# Avons-nous besoin de Wayland ?

Océane*

*[2023-11-18 sam. 16:29]*

J'écris ce document depuis sway, un gestionnaire de fenêtres compatible avec Wayland. Je me suis intéressée à Wayland sans trop savoir pourquoi, et j'ai donc consacré quelques jours à reconfigurer mes touches pour mon clavier Bépo (j'ai partagé mon fichier de configuration ici). De manière générale, les ingénieur·es en informatique ne savent pas trop ce qu'iels font, notamment car les outils qu'iels utilisent suivent la philosophie Unix selon laquelle un outil ne remplit qu'une fonction (les systèmes d'exploitation Unix et le langage C ayant le même créateur, Dennis Ritchie) ; or leur travail consiste à manipuler un environnement de logiciels en interaction, ce qui ne peut pas se faire en ne lisant que la documentation d'appoint de chaque utilitaire, *via* les pages de manuel Unix : il faut lire des livres. Iels peuvent donc se mettre à telle technologie à la mode, comme Kubernetes, sans trop savoir à quoi cette technologie sert, c'est-à-dire, selon Ariadne Connill, à gérer des microservices, ce qui ne servirait à rien avec des logiciels monolithiques.

L'enjeu est ici le rasoir d'Occam : les multiples ne doivent pas être utilisés sans nécessité. Alors qu'est-ce qu'un multiple ? Une théorie complexe et ambitieuse, comme celle de l'*habitus*, qui peut être plus encombrante que nécessaire si l'on ne s'intéresse pas *a priori* à des situations d'emprise. L'achat d'un produit sur lequel sont réalisées des marges importantes, sans vraiment en avoir besoin. Ou, par exemple, un protocole entre le noyau d'un système d'exploitation (composant critique, notamment du point de vue de la sécurité) et chaque programme graphique, sur un modèle client/serveur : ainsi le protocole Wayland doit-il être implémenté dans le noyau Linux, ou encore dans le noyau FreeBSD, mais aussi au niveau de chaque programme graphique, à des fins de sécurité informatique.

L'enjeu est ici qu'un programme exécuté par un utilisateur Unix peut lire et modifier tous les fichiers que cet utilisateur est autorisé à lire et à modifier, et bien évidemment exécuter tous les programmes qu'il est autorisé à exécuter, ce qui signifie que si j'exécute un programme propriétaire, dont j'ignore le fonctionnement, il peut être utilisé pour faire fuiter des données confidentielles, par exemple des enregistrements

*oceane@disroot.org | océ.fr

d'entretiens sociologiques, alors que mes enquêté·es me font plus ou moins confiance pour respecter leur confidentialité. Par ailleurs, un navigateur web est un programme pouvant exécuter du code arbitraire (et opaque) à chaque visite de site web, et il est donc particulièrement vulnérable à des attaques informatique : n'importe quel site web inconnu est une menace potentielle, qui peut être utilisé pour tenter de compromettre Firefox et donc accéder à mes fichiers. La solution est évidente : il faut pouvoir exécuter Firefox en tant qu'utilisateur non-privilégié, n'ayant pas accès à mes enregistrements d'entretiens, tandis que mon utilisateur personnel aurait bien évidemment accès à mes téléchargements Firefox. Or, comme l'exécution de programmes graphiques dans les systèmes d'exploitation libres se fait *via* une bibliothèque, Xorg, tous les logiciels graphiques, dont mon environnement de bureau, mon navigateur, et par exemple Discord, doivent être exécutés par le même utilisateur : Wayland, en tant que protocole graphique, apporte une solution à ce problème.

Il convient ici de relever que Wayland, l'environnement de bureau Gnome, PulseAudio, Red Hat, et Fedora Linux sont développés et appartiennent à IBM. Notons aussi que le noyau Linux est principalement développé par des entreprises comme IBM, notamment car Linus Torvalds a fait fuir des contributaires compétent·es pendant des décennies en leur tombant dessus à la moindre coquille, ce qui a cédé le terrain de son développement à des salarié·es d'entreprises comme IBM, Red Hat, Facebook, ou Intel.

Évidemment, une solution alternative à Wayland peut être d'enregistrer et de lire ses entretiens avec un utilisateur séparé, aux données du quel notre utilisateur courant n'aurait pas accès, et avec lequel on s'interdirait d'exécutre des logiciels propriétaires. Mais dans un environnement 100 % libre, le projet OpenBSD a fourni une *autre* solution, correspondant à ses principes de simplicité : l'implémentation de deux appels systèmes, unveil(2) et pledge(2). unveil(2) est un appel système que l'on peut intégrer au code source d'un programme, et par lequel ce programme, dès son lancement, dit au noyau n'avoir besoin d'accéder qu'à certains répertoires, et de lui masquer tout le reste, merci beaucoup. Avec pledge(2), le programme dit par exemple ne pas avoir besoin de modifier des fichiers, et il serait donc en lecture seule, ou alors ne pas avoir besoin d'accéder à l'internet, ce qui empêcherait tout transfert de données lui étant accessibles. OpenBSD permettant d'installer les programmes sous forme exécutable « classique » ou sous forme de « ports », du code source que l'on peut modifier avant de le compiler (grâce à une méthode livrée avec), on peut ajouter ces appels système à Firefox, Emacs, et à à peu près tous les logiciels un peu complexes, et réputés pour les risques de sécurité qu'ils représentent, en excluant des données sensibles de leur arborescence. Par ailleurs, puisqu'il est possible d'exécuter ces programmes même en tant que racine, ce qui en temps normal donnerait à un·e attaquant·e un accès illimité à notre machine, sans que le noyau ne nous autorise à accéder à une arborescence non-autorisée ou même à modifier des données, ces appels systèmes repré-

sentent un filet de sécurité bienvenu dans des environnements d'entreprise capitalistes, baissant autant que possible les coûts de main-d'œuvre à travers un modèle productiviste afin d'augmenter les marges et donc les recettes, ce qui favorise, sans parler d'incompétence, *a minima* des erreurs humaines, dues au surmenage – on parle d'administrataires systèmes gérant des centaines voire des milliers de serveurs. Ces filets de sécurité, qui ne sont pas encore implémentés dans le noyau Linux, éviteraient à coup sûr de nombreuses failles de sécurité à l'étape non de l'exécution du programme mais de son développement.

Il va sans dire que la sécurité informatique a d'autres enjeux, par exemple une fuite de mémoire dans un logiciel de machine à rayons X a donné lieu à l'émission de 10 000 fois la dose voulue, ce qui a donné lieu à des cancers mortels ; mais les appels systèmes unveil(2) et pledge(2) ne permettent pas seulement d'exécuter des programmes en limitant leurs privilèges, tout en conservant Xorg, en n'ajoutant que quelques lignes au code source d'un programme : s'ils avaient été implémenté dans le noyau Linux peu après celui d'OpenBSD, en 2013 et en 2015, ils auraient peut-être pu éviter la fuite de données du site de rencontres adultères Ashley Madison, qui a conduit au suicide d'un homme – un salaud sans aucun doute, mais qui aurait mérité de vivre et de devenir une meilleure personne, et qui surtout est emblématique d'une précarisation plus générale de ses utilisataires, hommes comme femmes (notamment dans des pays en voie de développement, comme les États-Unis, qui ne connaissent pas d'assurance obligatoire contre le chômage, sous forme de salaire secondaire, et donc pas de filet de sécurité en cas de licenciement ou de divorce). Le protocole Wayland paraît donc à peine utile, afin d'exécuter des logiciels graphiques propriétaires, comme Discord, dans des environnements graphiques où l'on traite des données sensibles par des utilisateurs non-privilégiés, sous réserve d'une implémentation directement dans le *framework* Electron. Tout ce travail me paraît bien complexe, pour des résultats promis en réalité improbables et, le cas échéant, plutôt minces.