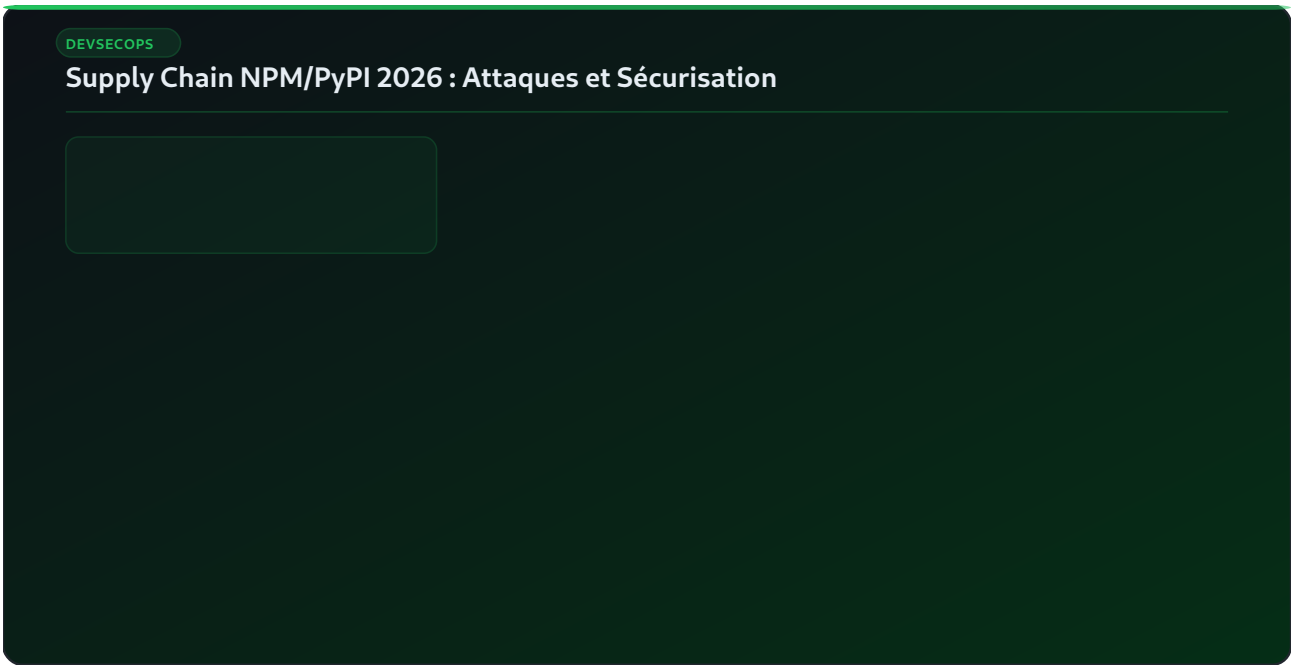
Supply Chain Attacks NPM/PyPI 2026 : Techniques et Sécurisation

Attaques supply chain NPM et PyPI 2026 : typosquatting, dependency confusion, SLSA et politiques ANSSI pour sécuriser les dépendances open source.

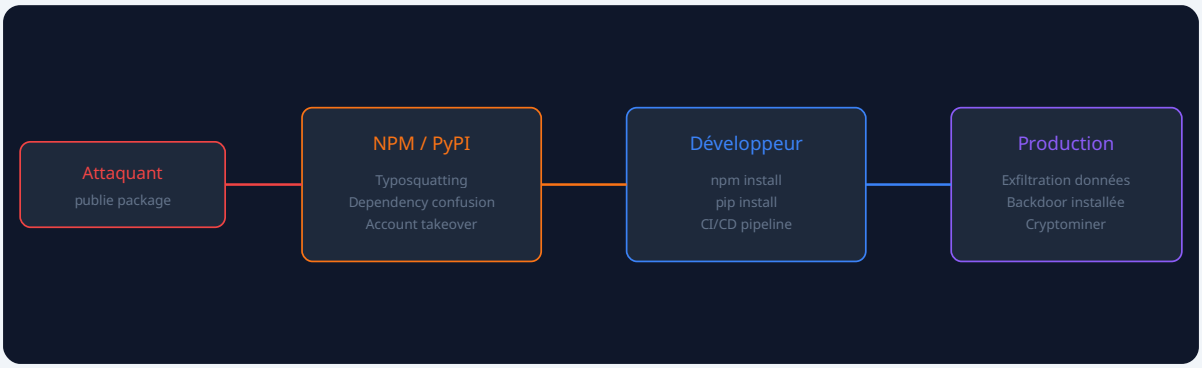
Les attaques de la chaîne d'approvisionnement logicielle (software supply chain attacks) ciblant les registres de packages NPM (Node Package Manager) et PyPI (Python Package Index) sont devenues l'une des menaces les plus sophistiquées de 2025-2026. En compromettant un package téléchargé par des millions de développeurs, un attaquant peut potentiellement infecter des dizaines de milliers d'organisations en une seule opération. L'attaque SolarWinds en 2020 a révélé l'ampleur du potentiel destructif des attaques supply chain ; depuis, les registres de packages open source sont devenus une cible prioritaire des groupes APT et des cybercriminels. OpenSSF (Open Source Security Foundation) a documenté en 2025 plus de 50 000 packages malveillants retirés de NPM et PyPI, contre 12 000 en 2023 — une multiplication par quatre qui reflète à la fois l'augmentation des attaques et l'amélioration de la détection. Notre équipe a analysé lors d'un audit pour une startup fintech parisienne un incident où un package NPM de 34 stars téléchargé 2 000 fois par jour exfiltrait silencieusement les variables d'environnement (incluant les clés AWS et les tokens d'API) vers un serveur contrôlé par l'attaquant. Le package avait été publié 6 semaines avant la détection, et son code malveillant était encapsulé dans une version minifiée d'une bibliothèque utilitaire légitime que personne n'avait auditée. Ce guide documente les techniques d'attaque supply chain contre NPM et PyPI en 2026, les outils de détection disponibles, et les bonnes pratiques pour sécuriser la chaîne d'approvisionnement logicielle de votre organisation.



NPM (JavaScript/Node.js) compte plus de 2,5 millions de packages avec 30 milliards de téléchargements mensuels. PyPI (Python) compte 500 000 packages avec 4 milliards de téléchargements mensuels. Ces chiffres illustrent l'ampleur de la surface d'attaque : un seul package populaire compromis peut propager du code malveillant vers des millions de projets. La chaîne de dépendances est le vecteur d'amplification : un projet peut dépendre directement de 50 packages, mais indirectement (via les dépendances de dépendances) de 500 à 2 000 packages — chacun étant un vecteur potentiel d'attaque.

La **confiance implicite** accordée aux packages open source est le facteur clé : les développeurs installent `npm install package-name` ou `pip install package-name` sans lire le code, sans vérifier l'intégrité, et sans auditer les nouvelles versions. Cette confiance implicite est exploitée par les attaquants qui publient des packages légitimes en apparence ou qui compromettent des packages légitimes existants. En France, le CERT-FR a émis plusieurs alertes spécifiques sur des campagnes supply chain ciblant des développeurs français, notamment dans le secteur financier et les startups technologiques.

Vecteurs d'attaque Supply Chain NPM/PyPI



Technique 1 : Typosquatting — Imitation de Packages Populaires

Le **typosquatting** est la technique la plus simple et la plus utilisée : l'attaquant publie un package avec un nom très similaire à un package populaire légitime, exploitant les fautes de frappe fréquentes des développeurs. Des exemples documentés : `requestss` (vs `requests`), `coloramma` (vs `colorama`), `python-dateutils` (vs `python-dateutil`). Ces packages malveillants s'installent sans avertissement et exécutent leur code malveillant lors de l'installation (dans `setup.py` pour PyPI ou dans `package.json` scripts install pour NPM).

⚡ **Retour terrain**

La dette de sécurité dans les dépendances npm/pip/Maven est le vecteur sous-estimé numéro un dans les projets DevSecOps que j'accompagne. Pour un projet Node.js moyen, `npm audit` remonte systématiquement entre 50 et 300 vulnérabilités — dont la grande majorité sont dans des dépendances transitives, pas dans les dépendances directes. La

règle que je recommande : 0 CVE critique dans les dépendances directes, et un processus de revue mensuelle des dépendances transitives critiques.

La détection du typosquatting est gérée par les équipes de sécurité NPM et PyPI via des algorithmes de similarité qui signalent les nouveaux packages ressemblant à des packages populaires. Cependant, la détection n'est pas systématique et les attaquants adaptent continuellement leurs techniques pour éviter la détection automatique. Des outils comme **Confused** (développé par Alex Birsan) et **Checkmarx Supply Chain** permettent aux organisations de vérifier leurs dépendances contre les packages typosquattés connus.

Technique 2 : Dependency Confusion — L'Attaque la Plus Dangereuse

La **Dependency Confusion**, découverte et documentée par le chercheur Alex Birsan en 2021, exploite le comportement par défaut de NPM et pip : quand ces outils cherchent un package, ils consultent d'abord le registre public (npmjs.com, pypi.org) AVANT les registres privés configurés par l'organisation. Si une organisation utilise un package privé nommé `company-internal-utils` dans son registre Nexus ou Artifactory, et qu'un attaquant publie un package public du même nom avec un numéro de version plus élevé, les builds automatisés de l'organisation téléchargeront le package malveillant public au lieu du package privé légitime.

Cette attaque est particulièrement dévastatrice car elle cible les environnements CI/CD qui exécutent les builds automatiquement et qui ont souvent accès aux secrets (tokens API, clés cloud, credentials de base de données) nécessaires pour les déploiements. Alex Birsan a démontré cette attaque avec succès contre Apple, Microsoft, Netflix, Shopify, et des dizaines d'autres grandes organisations, recevant des primes bug bounty totalisant plus de 130 000 dollars. Depuis, des corrections ont été déployées dans les gestionnaires de packages (flag `--private` dans pip, scoping npm), mais la surface d'attaque reste large dans les organisations qui n'ont pas revu leur configuration après 2021.

```
 # Detection de la vulnerability Dependency Confusion dans un projet Python
# Lister les packages internes qui pourraient etre confondus avec des packages publics

# 1. Lister les packages dans requirements.txt
grep -v "^#\|^-r\|^$" requirements.txt | cut -d'==' -f1 | cut -d'>=' -f1

# 2. Verifier si ces packages existent sur PyPI public
python3 -c "
import requests, sys

packages = open('requirements.txt').read().splitlines()
for pkg in packages:
    pkg_name = pkg.split('==')[0].split('>=')[0].strip()
    if not pkg_name or pkg_name.startswith('#'): continue
    try:
        r = requests.get(f'https://pypi.org/pypi/{pkg_name}/json', timeout=5)
        if r.status_code == 404:
            print(f'RISQUE Dependency Confusion: {pkg_name} n\\'existe pas sur PyPI public'
        else:
            data = r.json()
            print(f'OK: {pkg_name} v{data[\"info\"][\"version\"]} existe sur PyPI')
    except: pass
"

# 3. Corriger : utiliser le prefixe de scope npm ou pip config pour isoler
# Dans .npmrc : @company:registry=https://nexus.company.com/npm/
# Dans pip.conf : index-url = https://nexus.company.com/pypi/simple (sans fallback public)
```

Technique 3 : Account Takeover de Mainteneurs

La compromission du compte d'un mainteneur de package populaire permet à l'attaquant de publier une nouvelle version malveillante du package légitime, qui sera automatiquement téléchargée par tous les utilisateurs lors de leur prochain `npm update` ou `pip install --upgrade`. C'est la technique utilisée dans l'incident **event-stream** en 2018 (NPM, 2 millions de téléchargements/semaine) et l'incident **PyTorch nightly** en 2022 (package légitime remplacé par une version malveillante).

La défense côté développeur consiste à vérifier les nouvelles versions des packages critiques avant de les mettre à jour (audit du changelog et du diff de code), et à utiliser le *version pinning* (fixer les versions exactes dans `package-lock.json` ou `requirements.txt` avec les hashes SHA256 des packages). Le version pinning avec hash garantit que pip et npm n'installeront que le package dont le hash correspond à celui attendu — une nouvelle version malveillante publiée par un compte compromis aurait un hash différent et serait rejetée.

Technique d'attaque	Vecteur	Impact typique	Contre-mesure principale
Typosquatting	Faute de frappe développeur	Exfiltration secrets env	Audit dépendances, SCA tools
Dependency Confusion	Registre privé/public	Backdoor CI/CD	Scope npm, pip index isolé
Account Takeover	Phishing/MFA faible	Package légitime trojanisé	Hash pinning, OIDC publishing
Package abandonment	Rachat de domaine expir	Redirect vers malware	Audit packages dépréciés
Malicious install scripts	postinstall NPM hooks	Exec code au npm install	--ignore-scripts flag
Protestware	Mainteneur malveillant	Sabotage code produit	Revue code avant update

Technique 4 : Malicious Install Scripts dans NPM

NPM permet aux packages de définir des **scripts de cycle de vie** (preinstall, install, postinstall) qui s'exécutent automatiquement lors de l'installation. Ces scripts ont accès aux variables d'environnement du shell, aux fichiers du système, et au réseau — ce qui en fait un vecteur d'exfiltration idéal. Un package malveillant peut exfiltrer les variables d'environnement (qui contiennent souvent des tokens AWS, des clés API, des credentials de base de données) vers un serveur externe en quelques millisecondes, avant même que le développeur n'utilise le package.

La défense contre les malicious install scripts est d'utiliser le flag `--ignore-scripts` lors de l'installation (`npm install --ignore-scripts`), qui désactive l'exécution des scripts de cycle de

vie. Cependant, certains packages légitimes nécessitent ces scripts pour leur fonctionnement (notamment les packages avec des extensions natives en C++). Une approche équilibrée consiste à auditer les scripts de cycle de vie de chaque nouveau package avant installation (`npm view package-name scripts`) et à n'exécuter les scripts que pour les packages audités et de confiance. En environnement CI/CD, l'utilisation systématique de `npm ci --ignore-scripts` (au lieu de `npm install`) garantit une installation basée sur le `package-lock.json` sans exécution de scripts, ce qui est la bonne pratique recommandée pour tous les pipelines de build automatisés qui ne nécessitent pas de recompilation de modules natifs.

Technique 5 : Protestware et Sabotage Délibéré par les Mainteneurs

Le **protestware** est une forme d'attaque supply chain particulièrement insidieuse : le mainteneur légitime d'un package populaire modifie délibérément son code pour protester contre une situation politique ou sociale. L'incident le plus médiatisé est celui du package **node-ipc** (1 million de téléchargements par semaine) en mars 2022 : son mainteneur a ajouté un code qui, détectant une adresse IP russe ou biélorusse, effaçait le contenu des fichiers sur le système. Ce code malveillant délibéré a été intégré dans des dizaines de milliers de projets utilisant Vue CLI.

La défense contre le protestware et le sabotage par les mainteneurs est plus difficile car elle implique de maintenir une confiance dans des individus. Les mesures pratiques incluent : version pinning strict pour les packages critiques, review du code source avant mise à jour majeure pour les packages à fort impact, et utilisation d'un registry privé miroir (Nexus, Artifactory) qui permet de contrôler exactement quelles versions de quels packages sont disponibles dans l'environnement de développement.

SCA (Software Composition Analysis) : Les Outils d'Audit des Dépendances

Les outils de **Software Composition Analysis (SCA)** analysent automatiquement les dépendances d'un projet pour détecter les vulnérabilités connues (CVE), les packages malveillants connus, et les packages à risque élevé (mainteneur unique, pas de MFA activé, historique de modifications suspectes). Les principaux outils SCA en 2026 :

Snyk Open Source : intégration CI/CD native (GitHub Actions, GitLab CI), base de données de vulnérabilités propriétaire enrichie, et détection des packages malveillants via son équipe de recherche. Snyk propose une version gratuite avec des limites sur le nombre de projets. **OWASP Dependency-Check** : outil open source qui analyse les dépendances contre la NVD (National Vulnerability Database) du NIST. Moins complet que les solutions commerciales mais gratuit et maintenu par OWASP. **Socket.dev** : spécialisé dans la détection comportementale des packages malveillants (analyse du comportement déclaré : un package utilise-t-il le réseau ? le système de fichiers ? des APIs inhabituelles ?). Particulièrement efficace contre les attaques zero-day non encore référencées dans les CVE.

```
 # Integration Snyk dans GitHub Actions (exemple)
name: Security Scan
on: [push, pull_request]

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
      - name: Install dependencies
        run: npm ci --ignore-scripts  # Desactive les scripts d'install
      - name: Run Snyk audit
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
        with:
          args: --severity-threshold=high

# Pour PyPI - audit avec pip-audit
pip install pip-audit
pip-audit --requirement requirements.txt --format json --output /tmp/audit-result.json
```

SLSA Framework : Software Artifacts Attestation

Le framework **SLSA (Supply-chain Levels for Software Artifacts)**, développé par Google et maintenu par l'OpenSSF, définit des niveaux de sécurité pour les artefacts logiciels (Level 1 à Level 3). SLSA Level 3, le plus élevé, exige que le processus de build soit cryptographiquement attesté, hermétique (sans accès réseau pendant le build), et reproductible (le même source produit exactement le même artefact). Ces garanties permettent de vérifier qu'un package publié a bien été construit depuis le code source déclaré, sans modification pendant le processus de build.

NPM implémente maintenant le *Provenance Attestation* (via le flag `--provenance` lors de la publication) qui inclut une attestation SLSA signée cryptographiquement dans le package. Les

consommateurs peuvent vérifier cette attestation pour s'assurer que le package a été construit depuis le dépôt GitHub déclaré via les GitHub Actions officielles. PyPI déploie un mécanisme similaire via le projet **Sigstore/Cosign**. Ces mécanismes d'attestation sont encore en adoption mais représentent l'avenir de la sécurité supply chain pour les registres publics.

La vérification de la provenance côté consommateur se fait avec la CLI npm : `npm audit signatures` vérifie que les signatures des packages installés correspondent aux packages publiés sur le registre npm officiel, détectant ainsi les tentatives de substitution de packages en transit (attaques "on the wire"). Pour PyPI, `pip install --require-hashes` en combinaison avec un fichier `requirements.txt` généré par `pip-compile --generate-hashes` offre une garantie équivalente. Ces mécanismes transforment l'installation de packages d'un acte de confiance aveugle en une vérification cryptographique systématique.

Technique 6 : Package Abandonment et Hijacking de Domaine

Le **package abandonment** est une menace silencieuse : des milliers de packages NPM et PyPI sont maintenus activement par un développeur, puis abandonnés quand celui-ci change de projet ou d'employeur. Si ce développeur n'a pas transféré la propriété du package à une organisation, un attaquant peut tenter de récupérer le compte via un account takeover (phishing, réutilisation de mot de passe depuis un autre breach) et publier une version malveillante. Des recherches académiques publiées en 2025 montrent que plus de 80 000 packages NPM ont un mainteneur unique et n'ont pas reçu de mise à jour depuis plus de 2 ans — constituant une surface d'attaque considérable.

Le **domain expiry hijacking** est une variante : certains packages NPM utilisent un email de mainteneur associé à un domaine personnalisé. Si ce domaine expire et est racheté par un attaquant, celui-ci peut déclencher une récupération de mot de passe vers l'email du domaine racheté et prendre le contrôle du compte npm. Cette attaque est particulièrement difficile à détecter car elle ne laisse aucune trace d'intrusion dans les systèmes du mainteneur original. La défense est d'utiliser des adresses email de mainteneur sur des domaines permanents (GitHub, Gmail, domaine d'organisation) plutôt que sur des domaines personnels potentiellement éphémères.

Attaques Ciblant les Pipelines CI/CD : Le Point Névralgique

Les environnements CI/CD (GitHub Actions, GitLab CI, Jenkins, CircleCI) sont le point d'injection le plus dangereux pour les supply chain attacks car ils ont accès à tous les secrets nécessaires aux déploiements : credentials AWS/GCP/Azure, tokens de registres Docker, clés de signature de code, certificats de déploiement. Un package malveillant exécuté dans un pipeline CI/CD peut exfiltrer l'intégralité de ces secrets en quelques secondes.

L'incident **CircleCI breach de janvier 2023** illustre ce risque : des secrets stockés dans CircleCI ont été compromis via un logiciel espion sur le poste d'un employé CircleCI, permettant l'accès aux secrets de milliers de projets clients. La réponse recommandée est de ne pas stocker de secrets à longue durée de vie dans les environnements CI/CD, mais d'utiliser des mécanismes d'authentification dynamiques : **OIDC (OpenID Connect)** permet à GitHub Actions d'obtenir des tokens AWS/GCP temporaires directement via une fédération d'identité, sans secret statique stocké. Ces tokens expirent en quelques minutes et ne peuvent pas être réutilisés en dehors du contexte du workflow qui les a demandés.

```yaml
 # Exemple GitHub Actions avec OIDC (pas de secret statique AWS)
name: Deploy to AWS
on: push

permissions:
  id-token: write  # Requis pour OIDC
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      # Authentification OIDC - pas de secret AWS necessaire
      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionsRole
          aws-region: eu-west-3
          # Pas de access-key-id ni secret-access-key !

      - name: Deploy
        run: aws s3 sync ./dist s3://my-bucket/
```

Score de Risque des Packages : OpenSSF Scorecard

L'**OpenSSF Scorecard** est un outil automatisé qui évalue le score de sécurité d'un dépôt open source sur une dizaine de critères : présence de policy de sécurité, activation de la signature de code, utilisation de GitHub Actions sécurisées, présence de tests de sécurité automatisés, etc. Des scores OpenSSF sont calculés automatiquement pour des millions de dépôts GitHub et accessibles via l'API OpenSSF ou via le badge Scorecard dans les README. Un package avec un score OpenSSF inférieur à 5/10 présente des risques de sécurité structurels qui doivent être considérés avant son adoption dans un projet critique.

La *deps.dev* (Google) est un autre outil de référence : elle agrège les métadonnées de sécurité de NPM, PyPI, Maven, et Go Modules pour fournir une vue unifiée des risques de chaque package

(score de criticité, dépendances, vulnérabilités, score de sécurité OpenSSF). L'API publique de deps.dev peut être intégrée dans les workflows CI/CD pour évaluer automatiquement les nouveaux packages ajoutés à un projet.

Réponse à Incident Supply Chain : Que Faire Quand un Package est Compromis

Quand un package utilisé par votre organisation est identifié comme compromis (via une alerte CERT-FR, une annonce OpenSSF, ou une détection interne), la réponse doit être rapide et structurée. Les étapes immédiates sont : identifier toutes les applications et environnements utilisant le package compromis (SBOM si disponible, sinon recherche dans les fichiers de dépendances), vérifier si les versions installées sont affectées (toutes les versions ou seulement certaines), et si affecté, analyser si le code malveillant a pu s'exécuter (le package était-il dans le path d'exécution d'une application déployée ou seulement en dépendance de développement ?).

Si le package malveillant a été exécuté en production, une investigation de sécurité complète est nécessaire : vérifier les logs réseau pour des connexions sortantes inhabituelles (exfiltration), analyser les variables d'environnement accessibles à l'application (quels secrets étaient exposés ?), et selon le payload malveillant, évaluer si des backdoors ont été installées. La mise à jour ou la suppression du package et un redémarrage des applications n'est pas suffisant si du code malveillant a déjà été exécuté — les secrets exposés doivent être révoqués et rotés même si aucune exploitation n'est confirmée.

Gestion des Dépendances Transitives : La Face Cachée du Risque

La grande majorité des incidents supply chain exploitent des **dépendances transitives** — des packages que le projet n'utilise pas directement mais qui sont installés automatiquement car des dépendances directes les requièrent. Un projet React minimal peut dépendre directement de 50

packages, mais indirectement de 800 à 1 500 packages via les arbres de dépendances. Surveiller 1 500 packages est hors de portée humaine — c'est le rôle des outils SCA automatisés.

La réduction de la surface d'attaque des dépendances transitives passe par plusieurs pratiques : privilégier les packages avec peu de dépendances propres (une dépendance "zero-dep" n'introduit aucune dépendance transitive), utiliser `npm dedupe` pour éliminer les versions dupliquées, auditer régulièrement les packages qui n'ont pas de mise à jour depuis plus d'un an (indicateur de projet abandonné), et favoriser les packages maintenus par des organisations plutôt que par des individus uniques (moindre risque d'account takeover ou d'abandon). L'outil `npm ls --depth=10` produit l'arbre complet des dépendances transitives, tandis que `pip-tree` (PyPI) offre une visualisation équivalente pour les projets Python — des commandes précieuses pour comprendre pourquoi un package suspect se retrouve dans l'environnement d'un projet qui ne l'a pas explicitement référencé.

Une stratégie émergente pour les projets critiques est le **vendor/bundle** : copier le code source des dépendances critiques directement dans le dépôt du projet (dossier `vendor/`) plutôt que de les télécharger à chaque build depuis un registre externe. Cette approche, populaire dans l'écosystème Go depuis ses débuts, garantit que les builds sont reproductibles et non affectés par une modification du registre distant, au prix d'un dépôt plus volumineux et d'une charge de maintenance des mises à jour de sécurité entièrement transférée à l'équipe. Pour les composants avec une fréquence de mise à jour faible et un impact de sécurité élevé (bibliothèques cryptographiques, parseurs de format sensibles), le vendoring est une pratique défensive raisonnable.

Confiance Zéro dans les Dépendances : Principe du Moindre Privilège

L'approche **Zero Trust appliquée aux dépendances logicielles** consiste à ne jamais faire confiance implicitement à un package, même populaire, même maintenu par une organisation réputée. Cette approche se traduit par des pratiques concrètes : isoler les scripts d'installation dans des sandboxes (outil **deno** pour JavaScript, qui exige des permissions explicites pour l'accès réseau et filesystem), limiter les variables d'environnement exposées aux scripts d'installation (ne

pas avoir ses clés AWS disponibles pendant un `npm install`), et vérifier cryptographiquement les packages téléchargés.

Pour les environnements de build, le principe de moindre privilège implique que l'environnement CI/CD ne devrait avoir accès qu'aux secrets strictement nécessaires à l'étape courante du pipeline. Un job de build qui compile le code n'a pas besoin des credentials de déploiement — ces derniers ne devraient être disponibles que dans le job de déploiement qui suit la validation des tests. Cette segmentation des secrets dans le pipeline réduit l'impact d'un compromission supply chain à la phase où elle se produit, plutôt que d'exposer l'intégralité des secrets du pipeline à n'importe quel package malveillant qui s'exécute pendant le build.

Tendances Émergentes : LLM Package Hallucination et AI-Generated Malware

Une tendance préoccupante documentée depuis 2024 est l'**AI Package Hallucination** : les assistants IA (ChatGPT, Claude, Copilot) peuvent "halluciner" des noms de packages inexistants dans leurs réponses à des questions de programmation. Des attaquants ont commencé à surveiller ces hallucinations et à publier rapidement des packages réels portant ces noms inexistants mais recommandés par les IA, sachant que des développeurs vont les installer en suivant les recommandations de leur assistant IA. Cette technique combine la confiance accordée aux IA et la vitesse de publication des packages pour créer un vecteur d'attaque original qui court-circuite les défenses habituelles basées sur la réputation des packages.

La contre-mesure contre cette nouvelle technique est simple : ne jamais installer un package recommandé par une IA sans avoir d'abord vérifié son existence sur npmjs.com ou pypi.org, son score de sécurité OpenSSF, et son historique de publications. Un package qui n'existe que depuis quelques jours et n'a que quelques téléchargements, alors qu'une IA le recommande comme "bibliothèque standard", est un signal d'alarme fort. La formation des développeurs à cette nouvelle catégorie de risque est une priorité pour les équipes de sécurité en 2026.

Politique de Sécurité Supply Chain dans les Organisations Françaises

En France, les entreprises soumises à NIS 2 doivent adresser la sécurité de la chaîne d'approvisionnement logicielle dans leur politique de sécurité (Article 21.2.d sur la sécurité de la chaîne d'approvisionnement). Pour les éditeurs de logiciels et les organisations qui développent des produits logiciels distribués à des tiers, les exigences de sécurité supply chain sont encore plus strictes : la directive CRA (Cyber Resilience Act) européenne, en cours de transposition, imposera des exigences de security by design et de gestion des vulnérabilités pour les produits avec éléments numériques commercialisés dans l'UE.

La mise en place d'une **Software Bill of Materials (SBOM)** — inventaire complet des composants logiciels d'une application — est recommandée par l'ANSSI depuis 2024 et deviendra obligatoire sous CRA pour certaines catégories de produits. Des outils comme **Syft** (Anchore), **CycloneDX**, et **SPDX** permettent de générer automatiquement des SBOMs dans des formats standardisés qui peuvent être partagés avec les clients et les régulateurs. La SBOM permet une réponse rapide lors de la publication d'une vulnérabilité critique dans une dépendance : en quelques secondes, on identifie tous les produits et environnements utilisant le composant vulnérable.

Contexte France : CERT-FR Alertes et Référentiels ANSSI Supply Chain

L'ANSSI a publié en 2024 une note technique sur la sécurité de la chaîne d'approvisionnement logicielle qui recommande : la mise en place de registres privés miroir pour les packages critiques, l'audit des dépendances lors de chaque nouveau projet, la formation des développeurs aux risques supply chain, et l'intégration d'outils SCA dans les pipelines CI/CD. Le CERT-FR publie des alertes spécifiques quand des packages malveillants populaires sont identifiés — ces alertes doivent être intégrées dans les flux de veille sécurité des équipes de développement.

Pour les startups et PME du numérique, le dispositif **MonServiceSécurisé** de l'ANSSI propose un accompagnement gratuit qui inclut désormais un volet supply chain logicielle. La démarche recommandée est progressive : commencer par les dépendances directes des projets les plus

critiques, puis étendre progressivement l'audit aux dépendances transitives et aux environnements CI/CD. L'objectif n'est pas le zero-dependency (irréaliste dans l'écosystème actuel) mais de comprendre, documenter et maîtriser les risques de chaque composant dans le contexte de l'organisation.

Foire Aux Questions sur les Attaques Supply Chain NPM/PyPI

Comment vérifier si un package NPM ou PyPI est sûr avant de l'installer ?

Plusieurs vérifications rapides réduisent significativement le risque : vérifier l'âge du package (méfiance pour les packages de moins de 6 mois sans historique), le nombre de téléchargements (méfiance pour les packages très peu téléchargés qui prétendent être des utilitaires utiles), l'identité du mainteneur (un package majeur maintenu par un compte créé il y a 3 jours est suspect), et l'existence d'un dépôt GitHub actif avec des issues et des contributeurs réels. Côté technique, l'outil `npm audit` (intégré à NPM) et `pip audit` (pypi.org/project/pip-audit/) analysent les vulnérabilités connues. Socket.dev propose un badge de sécurité intégrable dans les dépôts GitHub qui signale automatiquement les comportements suspects des packages.

Doit-on utiliser un registre privé miroir pour tous les packages ?

Un registre privé miroir (Nexus, Artifactory, AWS CodeArtifact) offre plusieurs avantages au-delà de la sécurité : disponibilité garantie (indépendance vis-à-vis des pannes npmjs.com/pypi.org), conformité réglementaire (inventaire complet des packages utilisés), et contrôle des mises à jour (une nouvelle version d'un package n'est disponible pour les

développeurs qu'après validation). L'inconvénient est le coût de maintenance de l'infrastructure et la latence d'adoption des nouvelles versions. La recommandation pour les organisations traitant des données sensibles ou développant des produits à haute criticité est d'utiliser un registre privé avec validation avant promotion des nouvelles versions — une pratique standard dans les secteurs financier et défense en France.

Le Dependency Confusion affecte-t-il uniquement NPM et PyPI ?

Non, la Dependency Confusion affecte tous les gestionnaires de packages qui consultent des registres publics et privés en parallèle. Ruby Gems, Maven (Java), NuGet (.NET), Go Modules, et même Composer (PHP) sont vulnérables aux variantes de Dependency Confusion. Pour Go Modules spécifiquement, le vecteur est différent (les modules sont identifiés par leur URL GitHub/VCS, pas par un nom court) mais des variantes similaires existent. La vérification de la configuration des registres doit être réalisée pour tous les langages et gestionnaires de packages utilisés dans l'organisation, pas seulement pour NPM et PyPI.

Threat Intelligence sur les Campagnes Supply Chain Récentes

Plusieurs campagnes d'attaque supply chain majeures ont été documentées en 2025-2026, illustrant la diversité des tactiques et des cibles :

Campagne "PyNukeBay" (2025) : Une série de 200+ packages PyPI malveillants ciblant spécifiquement les développeurs Python travaillant dans le secteur financier. Les packages imitaient des bibliothèques de trading algorithmique populaires et exfiltraient les configurations de broker (clés API Binance, Interactive Brokers, etc.). La campagne a duré 4 mois avant d'être

détectée par l'équipe de sécurité PyPI grâce à une signature comportementale dans les scripts d'installation.

Campagne "NodeStealer" ciblant les tokens Meta (2025) : Des packages NPM malveillants déguisés en outils de développement Facebook/Meta exfiltraient les tokens de session des navigateurs installés sur les machines des développeurs. L'objectif final était de compromettre les comptes Facebook Business des organisations dont les développeurs avaient installé ces packages. Cette campagne illustre que la cible finale d'une supply chain attack n'est pas toujours les systèmes de production — parfois c'est l'identité en ligne du développeur.

Campagne contre la communauté open source cryptographie (2026) : Des packages Python imitant des bibliothèques cryptographiques légitimes (pycryptodome, cryptography) ont été publiés avec des noms similaires. Le code malveillant interceptait les appels aux fonctions de chiffrement et exfiltrait les clés et données chiffrées vers un serveur externe avant de les passer à la fonction légitime — un attaque "man-in-the-middle" au niveau de la bibliothèque cryptographique. Ce type d'attaque est particulièrement difficile à détecter car l'application continue de fonctionner normalement.

Méthodes de Détection Avancée : Analyse Comportementale des Packages

Au-delà de la simple vérification de signatures et de CVE connus, la **détection comportementale** des packages malveillants analyse ce qu'un package fait réellement lors de son exécution, indépendamment de la réputation de son nom ou de son mainteneur. Cette approche est nécessaire pour détecter les attaques zero-day et les packages compromis via account takeover (dont le mainteneur légitime a une bonne réputation).

Des outils comme **Socket.dev** analysent statiquement le code de chaque package pour détecter les patterns comportementaux suspects : accès aux fichiers en dehors du répertoire de travail, connexions réseau vers des IPs non documentées, lecture des variables d'environnement, exécution de code shell via `child_process.exec()` ou `os.system()`. Un package de formatage de texte qui lit les variables d'environnement et ouvre une connexion réseau est immédiatement suspect, indépendamment de sa popularité ou de l'historique de son mainteneur.

L'analyse dynamique (exécution du package dans une sandbox isolée avec surveillance du comportement réseau et système) est plus complète mais aussi plus coûteuse en ressources. Des plateformes comme **Any.run** et **Joe Sandbox** offrent des capacités d'analyse dynamique applicables aux packages NPM et PyPI, bien que leur usage systématique sur l'ensemble des dépendances d'un projet soit impraticable pour les équipes sans budget dédié.

Licence et Conformité : Le SBOM au Croisement de la Sécurité et du Légal

La **Software Bill of Materials (SBOM)** sert un double objectif : sécurité (inventaire des composants pour réponse rapide aux vulnérabilités) et conformité légale (audit des licences open source utilisées). Les licences open source ont des obligations différentes selon leur type : les licences permissives (MIT, Apache 2.0, BSD) permettent une utilisation commerciale sans contraintes majeures, tandis que les licences copyleft fortes (GPL v3, AGPL) peuvent imposer de distribuer le code source de l'application si elle utilise ces composants.

Des outils comme **FOSSA** et **Black Duck** (commerciaux) ou **OSS Review Toolkit** (open source) combinent l'audit de sécurité des dépendances et l'audit de licences dans un seul outil. Pour les organisations françaises qui développent des logiciels propriétaires, une dépendance GPL non détectée dans la base de code peut créer une obligation légale de publier le code source sous GPL — une situation que l'audit régulier de licences via SBOM permet d'éviter. Cette obligation légale peut avoir des conséquences commerciales significatives pour les éditeurs de logiciels, en particulier dans les secteurs réglementés comme la finance ou la santé où la propriété intellectuelle est un actif stratégique. La SBOM comme outil de gouvernance des licences est donc pertinente au-delà du seul aspect sécurité, et son adoption par les équipes juridiques et product management en complément des équipes sécurité maximise son retour sur investissement et justifie plus facilement le budget nécessaire à sa mise en place.

Sécuriser les Pipelines de Publication : Protection des Comptes Mainteneurs

Du côté des mainteneurs de packages open source, la sécurisation du compte de publication est la première ligne de défense contre la compromission de leur package. Les mesures recommandées par OpenSSF pour les mainteneurs NPM et PyPI incluent : **activation du MFA obligatoire** sur le compte npm/PyPI (rendu obligatoire par npm pour les packages populaires depuis 2023, en cours pour PyPI), **utilisation du Trusted Publishing** qui permet de publier depuis GitHub Actions sans stocker de token API (similaire à OIDC pour AWS), et **revue des organisations et équipes** qui ont accès au package pour s'assurer que des comptes obsolètes ne trainent pas.

La rotation régulière des tokens d'API npm et PyPI est également importante : un token de publication avec une longue durée de vie qui est compromis (via phishing, breach d'une autre plateforme, ou malware sur le poste du développeur) peut être utilisé pour publier une version malveillante sans que le mainteneur légitime le réalise pendant plusieurs jours. NPM propose maintenant la **granularity des tokens** (tokens limités à un seul package, tokens read-only, tokens avec expiration configurable) qui permettent d'appliquer le principe de moindre privilège aux opérations de publication.

Cas d'Usage : Audit d'un Projet Python Existant

Voici une procédure d'audit pratique pour évaluer la posture supply chain d'un projet Python existant, applicable en moins de 2 heures par un développeur avec des outils gratuits :

```
 # Audit supply chain Python - procedure complete
# 1. Installer les outils d'audit
pip install pip-audit safety

# 2. Audit des vulnerabilites CVE
pip-audit --requirement requirements.txt

# 3. Verification safety (base de donnees PyUP.io)
safety check --requirement requirements.txt

# 4. Generer un SBOM (CycloneDX format)
pip install cyclonedx-bom
cyclonedx-py --requirement requirements.txt --output sbom.json

# 5. Verifier les packages non utilises directement
pip install pipreqs
pipreqs . --print  # Compare les imports reels vs requirements.txt

# 6. Identifier les packages abandonnes
python3 -c "
import requests, json
from datetime import datetime

with open('requirements.txt') as f:
    pkgs = [l.strip().split('==')[0] for l in f if l.strip() and not l.startswith('#')]

for pkg in pkgs:
    r = requests.get(f'https://pypi.org/pypi/{pkg}/json', timeout=5)
    if r.status_code == 200:
        info = r.json()
        releases = list(info['releases'].keys())
        last_release = max(releases, key=lambda v: info['releases'][v][0]['upload_time'] if
        last_date = info['releases'][last_release][0]['upload_time'][:10] if info['releases
        print(f'{pkg}: derniere version {last_release} ({last_date})')
"
```

Formation et Culture DevSecOps : Intégrer la Supply Chain dans le Cycle de Développement

La sécurité de la supply chain logicielle ne peut pas être uniquement une responsabilité de l'équipe sécurité — elle doit être intégrée dans la culture de développement. Les développeurs sont les premiers à installer et utiliser les packages, et leur sensibilisation aux risques supply chain est le facteur déterminant de l'efficacité des contre-mesures techniques. Une formation d'une demi-journée couvrant les types d'attaques documentées dans cet article, les réflexes de vérification avant installation, et l'utilisation des outils d'audit automatisés est un investissement minimal à haute valeur ajoutée.

L'approche **DevSecOps** intègre les contrôles de sécurité supply chain à chaque étape du cycle de développement : lors de l'ajout d'une dépendance (review de la dépendance par un pair avant merge), lors du build (scan SCA automatique dans CI/CD), lors du déploiement (validation du SBOM), et en production (surveillance des comportements réseau des applications). Cette intégration fait de la sécurité supply chain un processus continu plutôt qu'un audit ponctuel, ce qui est beaucoup plus efficace face à une menace qui évolue en permanence.

La mesure la plus simple et la plus souvent oubliée reste la revue manuelle du code des nouvelles dépendances critiques avant leur première intégration dans un projet. Cinq minutes d'examen du fichier `setup.py` ou `package.json` d'un package que l'on n'a jamais utilisé auparavant — en cherchant des appels réseau, des lectures de variables d'environnement, ou des exécutions de shell — peuvent prévenir un incident majeur. Cette vigilance de base, combinée aux outils automatisés documentés dans cet article, constitue une posture défensive robuste face aux supply chain attacks de 2026. Le coût d'une culture de sécurité dans les équipes de développement est infime comparé au coût moyen d'un incident supply chain : selon les études OpenSSF, chaque incident supply chain majeur coûte en moyenne 1,2 million de dollars aux organisations touchées, incluant les coûts de réponse à incident, de remédiation, et de perte de productivité.

Ressources Supply Chain Security

SLSA Framework — Supply-chain Levels for Software Artifacts (OpenSSF)

MITRE ATT&CK T1195 — Supply Chain Compromise

Articles Connexes Sécurité Développement

Pour les aspects de sécurité des pipelines CI/CD qui consomment ces packages, notre article sur Bypass EDR/XDR Red Team 2026 couvre les techniques d'injection de code dans les environnements de build. Pour la détection des attaques supply chain en production via les logs système, notre guide Durcissement Active Directory 2026 inclut la surveillance des comportements anormaux des applications. Notre analyse Ransomware Triple Extorsion 2026 illustre comment une supply chain attack peut être le vecteur d'entrée initial d'une campagne ransomware. Pour la conformité NIS 2 liée à la supply chain, notre guide Golden SAML et ADFS 2026 couvre les risques d'identité dans les environnements fédérés qui intègrent des composants tiers.

À RETENIR

Points clés Supply Chain Attacks NPM/PyPI 2026

Typosquatting, Dependency Confusion, Account Takeover : les trois vecteurs principaux à auditer systématiquement dans les projets existants

Version pinning avec hash SHA256 : la contre-mesure la plus efficace contre le remplacement de packages légitimes

npm install --ignore-scripts : désactiver les scripts de cycle de vie NPM lors des installations automatisées

Registre privé miroir : la solution architecturale pour les organisations avec exigences de contrôle élevées

SBOM (Software Bill of Materials) : exigé sous CRA pour les produits numériques, recommendé par l'ANSSI pour tous

SCA tools dans CI/CD : Snyk, pip-audit, Socket.dev — au moins un outil automatique obligatoire sur les projets critiques

SLSA Provenance Attestation : vérifier les attestations lors de l'installation de packages depuis NPM avec provenance

Formation développeurs : la vigilance humaine reste la première ligne de défense contre les packages suspects