GNU GUIX, VERS UNE GESTION DE PAQUETS SÉCURISÉE

Édouard KLEIN
Lead Data Scientist à Sekoia, Fondateur de Beaver Labs - edou@rdklein.fr

GNU Guix [GUIX] se présente comme un « gestionnaire de paquets purement fonctionnel ». Explorons ce que cela veut dire. Nous verrons que GNU Guix peut remplacer avantageusement aptitude, pip, virtualenv, Docker, Jenkins, et Ansible.

1. EMPAQUETAGE REPRODUCTIBLE

L'empaquetage consiste à sérialiser en un fichier les informations nécessaires à l'installation d'un logiciel. Il faut prendre en compte son code source, ainsi que le compilateur ou l'interpréteur et les librairies.

Les éléments nécessaires au fonctionnement sur la machine cible sont les dépendances d'exécution (interpréteurs, librairies dynamiques) tandis que ceux nécessaires à la création du paquet sont les dépendances de construction (compilateurs, librairies statiques).

L'empaquetage reproductible consiste à obtenir bit à bit le même paquet, quelles que soient les conditions d'exécution du processus d'empaquetage, qui doit être déterministe : produisant toujours les mêmes résultats à partir des mêmes entrées. Le processus d'empaquetage est une pure fonction.

La valeur de sortie (le paquet) ne dépend que des entrées : le code source du logiciel, et ses dépendances d'exécution et de construction. On y fait donc référence par un hash calculé sur les entrées.

Comme certaines de ces entrées (les dépendances) sont également des paquets, le nom d'un paquet est un hash dont la valeur dépend du nom/hash des dépendances, qui dépend du hash de leurs dépendances, et ainsi de suite.

La structure de donnée ainsi créée s'appelle Graphe Acyclique Dirigé de Merkle (Merkle-DAG). Bien connu en cryptographie, il permet de valider l'intégrité de n'importe quel noeud. Les développeurs le connaissent puisque chaque repo git est un Merkle-DAG.

2. REPRODUCTIBILITÉ ET SÉCURITÉ

En 2018, le fiasco **[LEFT-PAD]** fait découvrir à des informaticiens ayant plus de bon sens ou d'expérience que l'utilisateur moyen de `npm` la surprenante popularité de paquets triviaux comme `left-pad`, `is-odd`, ou `is-number`.

Il est facile à l'un de ces paquets d'ajouter du code malveillant à un grand nombre d'applications **[MERCY] [HACK]**.

Une meilleure gestion de la confiance améliore le rapport impact/coût, mais même Debian n'est pas à l'abri **[PRNG]**.

La construction reproductible de paquet mitige ce genre d'attaques en rendant explicite, par un changement de hashes en cascade, le changement d'une dépendance aussi lointaine et triviale soit-elle dans le Merkle-DAG.

Un autre type d'attaque plus difficile à mener, mais excessivement plus difficile à détecter a été imaginé dès 1984 par Ken Thompson et exposé dans son allocution de réception du Prix Turing **[KEN]**. Il s'agit d'empoisonner le compilateur pour qu'il insère des portes dérobées dans des logiciels cibles, y compris les autres compilateurs qu'il compile.

Contrer ces attaques requiert un audit du binaire du compilateur (le code source du compilateur peut être valide, mais s'il est compilé avec un compilateur empoisonné, alors il est empoisonné aussi). GNU Guix essaye de réduire la quantité de binaires nécessaire à la création initiale du premier compilateur. Ces efforts herculéens parviennent aujourd'hui à bootstraper la chaîne logicielle complète à partir de seulement 60Mo de code binaire, l'objectif étant de parvenir à utiliser les Équations de Maxwell du Logiciel et hex0, un assembleur de seulement 500 octets [**BOOTSTRAP**] !

Intégrer ce paquet hex0 à la base du Merkle-DAG permet d'éliminer les attaques décrites par Thompson.

3. QUARANTE ANS D'HISTOIRE

Les logiciels UNIX ont d'abord voyagé physiquement sur des cassettes. Installer une fois suffisait à mettre le logiciel à disposition de tout le campus.

En 1982, Unix System III inclut PKG_ADD, la première tentative de maintien d'une base de données des logiciels installés et de la localisation des artefacts correspondants. En 1983, 4.3BSD est doté de SPMS, un proto-gestionnaire de paquets. Le code source des logiciels libres commençait également à être distribué via Usenet.

À partir de 1992, les premières distributions Linux (SLS, puis Slackware) proposent un gestionnaire de paquets basé sur tar. En 1993, FreeBSD 1.0 étend le gestionnaire System III avec ports, le premier système à réunir la gestion de patches, la construction des paquets et la gestion des dépendances. Gentoo s'en inspire à partir de 1999.

Le premier commit de rpm date de 1995, alors que le travail sur dpkg commence en 1994. Apt est bâti par-dessus à partir de 1998 et apporte par exemple la détection des dépendances circulaires **[TUHS]**.

Aujourd'hui, chaque langage ou presque dispose de son gestionnaire de paquets (gem, cargo, npm, pip, etc.), qui vient s'ajouter à celui du système hôte.

En 2004 paraît le papier fondateur de Nix **[NIX]**, qui pose les bases utilisées par Guix, qui naît en 2012.

4. GNU GUIX : GUILE + NIX

Architecture

Guix partage l'architecture de Nix : chaque élément est stocké dans un sous dossier du store dont le nom commence par le hash des entrées ayant créé cet élément.

Ainsi, peuvent cohabiter dans le store deux paquets différant uniquement par une dépendance, même lointaine, car leur hash, donc leur sous-dossier, sera différent.

Le store ne peut être modifié que par un démon privilégié, qui réalise les étapes de construction des paquets dans un conteneur isolé du système hôte.

Cet isolation, destinée à garantir le déterminisme du processus de construction, a pour effet de bord de protéger de tout code malicieux présent dans le système de construction.

Scheme : un langage homoiconique

GNU Guix se démarque de Nix et acquiert ses super pouvoirs grâce au choix de Guile, une implémentation de Scheme pour décrire les paquets. « Guix » est la contraction de « Guile » et « Nix ». Ce choix efface la frontière entre le code servant à décrire un paquet et le code du gestionnaire de paquet lui-même.

Utiliser un langage de programmation plutôt qu'un langage de balisage comme XML, YAML, JSON est un avantage certain partagé par exemple par `pip` : le fichier `setup.py` est lui aussi du code Python, qui peut lister les fichiers d'un module avec un `glob.glob('**/*.py', recursive=True)` au lieu de les spécifier un à un.

La révolution vient de l'homoiconicité du Scheme : sa capacité, par sa syntaxe, à ne pas faire de différence entre le code et les données.

Dans un langage hétéroiconique comme le Python, à la frontière qu'est un appel de fonction tous les paramètres sont évalués dans le contexte de l'appelant, et l'appelé ne dispose que des valeurs, sans possibilité de savoir comment elles ont été construites.

À l'inverse, un langage homoiconique peut placer non pas des valeurs, mais du code en argument. L'appelé peut évaluer ses paramètres, ou non, ou plusieurs fois, ou les modifier avant de les évaluer. En Scheme, c'est le système des macros qui permet cela.

Dans le contexte de la construction de paquet, GNU Guix définit un parallèle au système des macros : les G-Expressions. Elles fournissent du code à évaluer dans le conteneur où a lieu l'empaquetage au démon privilégié qui seul peut manipuler le store.

Souvent, la construction d'un logiciel utilise des chemins codés en dur. Le « bon » chemin est un objet du store, préfixé par un hash qu'on ne peut

connaître dans le contexte de l'écriture du paquet. Le code de la G-expression exprime la correction à apporter aux chemins, la valeur de remplacement étant évaluée par le démon au moment de la construction.

Ce code peut recevoir des arguments, comme par exemple la version de Java ou Python à utiliser, et l'on construit plusieurs paquets pour le prix d'un.

Ces capacités d'introspection et de modification à la volée du code des paquets permettent à GNU Guix de dépasser l'existant sur tout le cycle de vie du logiciel, de son écriture à son déploiement auprès des utilisateurs.

5. ENVIRONNEMENT DE DÉVELOPPEMENT : GUIX VS. VIRTUALENV

Pour éviter les incompatibilités entre modules, les développeurs Python maintiennent dans un fichier `requirements.txt` la liste des modules dont ils ont besoin et créent un environnement virtuel où les installer. L'on « entre » dans cet environnement en exécutant un script qui va fixer les variables d'environnement adéquates.

La proposition de GNU Guix est similaire : rédiger un manifeste et l'utiliser pour créer un profil dans lequel on « entre » en exécutant un script.

Guix a cependant plusieurs avantages sur `virtualenv` :

- comme les objets sont stockés dans le store, commun à tous les utilisateurs, et non dans un dossier spécifique à l'environnement virtuel, si deux profils référencent le même paquet, celui-ci ne sera installé qu'une fois. Cette déduplication sauve beaucoup d'espace.
- GNU Guix n'est pas limité à Python, il est possible de mettre dans son manifeste des paquets écrits en n'importe quel langage.
- Ces paquets sont trivialement importables grâce à `guix import` depuis les autres gestionnaires les plus populaires : npm, cargo, etc.

6. EMPAQUETAGE : GUIX VS. APT

Les super pouvoirs de l'homoiconicité

Un paquet `.deb` est créé grâce à un paquet source référençant les dépendances par le nom de leur paquet `.deb`, qui est inscrutable.

C'est le cas hétéroiconique : les dépendances ont été évaluées, elles sont fixées, il n'est plus possible d'aller les modifier à la volée.

GNU Guix permet de remonter tout le graphe acyclique dirigé des dépendances et d'y opérer des modifications.

Ainsi, on peut composer des paquets en héritant d'un autre paquet et en changeant récursivement certains éléments. Un exemple typique est la définition d'un paquet Python utilisant Python 2 à partir de la définition utilisant la version 3.

Le parcours de l'arbre des dépendances est utilisé pour vérifier la présence de vulnérabilités dans la base des CVE ou pour vérifier et déclencher l'archivage du code source sur Software Heritage **[SH]**.

Enfin, GNU Guix permet de rédiger ses paquets dans un langage moderne et cohérent, tandis que le format `.deb` traîne le poids de trente années de bagage historique. Par exemple, créer un répertoire s'écrit `(mkdir "toto")` pour GNU Guix alors que pour un `.deb` il faut ajouter la ligne `toto` dans le fichier `debian/dirs`.

Isolation

GNU Guix propose deux outils pour vérifier le déterminisme de la construction des paquets :

- construire N fois les paquets afin de détecter des différences entre les N instances ainsi construites sur une même machine ;
- confronter bit à bit son résultat au résultat d'une autre installation de GNU Guix.

Ce souci de la recherche du déterminisme par l'isolation du processus de construction a rendu très facile pour l'utilisateur la compilation croisée, il suffit de passer l'option `–target`.

7. INSTALLATION : GUIX VS. PIP

Gestion des conflits

L'outil `pip` ne permet pas l'installation de deux versions d'un même paquet `toto`, car elles vont occuper le même dossier (`.../lib/python3.X/site-packages/toto`). L'utilisation du hash comme préfixe règle ce problème dans GNU Guix.

De plus, GNU Guix peut installer dans un même profil plusieurs paquets dépendant de versions incompatibles d'un même autre paquet, tant qu'une seule de ces dépendances est propagée, c'est-à-dire visible dans l'environnement de l'utilisateur.

Chaque exécutable est enrobé dans un script qui fait pointer les variables idoines (`PATH`, `LD_LIBRARY_PATH`, etc.) vers les sous-dossiers du store correspondants exactement aux dépendances non propagées. Il n'est nécessaire de créer deux profils distincts que si deux paquets incompatibles souhaitent être vus sous le même nom (par exemple, une commande dans le `PATH`) par l'utilisateur.

Atomicité et retour arrière

GNU Guix fonctionne par transactions atomiques. Il est possible de revenir à toute version antérieure d'un profil. L'appel au ramasse-miette devant être fait explicitement, tant que le disque a de la place on ne peut casser son environnement.

8. PUBLICATION

Pour mettre ses paquets à disposition d'autres machines, il suffit d'en héberger la description sur un repo git. Ils peuvent ainsi venir compléter en tant que canaux la distribution officielle.

Il existe des canaux communautaires proposant des logiciels spécifiques au calcul hautes performances, à la bio-informatique, et contrairement aux valeurs éthiques et sécuritaires de GNU Guix, proposant des logiciels propriétaires.

D'autres canaux sont dédiés aux jeux vidéos, pour préserver de manière durable ce pan de notre culture.

La durabilité est en effet une caractéristique des paquets créés avec GNU Guix. Les canaux sont référencés par l'adresse du repo git et le hash du commit utilisé. Les quelques lignes renvoyées par `guix describe` suffisent à fixer au bit près le code utilisé pour construire les paquets actuellement déployés. En copiant cette description sur une autre machine, on peut y reproduire exactement le même environnement.

Il est même possible de réunir dans le même profil des paquets construits avec des versions différentes de GNU Guix.

La machine cible n'a pas à reconstruire tous les paquets. Par exemple, mon VPS ne disposait pas d'assez de RAM pour compiler le compilateur du langage que j'utilisais pour un projet.

GNU Guix propose `guix publish` comme mécanisme pour partager le résultat binaire de la construction d'un paquet entre deux machines. J'ai pu construire le paquet sur une machine plus puissante et le faire passer sur le VPS :

```
$ # Authentifier la machine puissante auprès
du vps:
$ guix archive --generate-key
$ ssh root@vps guix archive --authorize <
/etc/guix/signing-key.pub
$ # Publier sur la machine puissante, et
utiliser un pont ssh pour que le VPS
$ # puisse y chercher les paquets binaires.
$ guix build MY-PACKAGE
$ guix publish&
$ ssh -N -R 8081:localhost:8080 user@vps&
$ ssh user@vps guix build --substitute-
urls=http://localhost:8081 MY-PACKAGE
```

En cas d'urgence, par exemple pour patcher une vulnérabilité critique, il n'est pas nécessaire de reconstruire tous les paquets qui dépendent du paquet corrigé : le binaire de celui-ci peut être appliqué à la place du paquet vulnérable dans le store et sera naturellement utilisé par les paquets en dépendant. Ce mécanisme s'appelle la greffe.

9. DÉPLOIEMENT

GNU Guix propose également d'autres mécanismes de déploiement.

Tarball

Il est possible de créer une archive contenant un paquet et ses dépendances d'exécution. Il est également possible de spécifier des liens vers des endroits standards comme `/usr/local/bin`, afin que le logiciel une fois décompressé soit disponible dans l'environnement de la machine cible, si elle ne dispose pas de Guix.

Guix vs. Docker et virtualisation

GNU Guix permet de créer des images Docker et des images de machines virtuelles (VM).

En contrepartie d'une perte de souplesse (une image n'est pas modifiable, elle doit être entièrement reconstruite) et de l'impossibilité de

mutualiser dans le store un paquet volumineux utilisé par plusieurs logiciels, cela permet à Guix de s'intégrer nativement dans une architecture reposant sur Docker comme par exemple Kubernetes. Il est plus avantageux de construire ses images Docker avec GNU Guix qu'avec un `dockerfile` :

- l'utilisation d'un langage de programmation homoiconique procure à GNU Guix les mêmes avantages que sur apt ;
- les images sont reproductibles, alors que les dockerfile utilisent souvent des commandes comme `apt-get install ...` ou `pip install ...` dont le résultat dépend de l'instant où elles sont exécutées.

Mais il n'est pas nécessaire d'avoir recours à Docker ou encore pire (en termes de performance et de complexité) à la virtualisation. La commande `guix environment` permet de rendre invisible le reste du système à la commande que l'on souhaite lancer, et réciproquement l'on peut protéger le système du logiciel en le conteneurisant grâce à l'option `--container` qui peut être invoquée par un utilisateur, à la différence de Docker ou d'un hyperviseur dont le démon est privilégié et donc présente une surface d'attaque.

10. ORCHESTRATION : GUIX VS ANSIBLE

Déclaration système

GNU Guix permet de spécifier un système d'exploitation complet de manière déclarative, comme l'on définit un paquet.

Cette déclaration, qui contient la configuration des logiciels à lancer, se transforme en image Docker, conteneur, ou en VM, fonctionnant sous la houlette de GNU Shepherd, qui remplace avantageusement systemd.

Les avantages sont similaires à ceux que GNU Guix possède sur apt, pip et Docker :

Tout ce qui est possible avec Ansible l'est avec GNU Guix, mais si la configuration que l'on souhaite appliquer n'est pas disponible dans l'API, Ansible doit se rabattre sur des commandes shell, alors que les G-expressions spécifient de manière déclarative (et donc modifiable et composable) les transformations souhaitées.

De plus, le système ainsi déclaré est reproductible, alors que des playbooks Ansible ne vont pas garantir le même résultat à chaque fois qu'ils sont appliqués. Particulièrement lorsqu'on enlève une fonctionnalité : en plus de la retirer des playbooks Ansible, il faut penser à nettoyer les traces (e.g. fichiers de configuration) qu'elle a laissées sur la machine hôte.

On ne peut pas « casser la prod » avec GNU Guix, puisqu'on peut revenir à l'état antérieur en une commande et que les opérations sont là aussi atomiques.

Guix deploy

La commande `guix deploy` permet de spécifier de manière déclarative pour tout un parc les services et logiciels à installer sur chaque machine. Il faut alors que tout le parc utilise la distribution Linux Guix SD.

11. INTÉGRATION CONTINUE : GUIX VS. JENKINS

Les hooks git `pre-` et `post-receive` alliés aux options de transformation de paquets en ligne de commandes de GNU Guix rendent triviale la création d'un système d'intégration continue-livraison continue :

- construire le paquet avec les nouveaux commits permet de vérifier que les tests passent avant d'accepter les commits ;
- le commit peut ensuite être déployé sur les machines clientes, de manière atomique, loggée, avec retour en arrière possible.

Les tests ayant lieu dans un conteneur isolé, ils m'ont par exemple un jour permis de découvrir un bug lié aux fuseaux horaires que les machines de développement, de test et de production avaient laissé filer, étant toutes réglées sur le même fuseau.

12. FUTUR PRIX TURING ?

GNU Guix règle de manière élégante et générique les problèmes fondamentaux de la confiance dans le code et de la préservation des artefacts de calcul. ▌

RÉFÉRENCES

[GUIX] **https://guix.gnu.org/**

[LEFT-PAD] **https://kodfabrik.com/journal/i-ve-just-liberated-my-modules**

[MERCY] **https://medium.com/commitlog/the-internet-is-at-the-mercy-of-a-handful-of-people-73fac4bc5068**

[HACK] **https://medium.com/hackernoon/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5**

[PRNG] **https://www.debian.org/security/2008/dsa-1571**

[KEN] **https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf**

[BOOTSTRAP] **https://guix.gnu.org/en/blog/2020/guix-further-reduces-bootstrap-seed-to-25/**

[TUHS] **https://minnie.tuhs.org/pipermail/tuhs/2020-November/thread.html#22431**

[NIX] Dolstra, Eelco, Eelco Visser, and Merijn de Jonge. "Imposing a memory management discipline on software deployment". Proceedings. 26th International Conference on Software Engineering. IEEE, 2004

[SH] **https://www.softwareheritage.org**