Les menaces liées aux dépendances des logiciels

Par Vincent GIRAUD
Chaire cyber et souveraineté numérique - IHEDN

Lors du développement de produits informatiques, les dépendances dont font preuve les logiciels produits sont un point de vigilance omniprésent chez les éditeurs et concepteurs. Si elles ont l'avantage d'accélérer ainsi que simplifier massivement le travail, et si elles peuvent permettre de bénéficier d'exécutables et de contenus à la fois robustes et éprouvés, celles-ci apportent également leur lot de menaces.

Ces dernières sont présentes à la fois sur les dépendances qui s'appliquent lors du développement comme sur celles qui se montrent à l'utilisation du logiciel. Par ailleurs, estimer qu'elles sont systématiquement le résultat d'une volonté malveillante ou d'une attaque explicite serait une erreur : beaucoup découlent simplement des caractéristiques même du processus de réutilisation ou d'exploitation de contenus externes. Les menaces et les risques associés sont ainsi particulièrement protéiformes.

Introduction

La production logicielle, de par son caractère essentiellement intellectuel, permet des formes de réutilisation et d'usages particulièrement avancés. Dès les premières versions de tous les systèmes d'exploitation multitâches majeurs, la possibilité d'embarquer et de profiter de briques logicielles tierces pendant le développement a été fournie aux concepteurs, soit *via* une intégration totale lors de la création (lien statique), soit en prévision d'une intégration ultérieure (lien dynamique). Par ailleurs, et de manière générale, l'interconnexion croissante des sociétés démontre un deuxième type d'appui logiciel : celui ayant lieu lors de l'utilisation. Les programmes s'utilisent entre eux *via* des interfaces de programmation, que ce soit localement, au sein d'une même machine, ou à distance, *via* un quelconque réseau.

Ces relations de dépendances se sont rapidement imposées comme des postulats de l'informatique moderne. Les cadres techniques et juridiques autour d'elles se sont développés en les considérant comme évidentes. Les premiers lorsque les systèmes d'exploitation se sont adaptés et lorsque la conception d'interfaces de programmation binaires ou applicatives s'est révélée être un savoir primordial dans ce domaine. Les seconds quand le développement de contrats de licence prenant directement en compte ces modes d'utilisation s'est popularisé.

En parallèle, le développement d'internet et des réseaux a largement facilité le partage de code et d'exécutables, notamment à travers les frontières. La mise en place de dépendances logicielles s'est alors multipliée dans une certaine insouciance. C'est seulement plus tard, lors de l'apparition sur la scène mondiale de plusieurs incidents majeurs, qu'un nouvel éclairage sur cette pratique permettra une remise en question sur le rapport bénéfice-risque, d'abord d'un point de vue purement technique. Les récentes transformations géopolitiques et l'évolution des rapports de force causeront une projection de

cette problématique sur le plan de la souveraineté, révélant ainsi tout l'enjeu d'autonomie stratégique qu'elle contient : dans un monde aussi numérique que le nôtre, une certaine indépendance logicielle est nécessairement un socle dont on doit disposer.

Les avantages et les risques associés aux dépendances logicielles varient selon qu'on les considère lors du développement, ou lors de l'utilisation.

Dépendances de développement

Lors du développement d'un logiciel, le concepteur peut en intégrer d'autres en son sein. Cette pratique est, on l'a vu, omniprésente aujourd'hui ; elle est d'ailleurs récursive : les logiciels intégrés ont eux-mêmes d'autres logiciels intégrés en eux. Ce phénomène de lien indirect, appelé « dépendance transitive », mène souvent à une embarcation en cascade de briques logicielles. Les chiffres présents dans un rapport de 2025 de la société Black Duck[1], spécialisée dans la sécurité des chaînes d'approvisionnement logicielles, sont éloquents. Sur 1 658 projets logiciels analysés, 97 % contiennent des briques logicielles externes en sources ouvertes. En moyenne, il y en a 911 par projet, et 64 % d'entre elles sont présentes par dépendance transitive. Les composants logiciels externes représentent au final la majorité du code : 70 % en moyenne à l'échelle d'un projet.

Il y a plusieurs risques associés aux dépendances de développement. Tous ne découlent pas d'une démarche offensive : certains résultent davantage d'événements comme l'abandon d'un projet, qu'il soit spontané ou causé par un manque de support financier ou institutionnel. On peut également mentionner les difficultés juridiques qui s'imposent lors de la réutilisation de code provenant d'un autre pays. On se concentrera ici sur les trois risques les plus prégnants dans la gestion logicielle : les failles dans le travail de maintenance, la perte de maîtrise ainsi que du savoir-faire, et l'insertion de portes dérobées.

Maintenance

Le plus présent au quotidien est celui lié à la maintenance. Chaque brique logicielle doit généralement être incluse au sein d'un projet dans une version précise. Lorsqu'elles sont mises à jour par leurs développeurs respectifs, le concepteur d'un projet doit rapatrier ces nouveautés : au-delà de nouvelles fonctionnalités pas forcément utiles, elles peuvent également apporter des correctifs face à des failles ou des risques de sécurité découverts en elles.

L'exemple des failles de sécurité découvertes dans la bibliothèque logicielle log4j est le plus parlant. Permettant de gérer les journaux et registres de projets écrits dans le langage Java, elle est particulièrement répandue. Une série de failles rendues publiques en 2021 rendront concrets des vecteurs d'attaque résultant notamment en des exécutions de code arbitraire. La gravité de leurs conséquences et l'échec dans le processus de divulgation responsable causeront d'abord un empressement mondial à identifier puis sécuriser les projets touchés, ensuite, une vague de remises en question sur la forme de plus en plus tentaculaire qu'adoptent les conceptions logicielles de par la masse de dépendances insoupçonnée qu'elles accumulent.

L'entretien, le suivi, la mise à jour et la correction des dépendances d'un projet implique nécessairement un travail manuel : il s'agit au mieux de simplement vérifier le bon fonctionnement par la suite, ou au pire, en cas de non-rétrocompatibilité, de reprogrammer l'utilisation du composant. Bien que l'on puisse en espérer une plus grande automatisation à l'avenir, ce suivi reste coûteux. On estime que dans le domaine du développement

[1] Black Duck (2025), "Open source security and risk analysis report".

logiciel, la majorité des coûts ne sont pas attribués à la conception initiale, mais plutôt à la maintenance[2].

Maîtrise et savoir-faire

Rajouter une dépendance de développement à son projet implique de déléguer une fraction algorithmique de son application à un composant tiers embarqué. Le concepteur initial conserve-t-il une totale maîtrise de cette partie de logique ? Faire une telle délégation est souvent motivé soit parce qu'on ignore comment réaliser la tâche concernée, soit parce qu'on la juge commune au point qu'il serait une erreur de la réimplémenter individuellement. Dans le premier cas, on ne maîtrise pas le procédé de base, dans le second, on décide de renoncer au savoir-faire associé en faisant le pari qu'il sera facilement assimilable si besoin est. Ce raisonnement, qui n'est pas dénué de viabilité, incite cependant à multiplier les dépendances logicielles de développement, au point qu'il devient difficile de les contrôler, surtout en considérant celles qui sont transitives.

L'historique des incidents récents tend à montrer qu'un imprévu auprès d'une brique logicielle populaire peut avoir de larges répercussions, que son contenu logique soit complexe ou non. En 2016, le développeur Azer Koçulu a supprimé 273 composants logiciels qu'il avait publiés sur npm, la principale plateforme de gestion de paquets pour le langage JavaScript. L'un d'entre eux était *left-pad*. Son rôle était trivial : procéder à un bourrage de caractères par la gauche dans un texte. Un grand nombre de concepteurs ont malgré tout préféré rajouter cette dépendance plutôt qu'implémenter la logique par eux-mêmes : le mois qui a précédé son arrêt, *left-pad* avait été installé environ 2 500 000 fois, et était utilisé dans des milliers de composants en sources ouvertes, eux-mêmes potentiellement vastement utilisés. Cet exemple est emblématique des risques portés par les dépendances transitives : *left-pad* était nécessaire pour babel, composant extrêmement populaire, qui lui-même était nécessaire pour React Native, lui aussi extrêmement populaire. Ce type de chaîne de dépendances, ici raccourcie, provoque une exponentiation de la surface d'impact : d'autres projets vastement utilisés comme ember ou atom ont également été touchés. La reprise des activités n'a pas été longue après cet incident car *left-pad* était un composant extrêmement simple (il n'était composé que de 11 lignes de code), mais le champ d'action révélé a néanmoins causé une véritable prise de conscience sur la maîtrise des projets informatiques et sur le savoir-faire dont il faut faire preuve vis-à-vis de leur code source.

Portes dérobées

Un autre risque lié aux dépendances logicielles est plus axé sur la sécurité informatique, dans la mesure où, lui, découle réellement d'une volonté hostile et impacte plus facilement la confidentialité. Il s'agit de l'insertion active de portes dérobées, réutilisées et importées dans les projets par les concepteurs, à leur insu. Celles-ci donnent secrètement à des acteurs malveillants un accès privilégié à des ressources dont l'accès est normalement restreint.

Les chaînes de dépendance pouvant être particulièrement longues et pouvant contenir un large nombre de branches, un composant élémentaire compromis peut obtenir à sa disposition un grand terrain d'action. On peut alors assimiler un tel risque à celui d'une ingérence logicielle. Lorsqu'une telle tentative est découverte, une course contre la montre

[2] Glass R. L. (2001), "Frequently Forgotten Fundamental Facts about Software Engineering", *IEEE Software*, n°3.

est lancée pour identifier toute présence de la brique logicielle compromise parmi toutes celles sollicitées dans un projet, et pour appliquer un correctif complet.

La tentative d'attaque autour de XZ Utils est emblématique de cette problématique et aurait certainement pu être la plus impactante à l'échelle mondiale si elle avait été menée à son terme. En 2024, Andres Freund, ingénieur au sein de Microsoft, a détecté des performances inhabituelles exhibées par son serveur SSH lors des authentifications de clients. En enquêtant sur la source de ce problème, il a découvert que XZ Utils, importé par la suite logicielle systemd, elle-même importée par le serveur SSH, influençait la vérification de certificat de ce dernier, sans raison légitime évidente. La suite des recherches a exposé un mécanisme complexe donnant à l'attaquant un vecteur menant potentiellement à un accès non autorisé ou à une exécution de code arbitraire. Ce montage, inédit par son mode opératoire et son ampleur visée, a été découvert prématurément, et n'a pas eu le temps d'atteindre les versions dites stables des distributions logicielles. Il a néanmoins soulevé de nombreuses interrogations sur la propagation voire la facilitation des portes dérobées par l'utilisation insouciante des dépendances logicielles. Une vaste réflexion a également eu lieu sur les conditions de développement des logiciels en sources ouvertes, et la viabilité de celles-ci. En effet, le processus d'attaque décrit est le résultat d'environ deux ans de manipulation sociale du seul développeur de XZ Utils, Lasse Collin, par un autre développeur malicieux, inconnu mais prétendument appelé Jia Tan. Lasse Collin peinant à soutenir le projet, et souffrant d'après ses propres termes de problèmes de santé mentale à long terme, s'est trouvé dans une situation douloureuse où il était plus susceptible d'accepter une main tendue.

Dépendances d'utilisation

Une fois un logiciel compilé, livré et installé, celui-ci peut avoir des dépendances envers d'autres logiciels durant l'utilisation. En utilisant une interface de programmation applicative *via* un système de communication inter-processus ou *via* des communications réseau, ce type de sollicitation a lieu de manière parfaitement intégrée, mais de façon plus ou moins transparente pour l'utilisateur.

L'enjeu de collaboration entre programmes locaux a acquis une importance majeure dès les années 2000 : c'est à cette période que les systèmes d'exploitation ont commencé à mettre en avant des mécanismes de communication inter-processus plus intelligents, pour répondre à la demande. Sur Linux, afin de disposer davantage de flexibilité que celle offerte par les simples sockets UNIX et les tubes nommés (pipes), on verra apparaître durant cette décennie D-Bus pour les serveurs ainsi que l'informatique bureautique, et Binder pour les téléphones fonctionnant sous Android, chacun motivés par les contraintes techniques associées à leur plateforme.

Sur le réseau, la normalisation de Common Gateway Interface (CGI), qui s'est étalée du début des années 1990 au milieu des années 2000, démontre le besoin associé : pouvoir appeler des instances de logiciels tiers potentiellement extrêmement éloignées géographiquement, avec des paramètres arbitraires, pour recevoir un résultat dynamique. Au-delà de CGI, ce paradigme s'imposera en ligne, en lieu et place du *web* classique dit « statique ».

Les risques associés aux dépendances logicielles lors de l'utilisation sont analogues à ceux des dépendances lors du développement, mais varient d'une certaine manière. Si l'on retrouve rarement des chaînes de dépendances hors de contrôle à l'utilisation, les développeurs peuvent malgré tout accuser un manque de compréhension et de maîtrise vis-à-vis de la logique de l'application globale. Ils doivent tout autant suivre les versions de leurs dépendances. Le problème de la disponibilité est cependant accru à l'exécution des logiciels : en plus de devoir s'assurer que toutes les dépendances sont toujours suivies et entretenues dans de bonnes conditions, il faut également veiller à leur maintien en exécution, puisqu'une erreur interrompant même temporairement leur fonctionnement

aura un impact sur le maintien du service. Le risque de porte dérobée est moindre dans la mesure où les dépendances ne retournent habituellement que des résultats d'opérations, mais peut avoir lieu dans des contextes cryptographiques ou lorsque que des exécutables sont retournés.

L'utilisation de dépendances logicielles sur internet, à l'exécution, présente un ensemble d'enjeux non techniques, mais juridiques et commerciaux. Le caractère international d'internet et la suppression des distances qu'il instaure facilite ces sollicitations logicielles entre entités de divers pays, ce qui s'assimile alors à une vente transfrontalière de service, impliquant ainsi les contraintes légales et réglementaires afférentes.

Conclusion

S'appuyer sur des composants logiciels tiers, que ce soit lors du développement ou lors de l'utilisation, permet de bénéficier d'une flexibilité accrue, et cette pratique a sans conteste été un vecteur de croissance considérable du secteur informatique. Il aura cependant fallu attendre plusieurs incidents techniques majeurs récents, ainsi qu'une reconfiguration géopolitique mondiale inédite pour une prise en compte profonde des risques associés. La connaissance exhaustive des dépendances d'un logiciel, caractérisée par la Software Bill of Materials (SBOM), devient de plus en plus une obligation légale. Un premier pas a été franchi par la Maison Blanche en 2021 avec son ordre exécutif numéro 14028, l'imposant à toute entité fournissant un logiciel au gouvernement fédéral. En Europe, le Cyber Resilience Act développe depuis 2024 cette même exigence, mais dans un champ plus large que celui des gouvernements : celui du marché européen dans sa globalité.