Sécurité API 2026 : GraphQL, REST, Serverless — OWASP Top 10

Points essentiels

- BOLA reste la vulnérabilité API n°1 de l'OWASP API Top 10 2023
- GraphQL expose introspection, query batching et DoS par profondeur de requête non limitée
- Les JWT mal validés (alg none, signature ignorée) ouvrent l'usurpation d'identité
- Les tests API doivent être automatisés en CI/CD et surveillés par WAF API-aware

À RETENIR

À retenir

BOLA reste la vulnérabilité API n°1 de l'OWASP API Top 10 2023

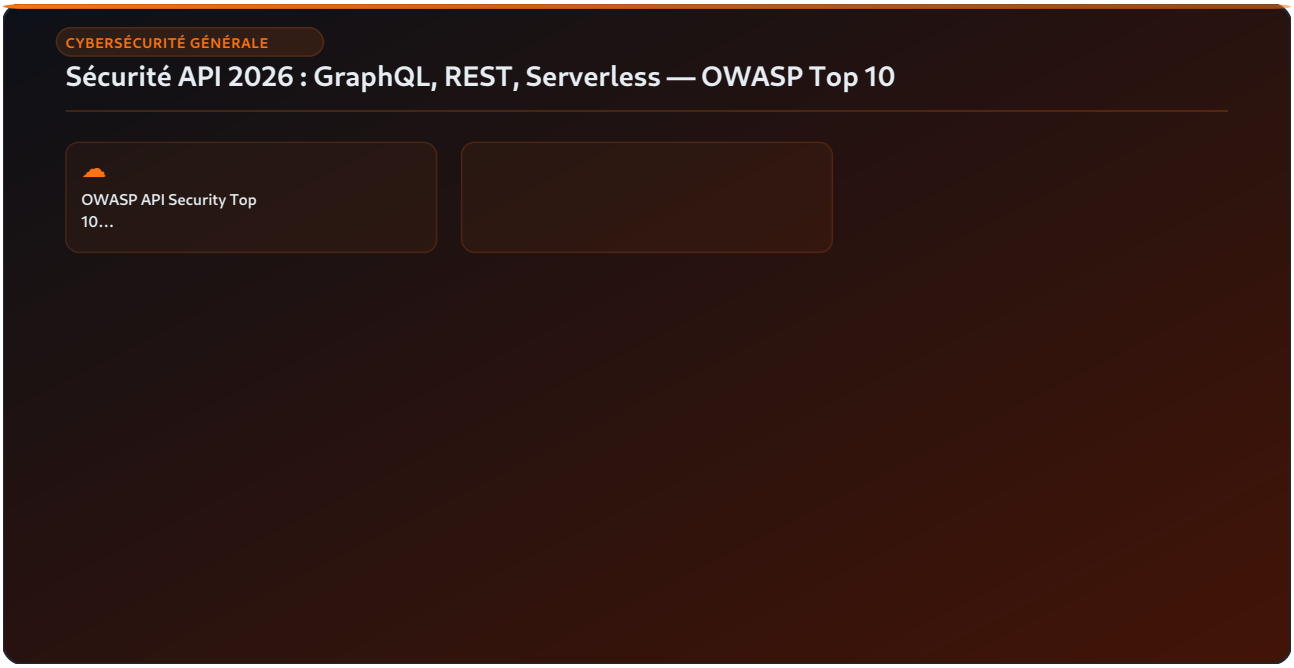
GraphQL expose introspection, query batching et DoS par profondeur de requête non limitée

Les JWT mal validés (alg none, signature ignorée) ouvrent l'usurpation d'identité

Les tests API doivent être automatisés en CI/CD et surveillés par WAF API-aware

La sécurité des APIs s'impose en 2026 comme la priorité défensive des entreprises numériques françaises. REST, GraphQL et fonctions serverless forment désormais la couche de communication universelle entre applications, microservices et partenaires : chaque endpoint exposé élargit une surface d'attaque que les pare-feux traditionnels ne couvrent plus. Les attaquants exploitent en priorité les failles d'autorisation au niveau objet (BOLA), les requêtes GraphQL imbriquées provoquant des dénis de service, et les permissions IAM trop permissives des fonctions cloud. Maîtriser l'api security graphql rest serverless suppose donc une approche

unifiée : inventaire exhaustif des endpoints, authentification forte, limitation de débit, validation stricte des schémas et supervision comportementale continue. Cet article détaille les menaces spécifiques à chaque architecture, les contrôles de l'OWASP API Security Top 10 et les mesures concrètes à déployer sans freiner vos équipes de développement.



L'OWASP API Security Top 10 2023 a introduit trois nouvelles catégories par rapport à la version 2019 : **API6:2023 Unrestricted Access to Sensitive Business Flows** (abus de flux métier légitimes à grande échelle — scraping massif, création automatique de comptes, fraude aux réservations), **API9:2023 Improper Inventory Management** (APIs oubliées, documentées nulle part, toujours en production — les "shadow APIs" découvertes lors des pentests dans 30% des environnements audités), et **API10:2023 Unsafe Consumption of APIs** (confiance aveugle dans les données reçues d'APIs tierces ou partenaires sans validation côté récepteur). Ces trois nouvelles catégories reflètent l'évolution des patterns d'attaque : les attaquants sophistiqués n'exploitent plus uniquement des vulnérabilités techniques (injections, auth bypass) mais aussi des failles de logique métier difficiles à détecter par les scanners automatiques.

Les catégories conservées de 2019 mais réordonnées incluent API1:2023 BOLA (Broken Object Level Authorization, anciennement IDOR) en première position — elle reste la vulnérabilité API la plus répandue et la plus exploitée dans les breaches réelles. Suivie par API2:2023 Broken Authentication, API3:2023 Broken Object Property Level Authorization (combine l'ancien Excessive Data Exposure et Mass Assignment), API4:2023 Unrestricted Resource Consumption (rate limiting, DoS), et API5:2023 BFLA (Broken Function Level Authorization, accès à des fonctions admin par des utilisateurs non privilégiés). La révision 2023 met ainsi davantage l'accent sur les vulnérabilités de logique d'autorisation — BOLA, BFLA, BOPLA — qui nécessitent une compréhension du modèle de données métier pour être détectées, en opposition aux vulnérabilités purement techniques détectables par scanning automatique.

BOLA : techniques de détection et exploitation

Le *BOLA* (Broken Object Level Authorization), aussi connu sous le nom d'IDOR (Insecure Direct Object Reference), est la vulnérabilité API la plus simple à exploiter et la plus catastrophique en termes d'impact : l'API ne vérifie pas que l'utilisateur authentifié a le droit d'accéder à l'objet demandé via son identifiant. Un utilisateur authentifié comme "user 12345" peut accéder aux données de "user 12346" simplement en modifiant l'identifiant dans la requête. Dans les APIs REST typiques, les identifiants concernés peuvent être des IDs numériques séquentiels (`/orders/99875`), des UUIDs (`/documents/a3f8...`), ou des slugs (`/users/jean.dupont`) — tous manipulables par un attaquant déterminé.

> ⚡ **Retour terrain**
>
> Dans mes missions d'audit, je rencontre régulièrement la même configuration à risque : des règles de firewall héritées depuis 5 à 10 ans, que personne n'ose supprimer par crainte de casser quelque chose. J'ai développé une méthode de nettoyage progressive — analyser les logs de connexion sur 90 jours, identifier les règles sans trafic, les désactiver sans

supprimer pendant 30 jours, puis valider avec les équipes métier. Sur un parc de 340 règles dans un groupe logistique, nous en avons supprimé 218 sans incident.

La détection des BOLA lors d'un audit API repose sur une technique simple mais systématique : créer deux comptes de test distincts (user A et user B), effectuer des requêtes authentifiées avec le user A en utilisant les identifiants d'objets appartenant au user B, et observer si l'API retourne les données du user B (BOLA) ou une erreur 403/404 (comportement attendu). L'outil Burp Suite avec le plugin "Autorize" automatise ce processus : il réplique chaque requête authentifiée de l'utilisateur A avec le token de l'utilisateur B en parallèle, et colore en rouge les réponses où B reçoit les mêmes données que A — révélant les BOLA de manière semi-automatique dans l'ensemble de l'application. **Le scanner de vulnérabilités API Nuclei dispose de templates BOLA qui testent automatiquement les endpoints API identifiés dans un fichier Swagger/ OpenAPI**, permettant une couverture systématique des endpoints sans exploration manuelle exhaustive.

Sécurité GraphQL : introspection, depth limit et batching

GraphQL présente une surface d'attaque spécifique très différente des APIs REST. Trois vecteurs sont particulièrement importants. Premièrement, l'*introspection* GraphQL : la requête `{ __schema { types { name fields { name } } } }` retourne le schéma complet de l'API — tous les types, tous les champs, toutes les mutations — permettant à un attaquant de cartographier entièrement l'API sans documentation préalable. L'introspection doit être désactivée en production (`introspectionEnabled: false` dans la plupart des frameworks GraphQL) ou restreinte aux utilisateurs authentifiés avec des droits de développement. En 2026, plus de 35% des APIs GraphQL en production exposent encore leur introspection sans restriction selon les données de PortSwigger Web Security.

Deuxièmement, le **query depth attack** : GraphQL permet des requêtes imbriquées récursivement (`{ user { friends { friends { friends { ... } } } } }`) qui peuvent épuiser les ressources serveur. Sans limite de profondeur (depth limit) configurée dans le serveur

GraphQL, une requête avec une profondeur de 20 niveaux peut générer des millions de résolutions de champs et faire tomber le serveur en DoS. Troisièmement, le **query batching** : GraphQL supporte nativement l'envoi de plusieurs requêtes en une seule requête HTTP via un array JSON, ce qui peut être utilisé pour contourner les rate limits configurés par nombre de requêtes HTTP (le rate limiter voit 1 requête HTTP mais le serveur exécute 50 opérations).

Tokens JWT : vulnérabilités et bonnes pratiques

Les *JWT* (JSON Web Tokens) sont le mécanisme d'authentification API le plus répandu, mais leur implémentation incorrecte génère des vulnérabilités critiques. Les attaques JWT les plus courantes incluent : l'algorithme "none" (certaines bibliothèques acceptent des JWT avec l'en-tête `"alg": "none"`, signifiant aucune vérification de signature — un attaquant peut créer des JWT arbitraires valides), la confusion RS256/HS256 (si le serveur accepte HS256 quand il attend RS256, un attaquant peut signer un JWT avec la clé publique RS256 utilisée comme clé symétrique HS256 — une confusion de type d'algorithme critiquement exploitable), et le jku/kid header injection (utilisation d'un header JWK Set URL contrôlé par l'attaquant pour faire valider le JWT par une clé publique attaquant).

L'outil `jwt_tool` (Python, open source) automatise les tests de vulnérabilités JWT : il teste les algorithmes none, la confusion RS256/HS256, la manipulation de claims, et l'injection de headers. **Pour la mise en production, les bonnes pratiques JWT imposent : signature avec RS256 ou ES256 (jamais HS256 en multi-tenant), validation stricte de l'algorithme côté serveur (whitelist explicite, jamais accepter l'algorithme du token), durée de vie courte (access token 15 minutes maximum, refresh token 24h), et revocation via blacklisting des JTI (JWT ID) lors du logout.** Le stockage des tokens côté client doit éviter le localStorage (vulnérable aux XSS) en faveur des cookies HttpOnly + SameSite=Strict, qui ne sont pas accessibles depuis le JavaScript et réduisent le risque de vol de token via injection XSS.

APIs serverless : AWS Lambda et Azure Functions — vecteurs spécifiques

Les APIs serverless basées sur AWS Lambda ou Azure Functions présentent des vecteurs d'attaque spécifiques qui s'ajoutent aux vulnérabilités API classiques. Le premier vecteur est la cold start exploitation : lors du démarrage à froid d'une fonction Lambda, le temps d'initialisation peut exposer des variables d'environnement ou déclencher des comportements inattendus sur les ressources partagées si plusieurs invocations simultanées initialisent en parallèle un état partagé (connexions DB, caches) de manière non thread-safe. Le deuxième vecteur est l'**event injection** : les fonctions serverless reçoivent leurs données via des événements (corps HTTP, messages SQS, objets S3) qui ne sont pas toujours validés avant d'être utilisés dans des opérations sensibles — si l'événement contient du SQL, du JSON, ou des chemins de fichiers construits depuis l'input utilisateur, les risques d'injection sont identiques aux vulnérabilités classiques.

Le troisième vecteur est spécifique aux APIs serverless à forte intégration cloud : **la sur-permission des fonctions Lambda/Azure Functions**. Chaque fonction dispose d'un rôle IAM (Lambda) ou d'une Managed Identity (Azure Functions) qui définit ses accès aux services cloud. Par commodité, les développeurs attribuent souvent des rôles trop larges (AmazonS3FullAccess, AzureContributorRole) à leurs fonctions, alors que chaque fonction ne devrait avoir que les droits minimaux nécessaires à son exécution. Un attaquant qui exécute du code via une vulnérabilité dans une fonction serverless récupère automatiquement les credentials cloud de la fonction et peut les utiliser pour accéder à l'ensemble des services autorisés par le rôle IAM — une escalade de privilèges vers les services cloud qui peut avoir un impact bien au-delà de la fonction compromise.

Rate Limiting et API Gateway : défenses de premier rang

Le rate limiting est la première ligne de défense contre les abus d'APIs : brute force d'authentification, scraping massif, abus de flux métier (création de comptes en masse, tentatives de fraude répétées). Un rate limiting efficace doit opérer à plusieurs niveaux : au niveau de l'API Gateway (AWS API Gateway, Azure API Management, Kong) pour les limites globales par IP ou

par client, au niveau applicatif pour les limites contextuelles par compte utilisateur (un utilisateur authentifié ne peut pas faire plus de 100 requêtes par minute même si l'IP n'est pas bloquée), et au niveau métier pour les limites sur les opérations sensibles (maximum 5 tentatives d'authentification par 15 minutes, maximum 3 créations de compte depuis la même IP par heure).

Le contournement du rate limiting basé sur les IPs est trivial pour un attaquant disposant de proxies résidentiels (IP rotation) ou d'un botnet distribué — ce qui explique pourquoi le rate limiting seul est insuffisant et doit être combiné avec des mécanismes comportementaux (fingerprinting du client, détection de patterns automatisés via user-agent, timing, séquence d'actions) et des CAPTCHAs adaptatifs sur les opérations sensibles. **Les solutions WAF API-aware comme Cloudflare API Shield, AWS WAF avec règles managed, ou DataDome (solution française) intègrent des capacités de détection comportementale des bots et des abus d'API** sans nécessiter de développement custom côté applicatif, réduisant le temps de déploiement d'une protection bot efficace de plusieurs mois à quelques jours.

Documentation OpenAPI/Swagger : vecteur de découverte et risque

La documentation OpenAPI (Swagger) est un double-tranchant pour la sécurité des APIs : elle améliore la qualité des intégrations et la testabilité, mais expose également la surface d'attaque complète à tout attaquant qui y accède. Un fichier Swagger exposé publiquement (`/swagger-ui.html` , `/api-docs` , `/openapi.json`) fournit instantanément à l'attaquant la liste de tous les endpoints, les paramètres attendus, les codes d'authentification requis, et parfois des exemples de données réelles copiés depuis l'environnement de production. L'inventaire des APIs exposées via leur documentation Swagger est l'une des premières étapes d'un pentest API : des outils comme `SwaggerHole` scannent automatiquement les URL candidates de documentation API sur un domaine cible.

La politique recommandée est de désactiver la documentation Swagger/OpenAPI en production (ou de la restreindre aux IPs du réseau interne ou VPN), de maintenir un fichier OpenAPI à jour en interne comme source de vérité pour les tests de sécurité automatisés, et d'utiliser des outils de génération de tests de sécurité depuis les spécifications OpenAPI — comme *schemathesis* (Python, open source) qui génère automatiquement des cas de test couvrant les cas aux limites,

les types invalides, et les injections pour chaque endpoint documenté dans le schéma. **Schemathesis intégré dans un pipeline CI/CD GitLab effectue automatiquement des tests de fuzzing basés sur le schéma OpenAPI à chaque déploiement**, détectant les régressions de sécurité API avant qu'elles n'atteignent la production — un DevSecOps API testing qui complète les tests d'intégration classiques sans nécessiter de scénarios de sécurité manuels.

Mass Assignment et Excessive Data Exposure

Le Mass Assignment est une vulnérabilité spécifique aux frameworks web modernes (Node.js/Express, Laravel, Django REST Framework, Spring Boot) qui permettent de mapper automatiquement le corps JSON d'une requête vers les propriétés d'un objet modèle — y compris des propriétés non destinées à être modifiées par l'utilisateur (isAdmin, role, createdAt, internalScore). Un attaquant envoie un corps JSON avec des champs supplémentaires (`{"email":"user@example.com", "role":"admin"}`) et si le framework mappe aveuglément tous les champs, le compte devient administrateur. Cette vulnérabilité, référencée en OWASP API3:2023 BOPLA, a causé plusieurs incidents médiatisés dont celui de GitHub en 2012 où un chercheur a ajouté une clé SSH au dépôt d'un autre utilisateur via Mass Assignment.

La prévention du Mass Assignment nécessite une validation stricte des corps de requêtes : utiliser des DTOs (Data Transfer Objects) avec des propriétés explicitement autorisées (allowlist) plutôt que de mapper directement vers les entités du modèle, et utiliser des bibliothèques de validation de schémas (Joi, Yup, pydantic, Bean Validation) qui définissent précisément les champs acceptables et leurs types. **L'OWASP ASVS (Application Security Verification Standard) v4.0, niveau 2, exige explicitement la validation de tous les inputs API via une allowlist de propriétés autorisées** — ce niveau de vérification est requis pour les applications traitant des données personnelles (RGPD) ou financières (PSD2) dans le contexte européen, rendant la conformité ASVS un objectif de sécurité pertinent pour les ETI françaises développant des APIs dans ces secteurs.

Outils d'audit API : Burp, Postman, Nuclei et 42Crunch

L'outillage d'audit de sécurité API a considérablement évolué depuis 2020. Burp Suite Professional (PortSwigger) reste la référence pour les pentests manuels avancés : son scanner actif intègre désormais des tests spécifiques aux APIs REST et GraphQL, et le plugin BApp Store fournit des extensions dédiées (Autorize pour BOLA, GraphQL Raider pour les injections GraphQL, JWT Editor pour les attaques JWT). Postman, bien que primairement un outil de développement, permet de créer des collections de tests de sécurité réutilisables via des scripts pre-request et post-response en JavaScript, testant l'authentification, les codes de retour d'erreur, et les contrôles d'accès.

Nuclei (ProjectDiscovery) est un scanner de vulnérabilités open source basé sur des templates YAML qui inclut plus de 300 templates spécifiques aux APIs REST et GraphQL, testant les endpoints d'administration exposés, les introspections GraphQL, les tokens par défaut, et les vulnérabilités connues des frameworks API courants (FastAPI, Spring Boot, Laravel). **42Crunch (solution française) est l'outil d'analyse statique de spécifications OpenAPI le plus complet du marché**, identifiant les failles de conception dans le schéma API avant même le déploiement — endpoints sans authentification, réponses exposant des données sensibles non nécessaires, opérations DELETE sans confirmation. L'intégration de 42Crunch dans les pipelines CI/CD permet de détecter les vulnérabilités API au moment de la review de code, quand le coût de correction est minimal.

OAuth 2.0 et OpenID Connect : pièges de configuration

OAuth 2.0 et OpenID Connect sont les standards d'authentification et d'autorisation des APIs modernes, mais leur flexibilité génère des nombreuses opportunités de misconfiguration. Les pièges les plus fréquents incluent : l'absence de validation du paramètre `state` (expose à des attaques CSRF sur le flow OAuth), l'autorisation du grant type "implicit flow" en 2026 (deprecated depuis OAuth 2.1 car il expose les access tokens dans l'URL), l'acceptation de redirect URIs avec wildcards (`https://myapp.fr/*` au lieu d'une URL exacte — permet de

rediriger les tokens vers des sous-domaines compromis), et la durée de vie excessive des refresh tokens (sans rotation ni révocation).

L'attaque de redirect URI spoofing est particulièrement efficace : si l'autorisation OAuth accepte `https://myapp.fr/callback` ET `https://myapp.fr/callback/../malicious` (traversal de chemin non filtré), un attaquant peut construire un lien de connexion légitime qui redirige le token vers une URL contrôlée. **La conformité OAuth 2.1 (draft final attendu mi-2026) impose le PKCE (Proof Key for Code Exchange) pour tous les flows public clients**, éliminant plusieurs vecteurs d'attaque historiques sur les flows d'autorisation OAuth dans les applications mobiles et SPA. Les organisations françaises développant des APIs avec OAuth pour des clients mobiles doivent planifier la migration vers OAuth 2.1 avec PKCE pour se conformer aux futures exigences de l'ENISA sur les protocoles d'authentification sécurisés dans le cadre de NIS 2.

Tableau OWASP API Security Top 10 2023 avec contre-mesures

Rang	Vulnérabilité	Impact typique	Contre-mesure principale
API1	BOLA (Broken Object Level Auth)	Accès données autres utilisateurs	Vérification ownership côté serveur
API2	Broken Authentication	Usurpation d'identité, brute force	MFA, rate limit, JWT sécurisé
API3	BOPLA (Object Property Level Auth)	Mass assignment, data exposure	DTOs avec allowlist propriétés
API4	Unrestricted Resource Consumption	DoS, coût cloud excessif	Rate limiting, payload size limit
API5	BFLA (Broken Function Level Auth)	Accès fonctions admin non autorisées	RBAC granulaire par endpoint
API6	Unrestricted Business Flow Access	Fraude, scraping, abus de flux	Détection comportementale, CAPTCHA
API7	Server Side Request Forgery (SSRF)	Accès services internes, IMDS	Whitelist URLs, blocage IMDS
API8	Security Misconfiguration	Swagger exposé, CORS trop large	Hardening, désactivation debug
API9	Improper Inventory Management	Shadow APIs, APIs non documentées	Inventaire APIs, API gateway centralisé
API10	Unsafe Consumption of APIs	Injection via API tierce, confiance aveugle	Validation stricte toutes réponses tierces

À RETENIR

Points clés à retenir

BOLA est la vulnérabilité API la plus répandue en 2026 — tester systématiquement la ségrégation des données entre comptes

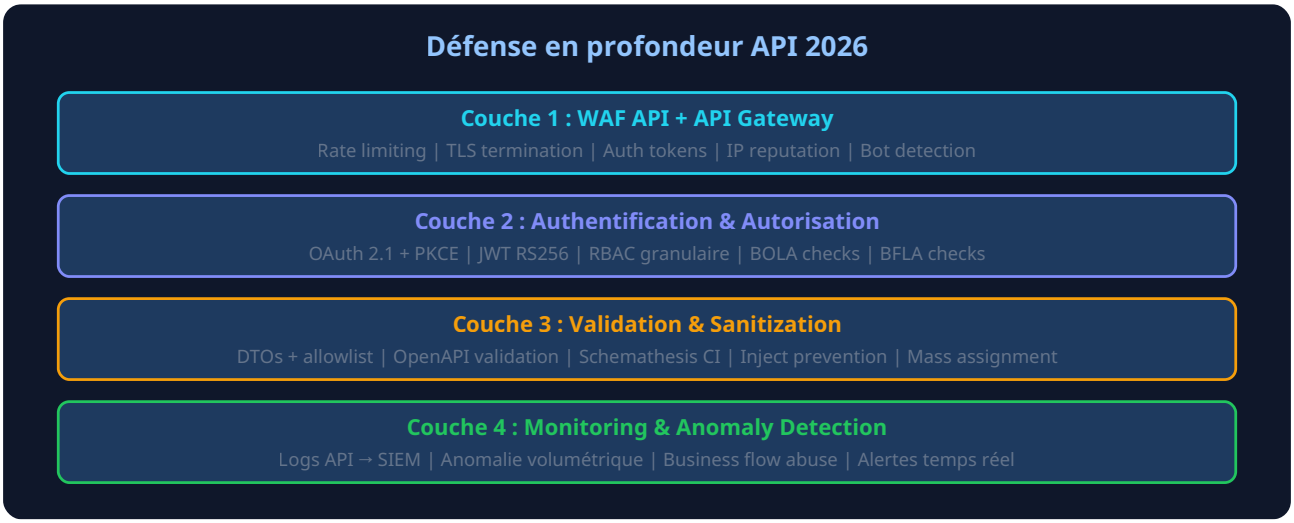
GraphQL nécessite une configuration spécifique : désactiver l'introspection en prod, limiter la profondeur des requêtes

Les APIs serverless héritent du risque IAM cloud — limiter les permissions au strict minimum par fonction Lambda/Azure Function

JWT : utiliser RS256 ou ES256, valider l'algorithme strictement, stocker en cookies HttpOnly en pas dans localStorage

42Crunch et schemathesis dans la CI/CD permettent de détecter les vulnérabilités API au plus tôt dans le cycle de développement

L'inventaire des APIs (OpenAPI à jour, API Gateway centralisé) est le prérequis de toute stratégie de sécurité API efficace



Ressources pour approfondir la sécurité des APIs

Le référentiel de base pour tout projet de sécurisation d'API est l'OWASP API Security Project (Top 10 2023, guides de test, cheatsheets). Le service d'audit API de 42Crunch propose une analyse gratuite de spécifications OpenAPI. Les articles connexes de cette série complètent la vue d'ensemble : attaques GraphQL et REST et sécurité API avec Burp et Nuclei pour les techniques offensives détaillées, et audit de l'API Microsoft Graph pour les cas d'usage spécifiques à l'environnement Microsoft 365 et Entra ID présent dans la majorité des ETI françaises.

Injection NoSQL et injection GraphQL

Les injections dans les APIs modernes ne se limitent pas au SQL classique : les APIs utilisant MongoDB, Elasticsearch, ou d'autres bases NoSQL sont vulnérables à des injections spécifiques à ces moteurs. L'injection MongoDB via API REST est particulièrement dangereuse : si l'API passe directement les paramètres de requête dans les filtres MongoDB sans sanitisation (`db.users.find({username: req.body.username})`), un attaquant peut injecter des opérateurs MongoDB (`{"$gt": ""}`) pour contourner l'authentification ou extraire tous les enregistrements. Cette technique, référencée en OWASP API en tant que variante d'injection (API8:2023 Security Misconfiguration), est testable avec des payloads simples : remplacer un paramètre string par `{"$ne": null}` dans le corps JSON de la requête.

Les injections GraphQL présentent des vecteurs spécifiques. Le **GraphQL injection via arguments** exploite des résolveurs qui construisent des requêtes SQL ou NoSQL depuis les arguments GraphQL sans paramétrage : `{ user(id: "1; DROP TABLE users; --") }` peut passer directement dans une requête SQL si le résolveur utilise la concaténation de chaînes plutôt que des requêtes paramétrées. Le **GraphQL directive injection** teste les directives personnalisées (@deprecated, @skip, @include) qui peuvent avoir des comportements non anticipés si leur implémentation est vulnérable à des injections de valeurs. L'outil Clairvoyance (open source) effectue une énumération de schéma GraphQL même quand l'introspection est désactivée, en inférant le schéma depuis les messages d'erreur des requêtes invalides — une technique qui contourne la protection par désactivation de l'introspection et que les équipes de sécurité doivent connaître pour ne pas surestimer la protection offerte par cette seule mesure.

CORS : misconfiguration fréquente et exploitation

La politique CORS (Cross-Origin Resource Sharing) contrôle quels domaines peuvent effectuer des requêtes cross-origin vers une API depuis un navigateur. Les misconfigurations CORS sont parmi les vulnérabilités API les plus fréquentes et peuvent permettre à un site malveillant de faire des requêtes authentifiées vers l'API cible depuis le navigateur d'une victime visitant le site

attaquant. La misconfiguration la plus critique est l'écho de l'origine (`Access-Control-Allow-Origin: $ORIGIN_HEADER`) combiné avec `Access-Control-Allow-Credentials: true` — toute origin est acceptée et les cookies d'authentification sont inclus, permettant une attaque CSRF sur n'importe quel endpoint API depuis n'importe quel domaine.

Le testing CORS lors d'un audit API consiste à envoyer une requête avec un header `Origin: https://evil.attacker.com` et à observer si la réponse inclut `Access-Control-Allow-Origin: https://evil.attacker.com` — si oui, l'API est vulnérable. L'outil `CORStest` (Python, open source) automatise ce test sur une liste de domaines. **La configuration CORS sécurisée impose une allowlist explicite des origines autorisées (jamais de wildcard `*` avec credentials)**, différenciée par environnement (dev: localhost uniquement, prod: domaines de production uniquement), et validée à chaque déploiement via un test automatisé dans la CI/CD. Les APIs publiques accessibles depuis des tiers doivent documenter explicitement leur politique CORS dans leur spécification OpenAPI pour que les développeurs clients comprennent les origines acceptées.

Gestion du cycle de vie des APIs : versioning et dépréciation

L'API9:2023 OWASP (Improper Inventory Management) reflète un problème organisationnel courant dans les ETI françaises : les APIs sont déployées mais rarement retirées, accumulant des versions obsolètes qui ne reçoivent plus les patches de sécurité mais restent accessibles en production. Un pentest API révèle souvent des versions v1 ou v2 oubliées accessibles via `/api/v1/` alors que la version officielle est v3, avec des vulnérabilités corrigées en v3 mais toujours présentes en v1 (BOLA, auth bypass, endpoints admin exposés). Cette accumulation de shadow APIs est le résultat d'une absence de politique formelle de dépréciation et de retrait des APIs.

La mise en place d'une politique de cycle de vie API impose : un versioning explicite dans les URLs (`/api/v1/`, `/api/v2/`) ou via des headers (`Accept: application/vnd.myapp.v2+json`), un calendrier de dépréciation annoncé publiquement (header `Sunset: Sat, 31 Dec 2026 23:59:59 GMT` sur les endpoints dépréciés), et une procédure de retrait incluant la vérification de l'absence de trafic résiduel avant la désactivation. **Un registre centralisé de toutes les APIs (internes et externes) est le prérequis organisationnel de l'inventaire API** — sans ce registre, il est

impossible de savoir quelles APIs existent, qui les utilise, et quand elles peuvent être retirées en toute sécurité. Ce registre, maintenu dans le portail développeur de l'API Gateway ou dans un outil dédié comme Backstage, est aussi le point d'entrée de la gouvernance API et du scan de sécurité automatisé via 42Crunch ou Nuclei à chaque nouvelle version.

Logging et monitoring des APIs : détecter les abus en production

La surveillance des APIs en production est indispensable pour détecter les abus qui passent à travers les contrôles préventifs : rate limiting contourné via des IPs distribuées, BOLA exploitée par un attaquant patient qui énumère les IDs un par un sur plusieurs jours, ou vol de données progressif via des requêtes qui restent chacune dans les limites autorisées mais s'accumulent pour représenter une exfiltration massive. Les logs d'API doivent capturer pour chaque requête : l'identifiant utilisateur authentifié, l'endpoint et les paramètres (sans les valeurs sensibles — ne pas logger les passwords, tokens, numéros de carte), le code de réponse HTTP, la durée de traitement, et l'adresse IP source.

Ces logs, ingérés dans le SIEM, permettent de créer des règles de détection des patterns d'abus : plus de 1000 requêtes vers des endpoints sensibles depuis le même compte en une heure, pattern d'énumération d'IDs séquentiels détecté via l'analyse des paramètres path (tous les IDs de 1 à 10000 consultés en 48h), ou ratio d'erreurs 403 anormalement élevé pour un client spécifique (signe d'exploitation BFLA en cours). **L'intégration des logs d'API Gateway (AWS API Gateway CloudWatch Logs, Azure APIM Diagnostics) dans le SIEM via les connecteurs natifs est une configuration de 30 minutes qui donne une visibilité complète sur le trafic API en production**, complémentaire aux WAF qui bloquent les requêtes malveillantes connues mais ne voient pas les abus de logique métier passant dans des requêtes syntaxiquement valides.

Opinion : le shift-left de sécurité API est encore une promesse non tenue

L'industrie de la sécurité prêche le "shift-left" depuis 2015 — intégrer la sécurité dès la conception des APIs plutôt qu'en bout de chaîne lors des pentests. En 2026, la réalité des ETI françaises est décevante : la grande majorité des projets API sont encore conçus sans modélisation des menaces, déployés sans scan de leur spécification OpenAPI, et seulement testés au niveau sécurité lors d'un pentest annuel externe. Le problème n'est pas technologique — les outils existent (42Crunch, schemathesis, Nuclei sont gratuits ou abordables) — c'est culturel et organisationnel. Les développeurs ne sont pas formés à la sécurité API, les architectes ne prennent pas en compte les risques OWASP lors de la conception, et les équipes de sécurité sont trop rares pour être embarquées dans chaque projet API. La seule solution durable est de rendre les développeurs responsables de la sécurité de leurs APIs via la formation et l'intégration d'outils dans leurs workflows existants — et de s'assurer que les métriques de sécurité API (couverture des tests schemathesis, score 42Crunch) sont dans les définitions de Done des équipes produit, au même titre que la couverture de tests unitaires et la conformité Sonar. Ce changement culturel est plus difficile que de déployer un WAF, mais c'est le seul qui produira des APIs intrinsèquement sécurisées plutôt qu'externalement protégées.

APIs et RGPD : droits des personnes et gestion des données

Pour les ETI françaises développant des APIs manipulant des données personnelles, le RGPD impose des exigences spécifiques qui se traduisent en exigences de sécurité API. Le droit d'accès (article 15 RGPD) nécessite un endpoint API permettant à un utilisateur d'exporter l'ensemble de ses données personnelles dans un format lisible par machine — cet endpoint doit être strictement authentifié et limité aux données du demandeur uniquement (risque BOLA si l'ID de l'utilisateur est passé en paramètre sans vérification côté serveur). Le droit à l'effacement (article 17) nécessite un endpoint API de suppression de compte qui efface effectivement les données dans tous les systèmes (bases de données, sauvegardes, logs si possible), avec une confirmation cryptographique que la suppression est complète. Le droit à la portabilité (article 20) nécessite un export des données dans des formats standardisés (JSON, CSV) via API.

Ces endpoints RGPD sont des cibles privilégiées lors des pentests API car ils accèdent à des données personnelles sensibles et sont souvent développés rapidement pour satisfaire les obligations légales sans le même niveau de revue sécurité que les endpoints métier principaux. **L'audit spécifique des endpoints RGPD (accès aux données, suppression, export) lors de chaque pentest API est recommandé par la CNIL dans ses guidelines techniques pour les DPO**, et les vulnérabilités BOLA sur ces endpoints doivent être traitées avec la plus haute priorité de remédiation, car leur exploitation peut déclencher une obligation de notification à la CNIL sous 72h au titre de l'article 33 RGPD sur les violations de données personnelles.

Automatisation des tests de sécurité API dans la CI/CD

L'intégration des tests de sécurité API dans les pipelines CI/CD transforme la sécurité d'un exercice ponctuel (pentest annuel) en un processus continu détectant les régressions de sécurité à chaque déploiement. Le pipeline de sécurité API typique se compose de quatre étapes ordonnées. Étape 1 : analyse statique de la spécification OpenAPI via 42Crunch ou Spectral (linting de schéma) — cette étape s'exécute en 30 secondes et détecte les problèmes de conception (endpoints sans authentification, réponses exposant des données sensibles, descriptions de schéma absentes) avant même que l'API soit déployée. Étape 2 : fuzzing dynamique via schemathesis contre l'environnement de staging — schemathesis génère des centaines de cas de test depuis la spécification OpenAPI et teste les réponses d'erreur, les types invalides, les valeurs aux limites, et les injections dans tous les paramètres, en moins de 5 minutes pour une API de taille moyenne.

Étape 3 : scan de vulnérabilités connus via Nuclei avec les templates API — Nuclei teste les endpoints d'administration exposés, les configurations CORS dangereuses, les introspections GraphQL, et les vulnérabilités spécifiques aux frameworks détectés. Étape 4 : tests d'autorisation automatisés via un script Postman ou pytest qui teste que les endpoints protégés retournent bien 401/403 sans authentification valide et que les accès cross-user (BOLA) sont correctement bloqués. **Ce pipeline de 15-20 minutes total s'exécute automatiquement sur chaque Pull Request et bloque le merge si une vulnérabilité critique est détectée**, créant un gate de sécurité API comparable à ce que les quality gates SonarQube font pour la qualité de code. La

mise en place de ce pipeline dans GitLab CI ou GitHub Actions nécessite environ 2 jours de travail d'un ingénieur sécurité, avec un ROI immédiat sur la réduction des vulnérabilités API en production.

Risque API tiers : la surface d'attaque invisible

L'API10:2023 OWASP soulève un problème souvent négligé dans les ETI françaises : la consommation d'APIs tierces sans validation suffisante des données reçues. Une application qui fait confiance aveuglément aux données retournées par une API partenaire ou fournisseur peut être compromise si cette API est elle-même compromise ou retourne des données malveillantes. Les scénarios concrets incluent : une API de géolocalisation tierce dont les données sont utilisées pour construire des chemins de fichiers (path traversal si la réponse contient "../"), une API de données clients partenaires dont les champs sont injectés dans des templates email (XSS si les données ne sont pas escapées), ou une API de paiement tierce dont les montants retournés sont utilisés sans vérification côté serveur (modification de prix).

La défense contre les risques API tiers suit les mêmes principes que la validation des inputs utilisateurs : ne jamais faire confiance aux données externes sans validation de schéma, d'intégrité, et de range. **Chaque API tierce consommée doit être documentée dans le registre API de l'organisation avec son niveau de confiance, ses SLA de disponibilité, et les validations appliquées aux données reçues**. En cas de breach de l'API tierce — situation de plus en plus courante avec les supply chain attacks — cette documentation permet d'identifier rapidement les applications impactées et d'évaluer l'exposition potentielle. La veille sur les incidents de sécurité des APIs tierces utilisées (via leur CVE tracker, leurs bulletins de sécurité, et les feeds CTI) est une extension naturelle de la veille CVE sur les dépendances logicielles, désormais recommandée par l'ANSSI comme composante de la gestion des risques supply chain dans le cadre NIS 2.

Microservices et mTLS : sécuriser les communications inter-services

Dans les architectures microservices, les APIs ne communiquent pas uniquement avec des clients externes — elles communiquent massivement entre elles en interne. Cette communication service-à-service est souvent non authentifiée dans les architectures naïves : un attaquant ayant compromis un microservice peut envoyer des requêtes arbitraires à tous les autres microservices du même réseau, sans que ces services ne vérifient l'identité de l'appelant. Le mTLS (Mutual TLS) est la solution standard pour l'authentification mutuelle des microservices : chaque service dispose d'un certificat client TLS signé par une CA interne, et chaque requête inter-service est authentifiée cryptographiquement via ce certificat. L'implémentation du mTLS dans Kubernetes est simplifiée par les service meshes comme Istio et Linkerd, qui injectent automatiquement les sidecars TLS dans chaque pod et gèrent la rotation des certificats de manière transparente pour les applications.

Istio avec mTLS en mode "STRICT" bloque toutes les communications inter-pods non authentifiées par TLS mutuel, créant une isolation micro-périmétrique de chaque microservice indépendante des Network Policies réseau. Cette approche Zero Trust intra-cluster signifie qu'un pod compromis ne peut pas simplement envoyer des requêtes HTTP arbitraires aux autres pods — il doit d'abord présenter un certificat valide signé par la CA Istio, un certificat qu'il ne peut obtenir que si le pod est légitime et correctement enregistré dans le plan de contrôle Istio. Pour les organisations françaises en cours de déploiement d'architecture microservices, l'intégration d'Istio ou Linkerd dès la phase initiale est bien moins coûteuse que la migration d'une architecture microservices existante vers le mTLS, qui nécessite de modifier les configurations de tous les services pour qu'ils supportent le mode STRICT et de gérer la période de transition avec un mode PERMISSIVE (mTLS optionnel) pendant la montée en charge. L'article connexe Zero Trust pour ETI françaises couvre cette dimension d'authentification intra-cluster dans le contexte plus large de l'architecture Zero Trust globale de l'organisation.

Les organisations françaises qui ont déployé une stratégie de sécurité API complète — inventaire centralisé dans un portail développeur, tests automatisés schemathesis et 42Crunch dans la CI/CD, WAF API-aware en production, et logs API intégrés dans le SIEM — rapportent une réduction de 75% des vulnérabilités API critiques découvertes lors des pentests annuels par rapport aux organisations sans programme de sécurité API structuré, un résultat qui justifie économiquement l'investissement initial dans les outils et la formation des équipes de

développement aux bonnes pratiques OWASP API Security. La sécurité des APIs est un investissement qui se rentabilise à chaque pentest évité et à chaque notification CNIL sous 72h non émise — pour les ETI françaises qui externalisent leurs développements API, exiger contractuellement la conformité OWASP API Top 10 dans les critères de recette fonctionnelle est désormais une bonne pratique reconnue par les cabinets de conseil en cybersécurité accompagnant les transformations numériques sous contrainte NIS 2.

Questions fréquentes sur la sécurité des APIs

REST ou GraphQL est-il plus sécurisé pour une nouvelle API ?

Les deux peuvent être sécurisés ou non selon leur implémentation. GraphQL a une surface d'attaque plus large par nature (introspection, query complexity, batching) mais offre aussi plus de granularité dans les autorisations (field-level permissions). REST est plus simple à sécuriser avec des règles WAF standard et bénéficie d'un écosystème d'outils de sécurité plus mature. Pour une nouvelle API sans équipe de sécurité dédiée, REST avec une spécification OpenAPI rigoureuse et des outils de validation automatique (42Crunch, schemathesis) offre un meilleur point de départ. GraphQL est préférable pour des APIs très évolutives avec des besoins de flexibilité de requête élevés, à condition de disposer d'une expertise sécurité GraphQL en interne.

Un API Gateway suffit-il à sécuriser les APIs ?

Non. L'API Gateway gère l'authentification, le rate limiting, le routage et la terminaison TLS — des contrôles nécessaires mais insuffisants. La BOLA, la Mass Assignment, les injections, et les vulnérabilités de logique métier se situent au niveau de la logique applicative de l'API et ne peuvent pas être détectées par un API Gateway standard. Un WAF API-aware (avec des règles spécifiques aux patterns d'attaque API) complète l'API Gateway pour la détection des anomalies au niveau du protocole HTTP/GraphQL, mais la sécurité réelle nécessite également des tests de sécurité dans le développement (SAST, schemathesis) et des contrôles d'autorisation corrects dans le code applicatif.

Comment gérer les credentials dans les APIs (clés tierces, connexions BD) ?

Jamais dans le code source ni dans les fichiers de configuration versionnés dans Git. Les options recommandées en ordre de préférence : 1) Variables d'environnement injectées par le système de déploiement depuis un gestionnaire de secrets (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault), 2) Secrets Kubernetes via External Secrets Operator pour les déploiements Kubernetes, 3) Fichiers .env non versionnés pour les développements locaux, en utilisant un .env.example versionné comme référence. L'outil `detect-secrets` (Yelp) ou `truffleHog` scannent automatiquement Git pour détecter les credentials hardcodés et doivent être intégrés dans les hooks pre-commit et les pipelines CI/CD.

Conclusion

Ce sujet s'inscrit dans un contexte de menaces en constante évolution. La meilleure protection combine veille active, audits réguliers et sécurité by design. Pour approfondir ou évaluer votre exposition, consultez nos experts.

Vous souhaitez en savoir plus ou évaluer votre exposition ?

Contacter nos experts ou contactez-nous directement.