RFC 6819 : OAuth 2.0 Threat Model and Security Considerations

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 8 janvier 2013

Date de publication du RFC : Janvier 2013

https://www.bortzmeyer.org/6819.html

Le protocole d'autorisation sur le Web OAuth version 2 est riche, voire complexe. Son cadre général est décrit dans le RFC 6749 [1]. Et ce nouveau RFC documente les questions de sécurité liées à OAuth v2. Quelles sont les menaces ? Quels sont les risques ? Et comment OAuth gère-t-il ces risques ?

Le document reste assez abstrait car les détails, notamment les probabilités d'une attaque réussie, dépendent de la façon dont OAuth est utilisé. OAuth v2 n'est pas réellement un protocole mais plutôt un cadre général dans lequel peuvent être décrits plusieurs protocoles, qui n'auront pas la même utilisation, ni les mêmes vulnérabilités. En prime, ce mécanisme complexe n'est pas au cœur de mon domaine de compétence personnel, et je vais devoir me contenter de survoler le RFC. Il n'était à l'origine que la section "*Security Considerations*" du RFC 6749 mais il a grossi suffisamment pour devenir un document autonome, très détaillé.

Donc, d'abord, en section 2, le cadre général. Par exemple, l'analyse des risques de ce RFC suppose que l'attaquant a un accès complet au réseau et peut écouter à sa guise, voire changer les paquets. On considère également possible que deux des parties impliquées peuvent s'allier pour tromper la troisième. OAuth a trois - ou quatre, dans la v2 - acteurs, le client, le serveur et le propriétaire. Dans la v2, le serveur a été séparé en deux, le serveur d'autorisation et le serveur de ressources. La section 2.3 fait la liste de tout ce que connaissent les différentes parties, Le serveur d'autorisation connait ce qui sert à authentifier (par exemple les condensats des mots de passe des propriétaires). Le serveur de ressources connait les données qu'on veut protéger. Le client (l'application qui cherche à accéder aux données), par contraste, connait nettement moins de choses : le but principal d'OAuth était justement de permettre au propriétaire de donner un accès limité à des clients tiers, à qui il ne fait pas forcément une confiance aveugle.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6749.txt

OAuth repose largement sur des jetons ("*tokens*"), un petit bout de données qui sert à prouver qu'on est légitime. Avec certains de ces jetons (par exemple les jetons au porteur, "*bearer tokens*"), la seule possession du jeton suffit. Il est donc important de ne pas permettre à un tiers de les copier (donc, il faut communiquer en HTTPS, comme avec un "*cookie*", cf. section 4.3.1, et aussi section 4.6.7 pour un moyen plus rigolo de faire fuiter les jetons).

Les jetons peuvent avoir une durée de vie limitée (RFC 6749, section 4.2.2), et cela permet à un utilisateur OAuth de retirer sa confiance à un client. Les jetons qu'a obtenu ce dernier avant la révocation de la confiance ne lui serviront pas éternellement.

La section 4 est le cœur du RFC. Elle expose toutes les menaces auxquelles a pu penser le groupe de travail, avec les solutions existant dans le cadre OAuth. Je ne vais pas la reprendre intégralement (pas moins de cinquante menaces ont été identifiées), juste en expliquer quelques-unes prises au hasard.

Par exemple (section 4.1.1), OAuth permet de stocker certains secrets (comme des jetons de rafraîchissement) dans le client, pour éviter de devoir passer par une autorisation explicite de l'utilisateur. La possession de ces secrets permet évidemment de se faire passer pour un client légitime. Certains programmeurs peuvent être tentés de planquer ces secrets dans un client distribué en binaire, en croyant naïvement que cela empêchera les "*hackers*" de les trouver. Le RFC rappelle donc à juste titre que la rétro-ingéniérie retrouvera rapidement ces secrets, même si le code source du client n'est pas distribué. La protection est donc simple : pas de secret dans un client public (section 5.3.1).

Même si les clients n'ont pas de secret codé en dur dans leur code, une fois qu'ils ont été utilisés, des secrets se retrouveront dans leur mémoire. En volant un engin qui fait tourner le client (un "*smartphone*", par exemple), le méchant se retrouvera en possession des secrets. Les solutions sont dans les systèmes de verrouillage mais aussi dans les possibilités de révocation d'OAuth (encore faut-il que l'utilisateur y pense, lorsque son "*smartphone*" a été volé).

Autre menace, le hameçonnage. Comme OAuth dépend beaucoup des redirections HTTP (par exemple du site Web qu'on veut visiter vers le site Web d'autorisation), un utilisateur peut se faire tromper si la redirection est faite vers un site Web qui **ressemble** au site d'autorisation habituel. Autre façon de hameçonner, le client malveillant peut aussi, au lieu d'utiliser le navigateur Web normal du système, présenter à l'utilisateur humain sa propre interface, en la remplissant d'indicateurs rassurants, petit cadenas et tout. Le client, dans OAuth, ne devrait jamais avoir accès aux informations d'authentification de l'utilisateur (comme le mot de passe) mais ces techniques de hameçonnage le permettent. La seule solution est la vigilance de l'utilisateur, qui ne devrait se servir que de clients honnêtes. (Oui, je sais que c'est plus facile à écrire dans un RFC qu'à faire respecter en pratique. Le RFC suggère des contrôles de sécurité des applications avant de les mettre dans les dépôts comme l'App Store mais on sait que de tels contrôles ne sont pas faits `<https://www.bortzmeyer.org/malware-iphone.html>`.) Si le client est un navigateur Web, on sait que ces logiciels très complexes ont déjà eu de nombreuses failles de sécurité, permettant à un méchant de les détourner de leur rôle.

Outre le hameçonnage par le client (l'application qui tourne sur la machine de l'utilisateur), il y a un risque de hameçonnage par le serveur d'autorisation. Si celui auquel on accède n'est pas celui qu'on croit (par des trucs comme l'empoisonnement DNS ou ARP), et qu'on lui confie son mot de passe, on est fichu. L'utilisation de HTTPS est donc nécessaire.

OAuth soulève aussi des problèmes d'ergonomie : le serveur d'autorisation informe (voire permet de choisir) sur les permissions exactes qui seront données au client. Cette liste n'est pas toujours comprise par l'utilisateur, qui risque de donner au client trop de droits (section 4.2.2). Voici un exemple d'une application qui demande beaucoup de droits (car c'est un client Twitter généraliste) :

D'une manière générale, comme l'analyse la section 5.5, l'utilisateur manque d'expertise pour répondre intelligemment à des questions comme « il y a un problème de sécurité [ici, texte incompréhensible par l'utilisateur], voulez-vous continuer ? ». De même, la plupart des bonnes pratiques de sécurité (par exemple, l'utilisation obligatoire de seulement certains logiciels, soigneusement validés) vont être une gêne pour l'utilisateur. OAuth n'offre pas de solution ici, mais il est probable qu'il n'en existe pas : c'est le conflit classique entre sécurité et utilisabilité.

Certaines attaques ne sont pas réellement spécifiques à OAuth et visent le processus d'authentification de l'utilisateur, par exemple en essayant, auprès d'un serveur d'autorisation, des dictionnaires entiers de mots de passe pour essayer d'en trouver un qui marche. Les contre-mesures classiques s'appliquent ici, comme les si pénibles CAPTCHA (section 5.1.4.2.5) ou comme le verrouillage automatique des comptes après N essais infructueux (section 4.4.3.6).

Naturellement, le même processus d'authentification de l'utilisateur est vulnérable aux attaques Web classiques comme le "*clickjacking*". Un navigateur Web moderne est rempli de tellement de fonctions différentes, visant en général à faire joli, qu'il est difficile de vérifier la totalité des risques de sécurité. Si un site Web charge la page du serveur d'autorisation dans un "*iframe*" transparent, au-dessus de ses propres boutons, il peut faire en sorte que l'utilisateur, croyant cliquer sur les boutons visibles, donne en fait une autorisation OAuth. Il existe plusieurs contre-mesures (section 4.4.1.9) mais des tas d'autres vulnérabilités attendent certainement, dissimulées dans les millions de lignes de code des navigateurs.

L'enfer étant pavé de bonnes intentions, ce sont aussi les solutions de sécurité, conçues pour résoudre une vulnérabilité, qui peuvent être utilisées pour monter une attaque. Ainsi, un méchant peut utiliser OAuth pour faire en sorte que des clients se connectent au serveur d'autorisation en masse. Cela oblige l'attaquant à faire une connexion HTTP mais, en échange, il obtient des connexions HTTPS, bien plus coûteuses (cf. « "*SSL handshake latency and HTTPS optimizations*" <http://www.semicomplete.com/blog/geekery/ssl-latency.html> » de J. Sissel), vers le serveur d'autorisation. De même, si vous vous y connaissez en sécurité, vous avez peut-être noté que le verrouillage automatique des comptes après N essais ratés, proposé plus haut, permet une belle attaque par déni de service en autorisant n'importe qui à verrouiller un compte (cf. section 5.1.4.2.3).

Comme indiqué au début, je suis loin d'avoir listé toutes les menaces, notamment celles reposant sur de subtiles propriétés d'OAuth. N'écrivez pas un logiciel OAuth en suivant uniquement cet article, lisez bien tout le RFC !

Et les solutions à ces problèmes ? La section 5 regroupe les bons principes à suivre lorsqu'on développe ou déploie du OAuth. Chiffrer le trafic, avec TLS ou IPsec (OAuth vérifie l'authenticité et l'intégrité des requêtes mais, utilisé seul, ne fait rien pour la confidentialité), vérifier l'identité du serveur d'autorisation, etc.

Il y a aussi des mesures moins techniques, liées à l'interaction avec un humain, l'utilisateur. Il doit toujours être maintenu « dans la boucle », en lui demandant avant et en l'informant après de ce qui a été fait en son nom (section 5.1.3).

La section 5.1.4 se penche sur le problème de la conservation des « lettres de créance » ("*credentials*"), ces différentes informations que le serveur d'autorisation doit stocker en mémoire pour authentifier les utilisateurs. Elle rappelle l'importance de suivre les meilleures pratiques de ce domaine (comme documenté par OWASP <https://www.owasp.org/>). Cela va des protections contre les injections SQL <https://www.bortzmeyer.org/sql-injection.html> (requêtes paramétrées, accès à la base de données sous un utilisateur non privilégié, etc) à la nécessité de ne **pas** stocker les mots de passe en clair. En 2012, garder des mots de passe en clair dans une base de données (qui se

fera forcément copier un jour ou l'autre par un attaquant quelconque) est une faute grave. La méthode sûre la plus souvent utilisée est de ne stocker qu'un condensat du mot de passe mais cela ne suffit pas toujours, face aux attaques par dictionnaire, facilitées notamment par les tables arc-en-ciel. Ainsi, la copie de la base de mots de passe condensés de LinkedIn <http://mashable.com/2012/06/06/6-5-million-linkedin-passwords/> avait eu des conséquences sérieuses car les condensats n'étaient pas salés (section 5.1.4.1.3).

Si on génère des secrets qui n'ont pas besoin d'être manipulés par des êtres humains (comme les jetons), il faut bien penser à les faire longs et réellement aléatoires (RFC 4086).

Voilà, il existe encore plein de solutions décrites dans cette section, notamment sur les problèmes spécifiques à OAuth (sur lesquels je suis passé un peu rapidement, car je ne connais pas assez ce protocole). Si vous programmez un logiciel OAuth, il va évidemment falloir les lire.