contrôle fin des instructions de génération. Point d'attention sécurité : l'indexation du codebase est envoyée aux serveurs Cursor — à évaluer selon votre politique de données.

Codeium / Windsurf : Alternative plus respectueuse de la vie privée avec option d'hébergement on-premise. Moins de fonctionnalités avancées que Cursor mais données non envoyées aux serveurs pour les clients enterprise. Adapté aux environnements avec exigences de confidentialité du code source.

Claude Code : Approche terminal-first avec accès complet au système de fichiers. Les fichiers CLAUDE.md permettent de définir des règles de sécurité persistantes. Particulièrement efficace pour les tâches de refactoring sécurité et d'audit de code existant.

Cadre de gouvernance IA pour développeurs

La gouvernance de l'utilisation des outils de génération de code IA doit couvrir quatre axes. D'abord, la politique d'utilisation : définir quels outils sont autorisés, quels types de code peuvent être générés par IA (code interne vs code traitant des données sensibles), et les règles de revue. Ensuite, la formation : tous les développeurs utilisant des outils IA doivent être formés aux risques documentés dans ce guide, notamment le risque de package hallucination et la vérification des patterns d'authentification. Troisièmement, l'outillage : SAST automatique obligatoire dans la CI/CD pour tous les projets, gestion des dépendances avec audit automatique.

Enfin, l'audit : maintenir un inventaire des outils IA utilisés dans les projets, inclure la revue du code IA généré dans les processus de code review.

Pour les menaces liées aux LLM eux-mêmes, nos articles sur le red teaming IA et les prompt injections et la taxonomie des jailbreaks LLM complètent ce guide. Les attaques multimodales par prompt injection représentent un vecteur émergent. Le protocole MCP présente ses propres risques détaillés dans notre article sur la sécurité du Model Context Protocol.

Les référentiels officiels incluent l'OWASP Top 10 pour les applications LLM qui classe et documente les risques de sécurité spécifiques aux LLM, et le document NIST AI RMF (AI 100-1) qui fournit un cadre de gestion des risques IA applicable en entreprise.

FAQ — Questions fréquentes sur la sécurité du vibe coding

Peut-on utiliser les outils IA de génération de code dans des projets à données sensibles (santé, finance) ?

La réponse dépend des données transmises au LLM et des obligations réglementaires. En France, les données de santé (SNDS, DMP) sont soumises au référentiel HDS et ne peuvent pas transiter par des services cloud non-certifiés. Les données financières peuvent être soumises à des obligations PCI-DSS ou DORA qui limitent l'externalisation des traitements. Sur le plan pratique, les outils d'IA envoyant le code source à des serveurs tiers pour inférence sont problématiques si ce code contient des schémas de base de données, des clés de chiffrement ou des logiques métier sensibles. Les solutions d'hébergement on-premise (Ollama avec CodeLlama, Codeium Enterprise on-prem, GitHub Copilot for Business avec configuration réseau privée) permettent de conserver les données en interne.

Le SAST peut-il détecter toutes les vulnérabilités introduites par le code IA ?

Non, le SAST a des limites bien documentées. Il est efficace pour détecter les patterns connus de vulnérabilités (SQL injection, XSS, path traversal, secrets hardcodés) via des règles basées sur

l'analyse syntaxique et sémantique du code. En revanche, il ne détecte pas les vulnérabilités logiques (une logique d'autorisation incorrecte sémantiquement mais syntaxiquement valide), les problèmes de configuration d'infrastructure (SSRF via une URL configurable en dehors du code), ou les backdoors subtiles dans la logique métier. Le SAST doit être complété par la revue de code humaine (peer review avec une checklist sécurité), les tests DAST sur les endpoints, et les tests de pénétration réguliers. La défense en profondeur reste nécessaire même avec un SAST robuste.

Comment former les développeurs aux risques du vibe coding sans freiner l'adoption des outils IA ?

L'approche la plus efficace est la formation pratique basée sur des cas réels plutôt que des formations théoriques génériques. Organisez des ateliers de "secure code review" où les développeurs analysent du code réellement généré par les outils IA qu'ils utilisent et identifient les vulnérabilités. Intégrez des challenges de type CTF (Capture The Flag) avec du code IA vulnérable à exploiter — l'expérience de l'attaque reste le meilleur vecteur pédagogique. Montrez concrètement comment le SAST détecte (ou ne détecte pas) les vulnérabilités, et comment les .cursorrules/copilot-instructions peuvent améliorer la qualité sécurité des suggestions. L'objectif est de créer des développeurs qui font confiance aux outils IA avec discernement, pas de créer une méfiance qui freinerait l'adoption.

Synthèse et perspectives 2026

Les techniques et recommandations présentées dans ce guide s'inscrivent dans un contexte de menaces en constante évolution. La cybersécurité offensive et défensive sont deux faces d'une même médaille : comprendre les mécanismes d'attaque est indispensable pour construire des défenses robustes et résilientes face aux acteurs malveillants les plus sophistiqués.

Pour les équipes sécurité, l'enjeu de 2026 est double : maintenir une veille continue sur les nouvelles techniques publiées par la communauté de recherche (CVE, exploit-db, GitHub, Secrech, SSTIC) tout en assurant le durcissement progressif de l'infrastructure existante. Le référentiel MITRE ATT&CK reste le fil conducteur le plus efficace pour structurer un

programme de détection et de réponse face aux tactiques, techniques et procédures des groupes APT ciblant les secteurs critiques.

La formation continue des équipes, la simulation régulière d'incidents (exercices tabletop, exercices Red/Blue/Purple Team), et l'automatisation des tâches répétitives via des outils SOAR constituent les piliers d'une organisation cyber mature. Les organisations qui investissent dans ces trois axes démontrent systématiquement de meilleures métriques de détection et de réponse (MTTD et MTTR réduits de 40% en moyenne selon les benchmarks sectoriels) face aux incidents de sécurité.

Sources et références

ANSSI — Guide de sécurité des systèmes d'IA

MITRE ATLAS — Matrice des menaces adversariales IA

OWASP LLM Top 10
