LES ESSENTIELS

Sécurisation des API

Mars 2024

CLUSIF

Table des matières

Remerciements

Le Clusif tient à mettre ici à l'honneur les personnes qui ont rendu possible la réalisation de ce document, tout particulièrement :

Le responsable du groupe de travail :

Antony	**GONÇALVES**	NUTRITION & SANTE

Les membres du groupe de travail :

Caterina	**PREDINE**	ONEY BANK
Henri	**CODRON**	SCHINDLER
Bertrand	**CARLIER**	WAVESTONE
Xavier	**AGHINA**	ORANGE
Cédric	**CAILLEAUX**	AXIANS - VINCI ENERGIES
Arnaud	**LEFEBVRE**	SAINT GOBAIN
Kevin	**MARICHY**	PLUS QUE PRO
Antoine	**BAJOLET**	HENNER
Benjamin	**FINO**	VICAT
Ghizlane	**OUTAGHYAME**	ALLIANZ IARD
Thomas	**CHEVALLIER**	NS GROUP
David	**BRAZ**	SAFRAN
Alexandru	**LATA**	CLOUD TEMPLE
Benjamin	**CAZIER**	ELOQUANT

Le Clusif remercie également les adhérents ayant participé à la relecture.

1. Objet

Avant 2000, il n'existait aucune norme sur la manière de concevoir une API ni de l'utiliser. Son intégration requérait l'utilisation de protocoles, comme SOAP, notoirement complexes à construire, à manier et difficiles à débuguer.

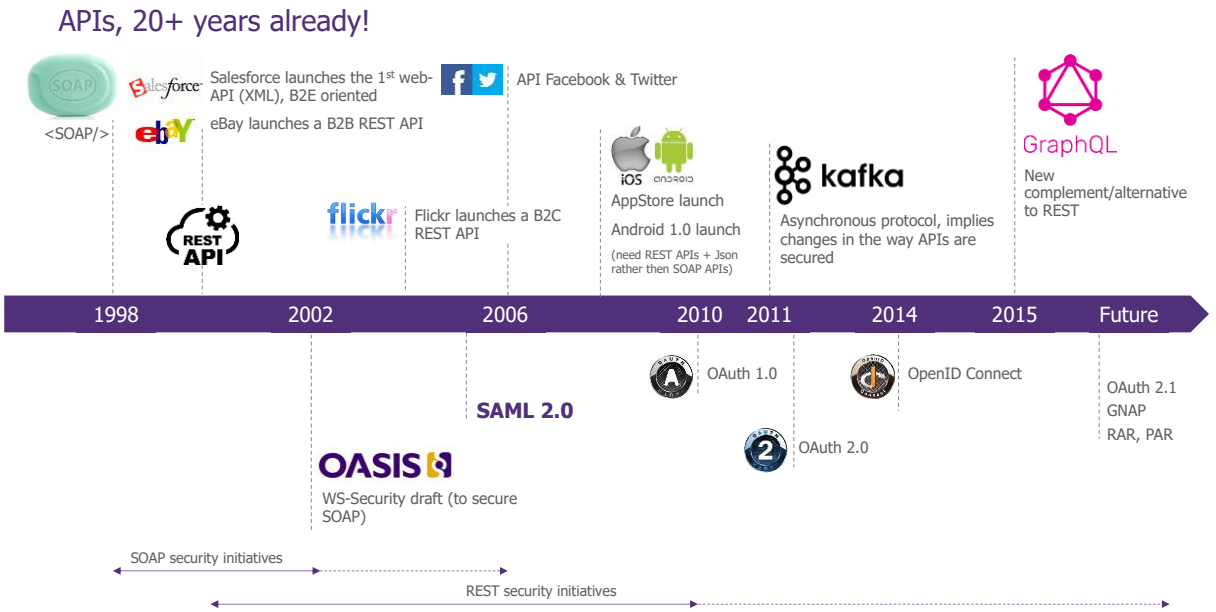
Cela change en 2000 quand est reconnu le véritable potentiel des API web : un groupe d'experts, dirigé par Roy Fielding, va inventer REST et modifier le paysage API à jamais.

L'objectif avoué est simplement de créer une norme permettant la communication, l'échange de données entre deux serveurs, n'importe où dans le monde. Ils conçoivent donc un ensemble de principes, de propriétés et de contraintes baptisé REST, une architecture orientée ressource : uniformité de l'interface, architecture client/serveur, sans état ni maintien de session, mise en cache de la représentation de la ressource, utilisation du protocole HTTP et de ses méthodes.

Les premiers à s'intéresser au phénomène sont les géants de l'e-commerce, eBay puis Amazon.

L'accès à l'API REST eBay, facile à utiliser, doté d'une solide documentation, est offert à une sélection de partenaires. Elle va démontrer combien lucrative peut se révéler une API désormais accessible. Ainsi, sa place de marché n'est plus limitée aux seuls visiteurs de son site internet, mais s'étend à tout site accédant à son API.

Le bénéfice est évident : visibilité accrue de son offre de produits, et donc démultiplication d'occasions de vente ! La simplicité du système séduit immédiatement d'autres plateformes en ligne, qui se mettent à réfléchir à la valeur de leur code, et non plus uniquement à leurs produits de consommation.



Ce document a pour but de décrire les bonnes pratiques concernant les API – Application Programming Interface. Dans un contexte d'informatique décentralisée, les API permettent au système d'information de l'entreprise de se connecter aux applications métiers, aux partenaires, aux terminaux, mais aussi aux clients.

La sécurisation des API permet de garantir les quatre principes fondamentaux que sont la **confidentialité** : les données ne vont qu'aux personnes autorisées ; l'**intégrité** : les données ne sont ni transformées ni supprimées ; la **disponibilité** : les données sont disponibles au moment souhaité ; et enfin la **traçabilité** pour garantir une chaîne de responsabilité.

2. Définitions

API : Une **API** (pour **Application Programming Interface**) est un programme permettant à deux applications distinctes de communiquer entre elles et d'échanger des données. Cela évite notamment de recréer et redévelopper entièrement une application pour y ajouter ses informations. Par exemple, elle est là pour faire le lien entre des données déjà existantes et un programme indépendant.

Les protocoles : Du fait de la grande diversité d'applications client, les API doivent s'appuyer sur un protocole de communication, le **SOAP** (Simple Object Access Protocol) ou le **REST** (Representational State Transfert) afin d'être compatibles avec les diverses plateformes mobiles, qu'il s'agisse d'une application Windows, Apple ou Android. L'API Rest (ou Restful) est à présent la plus utilisée car elle offre plus de flexibilité.

EAI : L'EAI (Entreprise Application Integration) est une architecture permettant à des applications hétérogènes de gérer leurs échanges. Par extension, l'EAI désigne un système informatique permettant de réaliser cette architecture en implémentant des flux interapplicatifs du système d'information. Enfin, l'EAI définit des protocoles d'interaction des systèmes, afin de normaliser les architectures logicielles et le formatage des données.

API SOAP : Ces API utilisent le protocole simple d'accès aux objets. Le client et le serveur échangent des messages via XML. Il s'agit d'une API moins flexible et qui est de moins en moins utilisée.

API WebSocket : Il s'agit d'un autre développement d'API web moderne qui utilise des objets JSON pour transmettre des données. Une API WebSocket prend en charge la communication bidirectionnelle entre les applications client et le serveur. Le serveur peut envoyer des messages de rappel aux clients connectés, ce qui le rend plus efficace que l'API REST.

API Rest : Il s'agit des API les plus demandées et les plus flexibles que l'on trouve de nos jours. Le client adresse ses demandes au serveur sous forme de données, au travers de fonctions GET, PUT, DELETE, etc.

JSON : JavaScript Object Notation est un format de fichier permettant de stocker des données de manière organisée et lisible, très souvent utilisé dans les API.

Système d'information : Ensemble des moyens humains et matériels ayant pour finalité d'élaborer, traiter, stocker, acheminer, présenter ou détruire l'information.

XSS : Le cross-site scripting (abrégé XSS) est un type de faille de sécurité des sites web qui permet d'injecter du contenu dans une page, provoquant ainsi des actions sur les navigateurs web visitant la page. Les possibilités des XSS sont très larges puisque l'attaquant peut utiliser tous les langages pris en charge par le navigateur (JavaScript, Java...), et de nouvelles possibilités sont régulièrement découvertes, notamment avec l'arrivée de nouvelles technologies. Il est par exemple possible de rediriger vers un autre site pour de l'hameçonnage ou encore de voler la session en récupérant les cookies.

CSRF : Cross-Site Request Forgery. L'objet de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. L'utilisateur devient donc complice d'une attaque sans même s'en rendre compte. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés.

Injection (SQL) : L'injection SQL est une faiblesse de cybersécurité qui permet à un hacker d'interférer avec les requêtes qu'une application effectue sur sa base de données en utilisant un bout de code SQL (Structured Query Language).

3. Menaces sur les APIS

La multiplication de l'utilisation des API, ainsi que des services cloud, induit une augmentation de la surface d'attaque pour les organisations.

Les attaques ciblant les API ne cessent de croître et deviennent un point de vigilance majeur pour l'entreprise.

OWASP – Open Web Application Security Project – API

Source : https://owasp.org/www-project-api-security/

L'OWASP a réalisé un classement des vulnérabilités sous forme de Top 10 par ordre d'importance :

API1 : 2019 Broken Object Level Authorization (ex. : il s'agit de la manipulation d'identifiants d'objets dans une requête pour obtenir un accès non autorisé à des données sensibles).

API2 : 2019 Broken User Authentication (ex. : les attaquants peuvent être en mesure de se faire passer pour des utilisateurs d'API, ce qui leur permet d'accéder à des données confidentielles).

API3 : 2019 Excessive Data Exposure (ex. : cela peut permettre à des personnes non autorisées de visualiser les données).

API4 : 2019 Lack of Resources & Rate Limiting (ex. : par défaut, de nombreuses API ne limitent pas le nombre ou la taille des demandes qu'elles peuvent recevoir à un moment donné. Cela les rend vulnérables aux attaques par déni de service (DoS)).

API5 : 2019 Broken Function Level Authorization (ex. : les utilisateurs de l'API peuvent être autorisés à en faire trop, ce qui entraîne une exposition des données).

API6 : 2019 Mass Assignment (ex. : un attaquant pourrait utiliser cette vulnérabilité pour, par exemple, s'attribuer des droits administrateur tout en mettant à jour une autre propriété inoffensive de son profil d'utilisateur).

API7 : 2019 Security Misconfiguration (ex. : cela couvre une variété d'erreurs dans la configuration de l'API, y compris des en-têtes HTTP mal configurés, des méthodes HTTP inutiles et ce que l'OWASP appelle « des messages d'erreur verbaux contenant des informations sensibles »).

API8 : 2019 Injection (ex. : dans une attaque par injection, l'attaquant envoie des commandes spécialisées à l'API pour l'inciter à révéler des données ou à exécuter une autre action inattendue).

API9 : 2019 Improper Assets Management (ex. : cela se produit lorsqu'il n'y a pas de suivi des API de production actuelles ni de celles qui ont été dépréciées, ce qui conduit à des API fantômes. Les API sont vulnérables à ce risque car elles ont tendance à en mettre à disposition un grand nombre).

API10 : 2019 Insufficient Logging & Monitoring (ex. : une journalisation détaillée des événements et une surveillance étroite peuvent permettre aux développeurs d'API de détecter et d'arrêter les brèches bien plus tôt).

4. RECOMMANDATIONS DE MESURES DE SÉCURITÉ

Rendre des API disponibles sur internet ne signifie pas qu'elles sont en accès libre, de manière non sécurisée. Il existe des protocoles Web standardisés pour gérer l'authentification et l'habilitation des applications appelantes et des utilisateurs connectés à ces applications. La pierre angulaire étant le protocole OAuth2, qui est un framework d'autorisation et qui peut être étendu pour prendre en charge l'authentification via le protocole OpenID Connect. Les recommandations abordées dans ce guide sont également valables pour les API utilisant des protocoles propriétaires.

L'objectif étant de permettre aux développeurs de consommer facilement l'API, ces protocoles n'incluent pas de mécanismes de signature ou de chiffrement applicatif des messages (ils reposent uniquement sur la couche de transport TLS qui garantit l'intégrité et la confidentialité des échanges) et permettent une consommation de l'API par tout type de service : non seulement des serveurs, mais aussi des navigateurs, des applications natives, etc.

Le mécanisme de sécurité à implémenter dépend du type de ressources. Il existe en effet deux types de ressources :

- Les ressources publiques dont les données ne dépendent pas d'un utilisateur final. L'application appelante est habilitée via une API_KEY.
- Les ressources privées dont les données dépendent d'un utilisateur final.

L'application appelante est habilitée via un access_token OAuth2 qui permet d'identifier l'utilisateur connecté.

En complément, le niveau de risque dépendra du degré d'exposition de l'API elle-même. Par exemple, une API privée utilisée uniquement dans un réseau privé authentifié sera par nature moins exposée qu'une API publique ouverte sur Internet. Cette évaluation doit être intégrée dans une approche d'analyse de risque plus globale et en accord avec la politique de sécurité des systèmes d'information de l'entreprise.

4.1 Authentification

R1 : Ne pas utiliser une authentification basique http (Basic Auth) ou keyless, mais plutôt un standard d'authentification tel que OAuth2.

R2 : Privilégier l'utilisation d'OAuth2 pour la couche authentification.

R3 : Vérifier l'autorisation à chaque requête de la BDD. En complément il est indispensable de mettre en place des seuils de bannissement (jail) avec des seuils de tentatives maximales.

R4 : Toutes les données sensibles doivent être chiffrées. On entend par « données sensibles », des données personnelles et les données confidentielles.

R5 : Protéger les secrets durant tout leur cycle de vie. Pas de secrets stockés par exemple dans un Git.

4.2 JSON Web Token

R6 : Vérifier l'intégrité du JSON Web Token (JWT) via une signature si cela est possible.

R7 : Utiliser des clés aléatoires complexes et non prédictibles pour rendre les attaques par brute force complexes.

R8 : Limiter la durée de vie des jetons au strict nécessaire (TTL, RTTL).

4.3 OAuth 2.0

R9 : Toujours valider la redirection d'URL côté serveur afin d'accéder uniquement aux URL autorisées.

R10 : Utiliser le paramètre d'état (state) avec un hash aléatoire pour prévenir les CSRF sur le processus d'authentification OAuth2.0.

4.4 Accès

De façon générale, l'accès doit être limité seulement aux personnes ayant un « besoin d'en connaître ».

R11 : Limiter le nombre de requêtes pour limiter les dénis de services ou les attaques par brute force.

R12 : Utiliser obligatoirement le protocole HTTPS côté serveur ainsi qu'une version TLS non obsolète (ex. : TLS 1.2 ou 1.3).

4.5 Validation des données / Sorties

R13 : Valider le content-type dans l'en-tête HTTP des requêtes (négociation de contenu) pour n'autoriser que les formats supportés (ex : application/xml, application/json, etc.) et renvoyer une réponse 406 Not Acceptable si ça ne correspond pas.

R14 : Valider les entrées utilisateur et leur format pour éviter les vulnérabilités classiques (e.g. XSS, SQL-Injection, Remote Code Execution, etc.) et rejeter les contenus non prévus ou incorrects.

R15 : Ne pas utiliser des données sensibles (identifiants, mots de passe, token de sécurité, clés API) dans l'URL, mais utiliser les en-têtes d'autorisations standards.

R16 : Ne pas utiliser d'identifiant auto-incrémenté, mais plutôt un UUID aléatoire non prédictible.

R17 : Supprimer les en-têtes d'empreinte sur les serveurs : Powered by, Server X, ASPversion, etc.

4.6 Journalisation

R 18 : Produire une trace d'audit avant et après chaque évènement de sécurité. Les erreurs de validation, les échecs d'authentification doivent également être journalisés et conservés sur une période minimale de 6 mois.

R19 : En cas d'erreur, des messages d'erreurs génériques ne divulguant pas d'informations techniques doivent être prévus.

R20 : Dans les fichiers de journalisation, les secrets ne doivent pas apparaître.

4.7 Architecture

Le choix d'une architecture adaptée est également structurant pour la bonne sécurisation des API. Il n'existe pas une architecture unique à même de répondre aux besoins de chaque contexte. Mais de la même manière qu'il existe des principes de sécurisation déclinables à chaque contexte particulier, un certain nombre de questions peuvent être posées pour *in fine* concevoir une architecture adaptée et sécurisée.

Type de consommateur de l'API : application autonome ou avec un utilisateur

Une des premières questions à traiter est de comprendre quel type de client va consommer l'API. S'agit-il d'une application autonome agissant en son nom ou bien d'une application avec laquelle un utilisateur interagit et qui devra consommer votre API au nom de l'utilisateur ? Parfois, un utilisateur interagit avec une application, mais cette dernière appelle les API sans transmettre l'identité utilisateur (car ce n'est pas un besoin fonctionnel, ou bien l'API ne le prévoit tout simplement pas).

Répondre à cette question va permettre d'orienter la réponse vers d'autres questions et pourra déjà orienter certains choix techniques (ex. : dans le cas d'une API REST protégée par Oauth2 et consommée par des applications sans utilisateur, la cinématique *client credentials* sera à privilégier).

4.7.1 Audience des APIs et exposition

Il est également important de se poser la question de la nature fonctionnelle des clients de l'API.

- S'agit-il d'applications connues et maîtrisées et s'exécutant sur des terminaux connus et maîtrisés ? (ex : un client lourd à destination d'employés sur leur poste de travail fourni par l'entreprise).
- S'agit-il d'applications connues et maîtrisées, mais s'exécutant sur les terminaux appartenant à des utilisateurs tiers ou des partenaires métier ? (ex. : une application mobile à destination des clients).
- S'agit-il d'applications développées par des tiers et consommant des API qui doivent être exposées vers l'extérieur ?

En fonction de la réponse à ces questions, on décidera d'exposer plus ou moins une API. Il ne sert à rien d'exposer sur Internet une API qui a vocation à n'être consommée qu'en interne, la minimisation de la surface d'attaque doit rester un principe clé de l'architecture choisie. D'autres API auront vocation à être exposées, mais sécurisées via de l'authentification et de l'autorisation (voir plus bas), enfin d'autres API auront vocation à exposer des données publiques et dans ce cas un suivi de la consommation peut suffire en fonction des besoins de disponibilité du service.

4.7.2 Authentification et Autorisations

Viennent ensuite les questions de l'authentification et de l'autorisation. Et ces questions peuvent concerner d'une part l'application appelante, et d'autre part l'utilisateur de cette application.

Commençons par l'authentification.

Si un utilisateur est présent, et si le cas d'usage le requiert, il est généralement possible de l'authentifier de façon fiable à l'aide d'un ou deux facteurs, toujours selon le niveau de sensibilité des opérations ou données manipulées.

L'application en revanche peut se révéler plus difficile à authentifier de manière fiable. En effet, une application s'exécutant sur un terminal non maîtrisé (ex. : application mobile, application Javascript s'exécutant dans le navigateur de l'utilisateur ou client lourd sur un PC) est incapable de protéger un secret permettant de s'authentifier ; il sera toujours possible à un attaquant disposant de temps suffisant de rétro-ingéniérer l'application et d'extraire ce secret. Une application s'exécutant sur un serveur hors de portée de l'utilisateur ou une application s'exécutant sur un terminal muni d'un équipement de sécurité matériel (ex. : TPM, puce, SIM) pour stocker son secret et le protéger correctement. L'authentification d'une telle application est alors considérée comme fiable.

Dernier point concernant l'authentification d'une application : elle peut prendre plusieurs formes.

- Un identifiant simple (ex. : API key) qui n'offre aucune garantie de sécurité.
- Un secret partagé (ex. : client_secret) à l'image d'un mot de passe applicatif qui sera considéré comme un premier niveau de sécurité.
- L'utilisation d'une bi-clé dont la clé publique est partagée entre le client et le serveur, mais dont la clé privée reste sous le contrôle exclusif de l'application et qui peut être utilisée de différentes manières : mutual TLS (ou mTLS), Proof-of-Possession (ou PoP) qui va faire transiter dans le flux applicatif la preuve de la détention de la clé privée.

Une fois l'authentification réalisée (celle de l'utilisateur et/ou celle de l'application), il est possible d'envisager de gérer des autorisations concernant indépendamment l'application ou l'utilisateur : un client de courrier électronique est légitime d'accéder à une API de gestion de courriels, mais pas une API de stockage de fichiers, un utilisateur donné ne peut accéder qu'à ses propres courriels, ses propres fichiers et les fichiers qui lui sont partagés.

Les autorisations finales fournies au moment de l'accès dans un tel exemple sont en quelque sorte l'intersection des droits applicatifs avec les droits de l'utilisateur présent et authentifié.

Enfin, on peut dans certains cas considérer que l'autorisation n'est valable que pour une transaction précise. Par exemple, la capacité de réaliser un virement bancaire d'un compte A vers un compte B pour un montant X peut être accordée de façon unitaire et non rejouable ; nous pouvons dans ce cas utiliser un jeton à usage unique.

4.7.3 Format des jetons

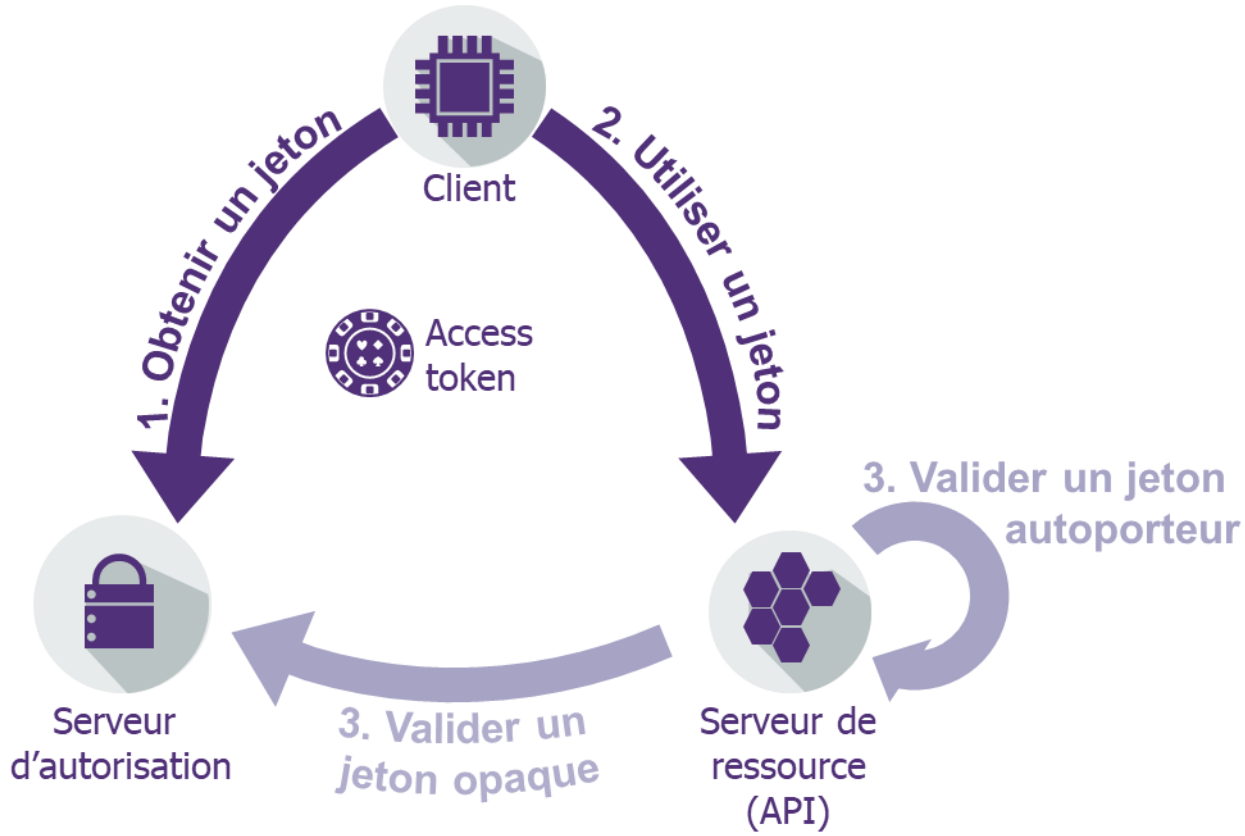
La plupart du temps, l'authentification (utilisateur et/ou application) est réalisée une fois, et un jeton valable pour une durée arbitraire est fourni à l'application pour appeler l'API. Si le format de ce jeton est généralement ignoré par principe par le client applicatif, il est important d'étudier, côté API, l'opportunité d'utiliser un format dit « opaque » ou « autoporteur ».

Un **jeton opaque** est un jeton généré et validé par un composant de l'architecture. Le client obtient le jeton de la part de ce serveur, l'utilise dans l'appel à l'API, et l'API, de son côté, doit contacter le serveur ayant généré le jeton pour le valider et en obtenir le contexte (identifiants de l'application et/ou de l'utilisateur, date et heure de génération du jeton, durée de validité et toute autre information pertinente). La plupart du temps, le jeton est une simple chaîne de caractères aléatoires correspondant à ce contexte stocké en mémoire côté serveur d'authentification. Plusieurs avantages existent dans cette approche : limiter la fuite d'information lorsque le jeton circule sur un réseau non maîtrisé (ex. : Internet), limiter la taille du jeton et donc des requêtes, permettre la révocation d'un jeton. Ces avantages peuvent en revanche venir au prix d'une latence de validation de jeton plus importante, une gestion de cache partagé pour tous les serveurs d'une grappe susceptibles d'être contactés pour valider un jeton.

Un **jeton autoporteur** constitue une autre approche dans laquelle le jeton contient toutes les informations nécessaires à l'API pour être validé de manière autonome (émetteur, utilisateur, client, expiration, etc.).

Le jeton est généralement signé par le serveur émetteur pour en assurer l'intégrité, il peut être également chiffré si l'on veut en maintenir la confidentialité sur un réseau non maîtrisé au prix d'un déchiffrement par l'API, mais surtout une gestion de clés de chiffrement avec chacune des API considérées. Il est également complexe de réaliser une révocation efficace de ce type de jeton.

Le schéma ci-dessous résume le mécanisme global d'utilisation d'un jeton pour sécuriser une API.



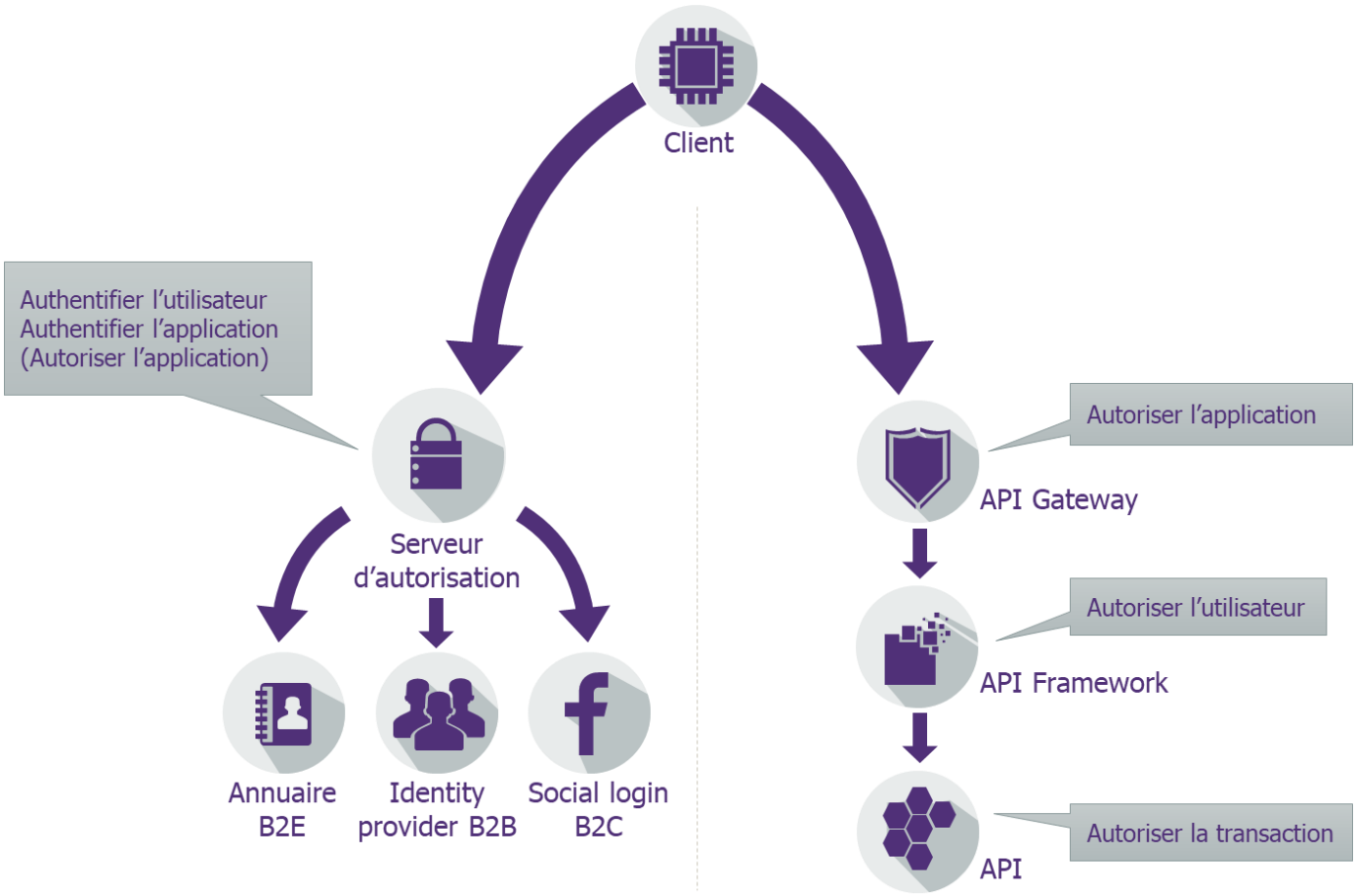
4.7.4 Architecture OAuth2

Le schéma présenté ci-dessus introduit le vocabulaire de serveur d'autorisation (*authorization server)* qui est le terme utilisé dans le protocole OAuth2 pour désigner le serveur chargé d'authentifier utilisateur et application, optionnellement de réaliser un premier niveau d'autorisation, d'éventuellement recueillir un consentement de l'utilisateur à partager ses données avec une application et *in fine* de générer et distribuer un jeton d'accès (*access token*).

Ce serveur d'autorisation peut réaliser lui-même l'authentification en s'appuyant sur un annuaire d'authentification ou la déléguer à un serveur tiers (typiquement en utilisant un protocole tel que SAML ou OpenID Connect).

Une fois le jeton transmis à l'application cliente puis transmis en direction de l'API (*resource server* dans le vocabulaire Oauth2), il peut être pertinent de positionner une API Gateway entre le client et l'API pour mieux gérer l'exposition des API sous-jacentes, valider un jeton opaque, et éventuellement le transformer au passage, réaliser un certain niveau de contrôle sur les autorisations, etc.

Le schéma ci-dessous représente une option d'architecture haut niveau qu'il convient de décliner et d'adapter à chaque contexte métier.



Un certain nombre de questions sont à étudier pour affiner cette architecture haut niveau :

- Quels composants d'infrastructure déployer en amont d'un serveur d'autorisation et d'une API Gateway ?
    - La recommandation face à un serveur d'autorisation OAuth2 est de n'avoir que des composants « réseau » ne modifiant pas la requête au niveau de la couche applicative (HTTP).
    - Une API Gateway peut quant à elle être positionnée derrière un composant type Web Application Firewall (WAF) pour se prémunir d'un certain nombre d'attaques si la Gateway n'est pas équipée elle-même de telles protections.
- Un framework de développement d'API est-il envisageable pour guider les développeurs d'API auprès de l'organisation pour en maîtriser le niveau de sécurité ?
- Comment correctement sécuriser le chemin d'accès à l'API sans pouvoir contourner l'API Gateway, par exemple ?
- Une seule API Gateway est-elle suffisante ou bien une dédiée aux accès externes doit-elle s'articuler avec une permettant les accès internes ? Il peut même être envisagé de déployer plusieurs API Gateway différentes pour servir différents métiers ou différents usages (ex. : interne et externe).

4.7.5 Cas des appels internes entre API

Une fois l'architecture définie pour des appels « simples » à des API, peuvent survenir des questions autour des appels chaînés entre API :

- Besoin de transmettre tout ou partie du contexte d'appel initial (propagation de l'identité de l'utilisateur) et de la chaîne elle-même (liste ordonnée des API intermédiaires).

- Besoin de repasser ou non par une Gateway et étude des impacts en termes de performances par rapport au gain de sécurité.
- Échange et/ou réévaluation du jeton d'accès entre chaque maillon de la chaîne d'appel ou entre certains appels uniquement (changement de « domaine métier » ou accès à une API d'un niveau de sensibilité supérieure).
- Opportunité de s'appuyer sur un framework d'exécution de microservices et des mécanismes intégrés de contrôle d'accès interservices.

Enfin, pour conclure, l'étude de l'ensemble de ces points d'architecture s'appuie le plus souvent sur une analyse de risque réalisée en amont et permettant de quantifier le besoin de sécurité en phase avec le besoin métier.

Campus Cyber
Tour Eria – 5 rue Bellini – 92821 Puteaux cedex
France
✆ +33 1 53 25 08 80
clusif@clusif.fr

clusif.fr