CENTOS: Utilisation de mock pour créer des fichiers RPM

Table of Contents

Mock est un outil qui permet de construire des RPMs de manière fiable et reproductible. Pour ce faire, il crée en fait un chroot dans lequel il va installer un système de base que l'on peut choisir dans une liste de système epel/red Hat/Mangeia fournit dans /etc/mock.

Installation et configuration

Installer EPEL

Pour installer mock, on doit d'abord installer le référentiel EPEL (Extra Packages for Enterprise Linux) sur le système.

Pour installer EPEL sur CentOS, lancer les commandes suivantes:

```
sudo yum install epel-release
```

Installer mock

```
yum install mock
```

Configurer mock

Tout utilisateur qui exécutera mock doit être ajouté au groupe mock. Vous pouvez le faire en lançant la commande usermod .

Par exemple, l'utilisateur joe peut être ajouté au groupe fictif de la manière suivante:Ajouter l'utilisateur avec lequel on veut construire les RPM au groupe mock, puis le déconnecter et reconnecter pour que les modifications prennent effet.

```
sudo usermod -a -G mock myusername
```

La configuration de mock se trouve dans le dossier /etc/mock. En fait dedans il y a trois fichiers qui nous intéressent, à savoir :

- **defaults.cfg :** déclaration d'options de configuration et des repos à partir desquels mock ira chercher ses fichiers. La syntaxe pour les options de configuration est config_opts['option'] = 'value'. Quant à la syntaxe des repos c'est la même que celle que vous pourriez trouver dans /etc/yum.repos.d
- **sites-defaults.cfg :** déclaration d'options plus globales. defaults.cfg sert pour définir les options pour une distribution cible donnée, alors que site-defaults.cfg s'applique à tout le monde.
- **logging.ini :** les options du loggueur, classique quoi.

Par défaut il est déconseillé de modifier les fichiers à cet endroit car ils sont notamment utilisés pour faire des paquets directement pour RedHat. Pour faire des fichiers custom, faites le à un autre endroit différent.

Si on est derrière un proxy, modifier **config_opts['http_proxy']** dans `/etc/mock/site-defaults.cfg`.

Utilisation de Mock

Mock accepte un certain nombre de commandes.

Mock accepte les commandes au format ancien (par exemple, “rebuild” ou “init” sans “-”), mais celles-ci sont obsolètes; utilisez des commandes avec un “-” en tête.

Utilisation basique

Créer un srpm en utilisant 'rpmbuild -bs'. Ensuite, aller dans le répertoire où le srpm a été créé.

```
mock -r <configfile> --rebuild package-1.2-3.src.rpm
```

où <configfile> est le nom d'une configuration de /etc/mock/ (sans l'extension .cfg). Il n'est pas nécessaire de passer -r si on a défini un fichier de configuration par défaut via un lien symbolique, comme indiqué ci-dessus. –rebuild est l'option par défaut. Donc, ce qui suit fonctionnerait aussi bien

```
mock package-1.2-3.src.rpm
```

L'option –verbose est utile pour afficher davantage de sorties à des fins de débogage.

Pour utiliser une version antérieure à 0.8.8 ou sur un système doté de Python 2.4, et générer des packages i386 sous x86_64, passer la commande **setarch i386**:

```
setarch i386 mock -r <configfile> --rebuild package-1.2-3.src.rpm
```

Les nouvelles versions de **mock** n’ont plus besoin de la commande **setarch**, bien que cela ne fasse pas de mal si elle existe.

Construction de packages qui dépendent de packages non situés dans un référentiel

Pour construire un paquet P qui dépend d'un autre paquet Q et que Q ne se trouve pas dans un référentiel, on poeut toujours utiliser mock. Voici une façon de le faire.

Tout d’abord, initialiser le référentiel factice:

```
mock -r MOCK_CONFIG --init
```

Si le message d'erreur “Erreur: il n'y a pas de dépôt activé.”est retourné c’est parce

que'on est actuellement sur Fedora et que le faux système de contrôle à l'intérieur du chroot tente de télécharger un paquet en utilisant yum. Dans ce cas, il faut ajouter l'option "–dnf" pour le forcer à utiliser DNF.

```
mock -r MOCK_CONFIG --dnf --init
```

Un exemple de **MOCK_CONFIG** est "fedora-14-x86_64"; consulter le répertoire /etc/mock pour connaître les configurations fictives existantes.

Première Méthode : Installer les packages

Installer les packages nécessaires à la construction du programme (nommé dans BuildRequires) à partir des référentiels yum et/ou des RPM locaux. En pratique, cela peut nécessiter un certain nombre de paquest (pour les différentes dépendances); Pour lister plusieurs paquets sur la même ligne utiliser la commande :

```
mock -r MOCK_CONFIG --installe PACKAGE_NAME_OR_PATH_TO_RPM
```

Les versions actuelles de mock devraient permettre de compiler à partir du RPM source. L'option –no-clean est nécessaire car par défaut, le chroot est nettoyé avant la construction.

```
mock -r MOCK_CONFIG --no-clean /PATH/TO/SRPM
```

Deuxième méthode : Copier le RPM source

Copier le RPM source dans /tmp (pour contourner les contrôles qui détectent que les paquetages ne sont pas dans le référentiel):

```
mock -r MOCK_CONFIG --copyin /PATH/TO/SRPM /tmp
```

Se connecter dans le Shell de l'environnement fictif et effectuer la construction:

```
mock -r MOCK_CONFIG --shell
CD
rpmbuild --rebuild / tmp / SRPM_NAME
```

Optimisation

Accélération

Dans `/etc/mock/site-defaults.cfg` appliquer les options d'optimation suivantes :

- Désactiver le plugin d'état du paquet: `config_opts ['plugin_conf']`

['package_state_enable']=False
- Paralléliser les constructions: `config_opts ['macros'] ['% _ smp_mflags'] = "-j17"`
- Améliorer les occurrences de ccache entre les versions de paquet:
```
config_opts['files']['etc/profile.d/zz-local.sh'] = """
unset CCACHE_HASHDIR
"""
```
- Compresser ccache: `config_opts ['plugin_conf'] ['ccache_opts'] ['compress'] = True`
- Utiliser la compression lzo pour le cache racine:
```
config_opts ['plugin_conf'] ['root_cache_opts'] ['compress_program'] = "lzop"
config_opts ['plugin_conf'] ['root_cache_opts'] ['extension'] = ".lzo"
```

Dans les fichiers de configuration par repo

- Compresser (dé) compresser au sein de **rpmbuild**:

```
    config_opts ['chroot_setup_cmd'] = 'installer @ buildsys-build
/usr/bin/pigz /usr/ bin/lbzip2'
    config_opts ['macros'] ['% __ gzip'] = '/usr/bin/pigz'
    config_opts ['macros'] ['% __ bzip2'] = '/usr/bin/lbzip2'
```

Utilisation de scan-build pour les paquets non modifiés

Ajouter /usr/bin/scan-build à l'option chroot_setup_cmd des fichiers de configuration par chroot:

```
config_opts['chroot_setup_cmd'] = 'install @buildsys-build /usr/bin/scan-
build'
```

Configurer un alias de shell pour l'utiliser à la place de plain mock (tout sur une seule ligne):

```
alias mock-scan-build="mock --define '__scan_build /usr/bin/scan-build' --
define '_configure %__scan_build ./configure' --define '__cmake
%__scan_build %{_bindir}/cmake' --define '__make %__scan_build
%{_bindir}/make' --define '___build_template
#!%{___build_shell}\\"$'\n'"alias make=\"%__make\"
cmake=\"%__cmake\"\\"$'\n'"%{___build_pre}\\"$'\n'"%{nil}'"
```

Voir les rapports générés dans le /tmp du chroot

Mise en cache

Mock 0.8.x introduit trois types de caches différents: 1) cache racine, 2) cache yum et 3) ccache. Ces caches accélèrent grandement la simulation.

Cache racine

À partir de mock 0.5, mock peut automatiquement mettre en cache la construction standard de chaque environnement dans un fichier tar local (`/var/cache/mock/$ CONFIG/root_cache/*`). Il décompresse ensuite ce fichier tar pour renseigner la buildroot, plutôt que de télécharger les RPM buildroot à chaque fois. Après avoir décompressé la buildroot à partir du cache racine, Mock effectue une **yum update** pour s'assurer que la buildroot est à jour avant l'installation supplémentaire de BuildRequires pour une construction de package.

Le cache racine est activé par défaut dans les versions 0.8.x et supérieures. Les caches racine sont automatiquement supprimés et recréés tous les 15 jours pour éviter qu'ils ne deviennent trop périmés.

Il est possible de désactiver le cache racine.

Cache yum

Par défaut, yum stocke les RPM téléchargés dans un répertoire sous `/var/cache/yum`. La fonctionnalité de cache yum monte le chroot /var/cache/yum dans un répertoire commun situé en dehors de l'environnement chroot (tel que `/var/cache/mock/$CONFIG/yum_cache/`) où il peut être enregistré et réutilisé par les générations suivantes. . Cela garantit que yum n'a pas besoin de télécharger chaque nouveau RPM sur le réseau pour chaque construction. Cette fonction est activée par défaut.

Il est possible d'utiliser SQUID pour étendre les possibilités offertes par la mise en cache des packages

Pour choisir un seul miroir rapide que squid n'utilise pas un miroir aléatoire pour le téléchargement des packages modifier les fichiers de configuration dans `/etc/mock/*.cfg`

Ne pas utiliser pas le miroir mirrorservice.org; ils empêchent la mise en cache en aval via l'en-tête HTTP "Cache-Control: no-store".

Ccache

L'outil **ccache** est un utilitaire qui encapsule les appels à des compilateurs tels que "gcc" et met en cache la sortie. Lorsqu'il est appelé une seconde fois pour compiler le même programme (avec les mêmes arguments **cmdline**, **gcc-version** et fichiers d'en-tête), **ccache** récupère la version mise en cache plutôt que d'exécuter à nouveau le compilateur.

Utiliser mock sous SELinux

L'utilisation de **SELinux** avec des environnements chrootés tels que **mock** présente un certain nombre de problèmes, tels que la nécessité de répliquer tous les contextes de fichier par défaut avec le préfixe de répertoire **chroot**, qui est un cauchemar à administré. L'approche utilisée par **mock** est très simple, même si elle ne fonctionne qu'avec la stratégie targeted : précharger un factice **libselinux** qui fait que tous les processus enfants pensent que **SELinux** est désactivé. Comme **mock** s'exécute en tant que processus non confiné , cela fonctionne généralement bien.

Problèmes liés à la protection de la mémoire SELinux

Cependant, à partir de FC5, le domaine **unconfined_t SELinux** n'est plus totalement non confiné par **SELinux**; La protection de la mémoire est mise en œuvre pour protéger la plupart des processus utilisateur contre une variété d'exploits possibles. Cela pose au moins deux types de problèmes lorsqu'on utilise mock dans FC5:

- Lors de la création de packages pour des distributions héritées , le processus de construction peut impliquer le chargement d'anciennes bibliothèques partagées ne comportant pas de sections séparées pour, par exemple, le code exécutable et les données accessibles en écriture. Cela entraîne des refus **execmod** lorsqu'un processus exécuté dans le domaine **unconfined_t** tente de charger une telle bibliothèque, sauf si la bibliothèque est libellée **textrel_shlib_t**
- Lors de la création de packages utilisant mono (la même chose pourrait s'appliquer à Java), le processus mono devrait normalement s'exécuter dans son propre domaine moins contraint (**mono_t**); lors de l'exécution sous **mock**, mono s'exécute dans le domaine **unconfined_t** , ce qui peut entraîner des **execstack** et **execheap** .

Bien qu'il soit possible de désactiver ces contrôles à l'aide de booléens SELinux, ce n'est pas souhaitable pour tous les processus. Une autre solution serait de s'assurer que tous les fichiers sont correctement étiquetés dans le chroot et que les transitions de domaine SELinux se produisent en cas de besoin. Cependant, cela n'est pas possible car le processus factice libselinux que SELinux est désactivé.

Module de politique SELinux pour maquette

La solution suivante à ces problèmes crée une stratégie SELinux pour la simulation comportant deux aspects:

- Il s'exécute dans son propre domaine (mock*t) aussi libre que tout processus devant s'exécuter, ce qui signifie jusqu'à présent qu'il autorise execheap et execmod , comme le domaine mono*t .
- Il étiquette tous les fichiers installés sous /var/lib/mock (où mock configure ses environnements chroot) avec le type de contexte spécial mock*var*lib*t , pour lequel execmod est autorisé pour les processus du domaine mock*t .

Cette solution assouplit suffisamment la protection SELinux pour que la simulation puisse fonctionner, sans compromettre la protection offerte au domaine non restreint normal.

Pour installer le module de stratégie:

- Créer un répertoire pour stocker les fichiers de règles locaux, tels que `/root/selinux.local`
- Télécharger les fichiers: **PackageMaintainers_MockTricks_mock.if**, **PackageMaintainers_MockTricks_mock.fc** et **PackageMaintainers_MockTricks_mock.te** dans ce répertoire
- Générer et installer le module de stratégie comme suit

```
# make -f /usr/share/selinux/devel/Makefile
Compliling targeted mock module
/usr/bin/checkmodule:  loading policy configuration from tmp/mock.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing binary representation (version 5) to
tmp/mock.mod
Creating targeted mock.pp policy package
rm tmp/mock.mod.fc tmp/mock.mod
```

Les packages selinux-policy-devel et checkpolicy sont obligatoires.

Si le module de stratégie est chargé avant l'installation du package mock*var*lib*t , /var/lib/mock sera étiqueté comme mock*var*lib*t* et /usr/bin/mock sera étiqueté comme mock*exec*t lors de l'installation. Sinon, il est nécessaire d'utiliser restorecon pour corriger les contextes de fichiers:

```
# restorecon -R /var/lib/mock /usr/bin/mock
```

Utiliser mock comme un outil sandbox chroot

Mock peut être utilisé pour créer des chroots permettant de tester des objets, pas seulement de construire des packages. Voici un petit guide:

```
Créer un fichier de configuration qui pointe vers le ou les référents de
votre choix, où se trouvent vos packages de test.

mock -r <nom_configuration> --init
mock -r <nom_configuration> --install <vos paquets>
mock -r <nom_configuration> --shell
```

Utiliser mock pour “shell” permettra à mock de créer les points de montage dont on aura probablement besoin à l'intérieur du chroot.