RFC 9943 : An Architecture for Trustworthy and Transparent Digital Supply Chains

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 1 juillet 2026

Date de publication du RFC : Juin 2026

https://www.bortzmeyer.org/9943.html

Des attaques sur la chaine d'approvisionnement du logiciel, il y en a tout le temps. L'attaquant arrive à glisser du code malveillant dans un logiciel, en amont de son installation par la victime et, quand celle-ci fait tourner le logiciel, le code de l'attaquant est exécuté et paf. Il n'y a pas de solution miracle à ce problème mais on peut au moins essayer de fournir une traçabilité forte des logiciels, qui permet d'être sûr de ce qu'on fait tourner sur ses machines, et, dans le cas d'une enquête postérieure, de mieux comprendre ce qui s'est passé. Ce RFC décrit une architecture <https://scitt.io/>, très inspirée du "*certificate transparency*" du RFC 9162 [1].

Vous voulez des exemples d'attaques contre la chaine d'approvisionnement du logiciel ? Deux cas récents sont la faille Trivy / Aqua Security <https://www.aquasec.com/blog/trivy-supply-chain-attack-wha
>, annoncée le 1 avril de cette année mais qui n'est pas une blague (tout juste de l'ironie car le logiciel Trivy servait à détecter les problèmes de sécurité…) Cette attaque a notamment permis de l'accès aux données de la Commission Européenne <https://cert.europa.eu/blog/european-commission-cloud-breac
Et il y a aussi, encore plus récente, l'attaque contre le paquetage Python litellm <https://www.truesec.com/hub/blog/malicious-pypi-package-litellm-supply-chain-compromise>. Mais le cas le plus fameux, quoique un peu moins récent, est celui du détournement de la bibliothèque XZ. (Olivier Poncet en a fait une conférence <https://www.emaxilde.net/posts/2024/10/11/anatomie-d-une-faille.html>.)

Les attaquants peuvent frapper à de nombreux endroits différents dans la chaine d'approvisionnement d'un logiciel. Cela peut être en modifiant une bibliothèque utilisée par le logiciel, en modifiant directement le code source, en modifiant l'environnement de compilation (de manière à produire un

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9162.txt

code exécutable malveillant à partir d'un code source intact), en remplaçant le code exécutable dans un magasin d'applications, etc. La figure 1 dans la section 2.1 du RFC donne une liste complète. Le choix va dépendre des capacités de l'attaquant, de ses choix de discrétion, et des solutions de sécurité déployées.

Je l'ai dit au début, il n'y a pas de solution parfaite au problème, qui est complexe. Dans le cas de XZ, le mainteneur a eu tort de faire confiance à un inconnu mais, si on arrêtait d'accepter des volontaires inconnus, tout le logiciel libre s'effondrerait. Et le logiciel privateur a d'autres failles, comme l'opacité, qui empêche de savoir quels sont les composants logiciels intégrés.

Que propose le groupe de travail SCITT <https://datatracker.ietf.org/wg/scitt/about/>, auteur de ce RFC ? L'approche choisie est celle de la transparence et de la traçabilité <https://scitt.io/>. Il faut qu'on puisse vérifier de quels logiciels on dépend. La principale source d'inspiration est le "*Certificate Transparency*" du RFC 9162 et l'architecture SCITT <https://scitt.io/> peut être vue comme une généralisation de "*Certificate Transparency*" au logiciel. Il repose sur une structure de données ordonnée, et qui garantit que seul l'ajout de données est possible, et que tout dans cette structure de données est signé. (Si vous criez « chaine de blocs » ici, vous avez tort : une chaine de blocs fournit en effet ce service mais elle n'est pas nécessaire, d'autres mécanismes existent depuis longtemps, tels que ceux utilisés par le RFC 9162.) D'autre part, il est important de se souvenir que notre RFC ne présente qu'une architecture, pas un protocole complet. Les programmeurs et programmeuses ne peuvent donc pas se mettre au travail tout de suite. L'architecture SCITT est conçue pour les chaines d'approvisionnement de logiciel mais peut a priori être étendue plus tard pour d'autres usages.

Le mécanisme de base consiste en des données en CBOR (RFC 8949), dont la structure est spécifiée en CDLL (RFC 8610), signées avec COSE (RFC 9052) et récupérable via des arbres de Merkle (RFC 9942). Grâce à cela, celui ou celle qui veut vérifier des informations sur un logiciel peut les récupérer facilement et efficacement, puis les vérifier (là encore, comme avec "*Certificate Transparency*"). C'est par exemple ce que le NIST appelle DevSecOps <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204C.pdf>.

La section 2.2 du RFC fournit trois cas d'usages de cette technologie. L'un d'eux concerne l'assemblage du logiciel pour un véhicule. (Dans le monde réel, les gens qui assemblent le logiciel pour une voiture ne se soucient guère de sécurité ; comme il y a zéro conséquence négative pour eux en cas de bogue, ou même de faille de sécurité, ils utilisent des versions antédiluviennes <https://arxiv.org/abs/2208.14367> de logiciels non testés <https://www.sciencedirect.com/science/article/abs/pii/S0167404822004606>.) La chaine de dépendance est longue car une voiture est faite de plusieurs composants, chacun apportant du logiciel qui, à son tour, dépend de diverses bibliothèques. SCITT permettrait au moins d'avoir des idées claires sur les composants logiciels utilisés, chacun ayant fait l'objet d'un ajout dans la structure de données de traçabilité.

Bon, maintenant, si on veut être un peu plus précis et comprendre ce qu'est l'architecture SCITT, par quoi on commence ? Par le vocabulaire, peut-être. Découvrons-le dans la section 3 du RFC :

— Déclaration ("*statement*") : une information portant sur un fait (par exemple « la version actuelle est la 3.14 » ou bien « ce logiciel dépend de libfoobar et de grutzbaum-plus »). Les logiciels peuvent être identifiés par un grand nombre d'identificateurs, notre RFC cite entre autres CoSWID (RFC 9393), in-toto <https://in-toto.io/>, SPDX, SWID <https://csrc.nist.gov/Projects/Software-Identification-SWID/guidelines>, etc.
— Enveloppe ("*envelope*") : les méta-données comme l'identité de l'émetteur.
— Émetteur ("*issuer*") : l'entité qui déclare quelque chose, et qui a une clé privée pour signer cette déclaration.
— Utilisateur ("*relying party*") : l'entité qui va utiliser les déclarations.
— Équivoque ("*equivocation*", quoiqu'il me semble que le terme soit moins fort en français) : déclarations contradictoires (ce qui est clairement mauvais).

— Reçu (”*receipt*”) : preuve comme quoi une déclaration signée a bien été incluse dans la structure de données. Si on les distribue sur l'Internet, le type de média à utiliser est `application/scitt-receipt+cose`
— Déclaration signée (”*signed statement*”) : ce sont elles qui sont encodées en COSE (RFC 9052). Si on les distribue sur l'Internet, le type de média à utiliser est `application/scitt-statement+cose`.
— Structure de données vérifiable (”*verifiable data structure*”) : là ou on met les déclarations. Cette structure de données doit être à ajout seul (et, je me répète, une chaine de blocs a cette propriété mais les chaines de blocs ne sont nullement indispensables pour cela), doit empêcher l'équivoque (cf. section 5.1.3) et doit permettre l'examen complet (peut-être après autorisation) pour vérifier sa cohérence. Le RFC 9942 donne des détails techniques.

Armé de ces termes, nous pouvons décrire l'architecture SCITT (mais regardez aussi la figure 2 du RFC) :

— Des émetteurs produisent des déclarations signées et les envoient à des services de transparence,
— ces services de transparence authentifient l'émetteur, vérifient les signatures, vérifient la conformité des déclarations à leur politique (cf. section 5.2.1), mettent la déclaration dans la structure de données vérifiable et produisent un reçu (suivant le RFC 9942),
— des utilisateurs vont accéder à ces informations pour vérifier les informations (« est-ce que je dépend ou pas de libfoobar ? », sachant que la dépendance peut être transitive), ou bien pour vérifier la cohérence du service de transparence, par exemple lors d'un audit (c'est une propriété importante d'un service de transparence, on doit pouvoir vérifier son intégrité).

Il peut y avoir plusieurs services de transparence (comme c'est le cas pour ”*Certificate Transparency*”), personne n'envisage évidemment d'avoir un unique point de dépôt des déclarations.

Le format des déclarations signées et des reçus figure (en CDDL, cf. RFC 8610) en section 6.1.

La section 9 du RFC est vraiment à lire car c'est celle consacrée à l'analyse de sécurité de l'architecture SCITT. D'abord, il faut être bien conscient que SCITT fournit de la transparence et de la traçabilité mais pas de la vérité. Si un émetteur signe que son logiciel tourne avec toutes les versions de libfrobnicate mais qu'en fait il lui faut au moins la version 2, le service de transparence ne va pas regarder les sources du logiciel pour vérifier. C'est vrai pour les erreurs des émetteurs et encore plus pour leurs malhonnêtetés. « ”*Transparency does not prevent dishonest or compromised Issuers, but it holds them accountable*” ».

Un émetteur grognon ou négligent peut aussi ne pas enregistrer toutes les informations qu'il a. Un utilisateur ne doit donc pas utiliser une déclaration signée sans vérifier le reçu, qui seul prouvera que la déclaration a été enregistrée.

Il existe apparemment au moins trois mises en œuvre de cette technique, chez deux entreprises spécialisées dans la traçabilité des chaines d'approvisionnement, Datatrails <`https://docs.datatrails.ai/developers/developer-patterns/scitt-api/`> et Tradeverifyd <`https://tradeverifyd.github.io/transparency-service/`>, mais aussi chez Microsoft <`https://github.com/microsoft/scitt-ccf-ledger`> (cf. leur article <`https://azure.microsoft.com/en-us/blog/enhancing-software-su`
>).

Enfin, une bonne lecture sur ces structures de données vérifiables est « ”*Attested Append-Only Memory : Making Adversaries Stick to their Word*” <`https://www.read.seas.harvard.edu/~kohler/class/08w-dsi/chun07attested.pdf`> ».