Documentation CLIP

1101

Génération de paquetages

Version	Date	Auteur	Commentaires
1.5.2	26/08/2008	Vincent Strubel	Correction de typos.
1.5.1	30/07/2008	Vincent Strubel	Convention plus lisible pour les références.
1.5	12/06/2008	Vincent Strubel	Evolutions de la génération d'entrées *veriexec*.
1.4	11/03/2008	Vincent Strubel	Ajout de 3 : description de la chaîne de compilation *Gentoo Hardened*. Ajout de *rootdisk.eclass* en 4.2. Passage au formate OpenOffice. Toujours à jour pour *portage-2.1.2.2-r8* (branche SGDN).
1.3	20/11/2007	Vincent Strubel	Ajout de 2.4 : gestion des fichiers de configuration. A jour pour *portage-2.1.2.2-r8* (branche SGDN).
1.2	20/09/2007	Vincent Strubel	Ajout de 4.2 : description des *eclasses* spécifiques à CLIP. Toujours à jour pour *portage-2.1.2.2-r3*
1.1	24/06/2007	Vincent Strubel	Mise à jour pour *portage-2.1.2.2-r3* : droits sur les bibliothèques et collisions entre *maintainer-scripts*.
1.0	19/06/2007	Vincent Strubel	Version initiale, à jour pour *portage-2.1.2.2-r1, clip-deb-1.0.9, clip-build-1.0.10*.

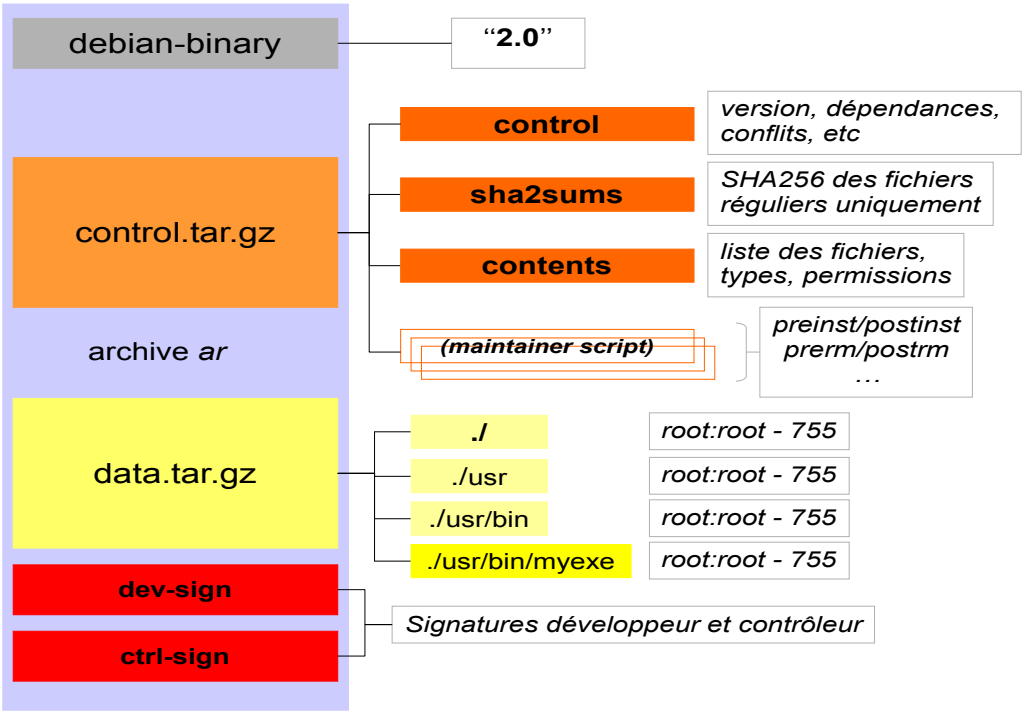
Table des matières

Introduction....4
1 Format de paquetage....5
2 Mise en oeuvre de portage....7
2.1 Principes de fonctionnement de portage....7
2.2 Modifications de portage....9
2.2.1 Ajout de FEATURES : noinclude, nostatlib....9
2.2.2 Préfixe de configuration ajustable : CPREFIX....9
2.2.3 Préfixe d'installation déporté : CLIP_VROOTS....11
2.2.4 Bibliothèques dynamiques exécutables....12
2.3 Génération de paquetages debian....13
2.3.1 Génération par portage....13
2.3.2 Script gencontrol.pl....15
2.3.3 Prise en compte de veriexec....18
2.4 Gestion des fichiers de configuration....19
3 Chaîne de compilation durcie....21
3.1 Propolice / SSP....21
3.2 Génération d'exécutables PIE....24
3.3 Durcissement de l'édition de liens dynamique....25
3.4 Génération automatique d'en-têtes PaX....26
4 Organisation des ebuilds CLIP....27
4.1 Arborescences Portage....27
4.2 Eclasses spécifiques à CLIP....29
5 Pilotage de portage : clip-build....33
Annexe A Références....36
Annexe B Liste des figures....37
Annexe C Liste des tableaux....37
Annexe D Liste des remarques....37

Introduction

La distribution *Gentoo* a été retenue comme point de départ pour la génération d'un système CLIP, autant du fait de sa mise en œuvre d'une chaîne de compilation « durcie » ([TOOLCHAIN]) que pour la maîtrise très fine qu'elle offre de la compilation et de l'installation de paquetages. Cependant, ce choix présente un inconvénient majeur vis-à-vis du format de paquetage binaire, qui doit être mis en œuvre pour distribuer des mises à jours aux postes clients. Bien que *Gentoo* supporte un format de paquetage binaire spécifique, celui-ci est peu adapté au contexte d'emploi de CLIP. En effet, l'utilisation d'un paquetage binaire *Gentoo* nécessite la présence sur le client d'un interpréteur *python*, de l'ensemble des modules et utilitaires *portage*, et de l'arborescence d'*ebuilds portage*, ce qui impose à la fois des transferts lourds pour la synchronisation périodique des arborescences, et un périmètre logiciel étendu. De ce fait, le choix s'est porté dans le cadre de CLIP sur un format de paquetage binaire issu d'une autre distribution Linux : le format *.deb* de *Debian* ([DEBPOLICY]), mais toujours généré à partir d'un *ebuild Gentoo* plutôt qu'à l'aide d'un paquetage source *Debian* classique. Le présent document détaille le format de paquetage binaire mis en œuvre dans CLIP, ainsi que les modifications de *portage* nécessaires à sa génération.

1 Format de paquetage



Les paquetages binaires utilisés dans CLIP sont, dans les grandes lignes, conformes au format *.deb* tel qu'il est défini par la *Debian Policy* (cf. [DEBOOTSTRAP]) : archive au format *ar*, contenant un fichier *debian-binary* (limité aux trois caractères ASCII "2.0") et deux archives *tar.gz*, *data.tar.gz* pour les fichiers à installer et *control.tar.gz* pour les méta-données du paquetage. En particulier, la compatibilité est suffisante pour permettre la mise en œuvre dans CLIP des outils de gestion de paquetage *Debian*, *dpkg* et *apt*, sans modification spécifique[1]. Cependant, les paquetages CLIP diffèrent quelque peu des paquetages purement *Debian* en ce qui concerne les méta-données embarquées dans l'archive *control.tar.gz*. Ces spécificités prennent la forme de champs supplémentaires dans le fichier « canonique » *control*, de fichiers supplémentaires obligatoires, *sha2sums* et *contents*, et d'une utilisation non-standard du fichier *conffiles,* qui recense dans CLIP les

[1] Les seuls patches appliqués à ces outils sont liés, d'une part, au basculement des répertoires d'état du gestionnaire de paquetage (en particulier, de la base de paquetages installés) de */var* vers */var/pkg*, et d'autre part à la suppression d'un test réalisé par *dpkg* préalablement à toute installation de paquetage, pour vérifier la présence dans le *PATH* d'un certain nombre d'utilitaires comme *ldconfig, start-stop-daemon* ou *update-alternatives* (spécifique à *Debian*), utilitaires qui ne sont pas forcément présents dans la vue UPDATE de CLIP qui invoque *dpkg*.

fichiers d'un paquetage qui doivent être recopiés lors du basculement de partition accompagnant la mise à jour du cœur du système. Ces différents éléments sont décrits en détail dans [CLIP_DCS_13018].

Les fichiers de méta-données de *control.tar.gz* peuvent être accompagnés, selon les paquetages, d'un certain nombre de *maintainers scripts* respectant les noms « canoniques » *Debian* (*preinst, postinst, prerm, postrm, etc*…). Les spécificités de l'environnement logiciel de CLIP imposent à ces derniers quelques restrictions propres. En particulier, les utilitaires de base dans les vues UPDATE sont ceux fournis par *BusyBox* ([BUSYBOX]), qui ne supportent pas nécessairement toutes les options proposées par les versions GNU des mêmes utilitaires. Ainsi, le fonctionnement d'un tel script ne doit pas reposer sur l'hypothèse que */bin/sh* est un alias vers *bash*, et supporte les extensions syntaxiques de ce dernier (ni sur l'hypothèse – équivalente – que */bin/bash* existe). De même, ces scripts doivent se conformer au contrôle strict des droits en écriture et en exécution sur les montages au sein de CLIP (par exemple, il n'est pas possible de créer un script temporaire dans */tmp* et de l'exécuter directement, pas plus qu'il n'est possible d'écrire dans */root* ou */etc*). Cependant, ces limitations sont compensées par le fait que le recours à des *maintainers scripts* demeure rare au sein de CLIP. En particulier, l'une des principales causes d'emploi des scripts *postinst* et *postrm*, la mise à jour du cache de bibliothèques par un appel à *ldconfig*, n'a pas lieu d'être dans CLIP (cf. 2.2).

Enfin, les paquetages CLIP peuvent contenir deux fichiers supplémentaires, contenant respectivement les signatures d'un développeur *(dev-sign*, portant sur les fichiers *debian-binary, control.tar.gz* et *data.tar.gz)* et d'un développeur (*ctrl-sign*, portant sur les mêmes fichiers plus *dev-sign*). La position de ces fichiers en fin d'archive *ar* fait qu'ils sont ignorés par les outils standard *debian*, et peuvent être facilement ajoutés à un paquetage après sa création.

2 Mise en oeuvre de *portage*

2.1 Principes de fonctionnement de *portage*

Les paquetages binaires CLIP sont générés à partir de fichiers sources et de scripts de compilation *ebuilds* à l'aide des outils *portage* natifs de la distribution *Gentoo*. Ces outils ont eux aussi fait l'objet d'un certain nombre de modifications nécessaires à leur emploi dans CLIP. *Portage,* proche dans le principe des *ports* BSD*,* offre une arborescence de scripts *ebuilds*, permettant chacun d'automatiser le téléchargement, la configuration, la compilation et l'installation d'un paquetage à partir de ses sources. A la différence des *ports* BSD, traditionnellement basés sur des *makefiles*, *portage* s'appuie sur l'articulation d'une couche logicielle de bas niveau (commandes *ebuild*), écrite en langage *shell* (*bash*), qui permet la manipulation directe des *ebuilds* et la réalisation d'opérations élémentaires comme la compilation ou l'installation, et d'une couche de haut niveau (commandes *emerge*), écrite en *python* et chargée essentiellement de la gestion des dépendances et du pilotage de la couche basse. Les *ebuilds,* organisés de manière arborescente par catégorie puis nom de paquetage puis version, contiennent chacun des méta-données sur le paquetage qu'ils permettent d'installer (dépendances, licence, etc…), ainsi que les instructions permettant de télécharger, configurer, compiler et installer le paquetage. Ces dernières sont structurées selon une *API (Application Programming Interface)* spécifique en *bash* : un ensemble de fonctions prédéfinies est appelé par la couche *ebuild*, elle-même pilotée par la couche *emerge*. Pour chacune de ces fonctions (par exemple*, src_compile()*, ou *src_install()*), un *ebuild* peut choisir de redéfinir la fonction de manière spécifique, ou de laisser la commande *ebuild* utiliser la définition par défaut, ou encore « hériter » une définition propre à un ensemble de paquetages et factorisée hors des *ebuilds* dans une *eclass*. Pour mémoire, les étapes principales de l'installation d'un paquetage à partir d'un *ebuild* sont (voir aussi Figure 2) :

- *fetch* (pas de fonction spécifique associée) : téléchargement de *${SRC_URI}* dans *${DISTDIR}* si absent, vérification des empreintes définies dans les fichiers *digest* associés à l'*ebuild.*
- *unpack (src_unpack())* : expansion des archives sources *${A}* (téléchargées depuis *${SRC_URI}*) dans *${WORKDIR}*, applications des patches éventuels.
- *compile (src_compile())* : configuration et compilation des sources depuis le répertoire source principal *${S}* ; correspond typiquement à la séquence classique *./configure <options> && make*, invoquée à travers des « *wrappers » portage, econf* et *emake*.
- *test (src_test())* : étape facultative, permet d'appeler *make check* et *make test* après la compilation, pour des paquetages intégrant une suite de test (ex. : *glibc*)
- *install (src_install())* : installation dans une arborescence temporaire *${D}* (aussi *${IMAGE}*), typiquement à l'aide d'un appel à *make DESTDIR=${D} install* (lui aussi invoqué à travers *emake*), complété de l'installation de fichiers spécifiques.
- *install_qa_check()* : cet appel ne constitue pas une étape *ebuild* distincte, mais est appelée par *emerge* suite à *src_install()*. Il réalise un certain nombre de vérifications de qualité sur les fichiers créés suite à l'installation, principalement suivant des critères de sécurité. Il lève en particulier des alertes en cas de détection dans *${D}* de fichiers réguliers inscriptibles par tous

(*+t*), d'exécutables *setXid* inscriptibles par des utilisateurs non privilégiés ou liés en *BIND_LAZY*, ou encore d'exécutables contenant des relocalisations dans le texte *(TEXTREL)*, des piles exécutables (*PT_GNU_STACK*), ou des segments projetés à la fois en écriture et en exécution[2].

- *preinst (pkg_preinst())* : actions supplémentaires à réaliser sur l'arborescence temporaire installée dans *${D}*, par exemple suppression des pages de *man*.
- *merge* (pas de fonction associée dans l'API) : déplacement des fichiers installés depuis l'arborescence temporaire vers le système de fichier réel (par défaut, déplacement de *${D}* vers la racine, mais cette dernière peut être surchargée par la variable *${ROOT}*)
- *postinst* (*pkg_postinst()*) : actions supplémentaires à réaliser sur les fichiers après installation dans l'arborescence réelle, par exemple mise à jour de cache.

L'ensemble offre de très importantes possibilités de configuration, en particulier à travers un certain nombre de variables d'environnement, comme *USE* et *FEATURES*, dont une définition globale est donnée dans le fichier */etc/make.conf,* mais aussi grâce à la notion de « profil » *(profile)* de compilation. La variable *USE* contient une liste de drapeaux affectant chacun une option de configuration d'un ou plusieurs paquetages. Par exemple, la présence dans *USE* du drapeau *png* entraînera le passage d'une option *--enable-png* ou équivalente au script *configure* de tous les paquetages qui offrent un support optionnel du format *png,* en ajoutant *media-libs/libpng* aux dépendances de ces paquetages. En l'absence de ce drapeau, l'option inverse *--disable-png* sera passée à ces mêmes paquetages. La présence de drapeaux *USE* peut aussi entraîner l'application de patches spécifiques à certains paquetages, ou des différences dans la nature des fichiers de configuration installés par défaut. *FEATURES* offre sur le même principe un contrôle plus générique du fonctionnement de *portage*, permettant par exemple de supprimer l'installation des pages de manuel par un drapeau *noman*, tandis que les variables *CFLAGS/CXXFLAGS, LDFLAGS* et *ASFLAGS* contrôlent les options qui seront passées respectivement au compilateur, à l'éditeur de liens, et à l'assembleur, lors d'une compilation. Ce degré de contrôle fait que *Gentoo* est souvent considérée comme une « méta-distribution », permettant de générer une distribution Linux sur mesure. Un profil, comme par exemple *hardened/x86/2.6* (le seul utilisé pour CLIP à ce stade), offre un ensemble de valeurs par défaut pour l'ensemble de la configuration de *portage*, qui s'apparente ainsi plus à une distribution classique, mais offre encore de nombreuses possibilités de configuration à travers *make.conf* et la surcharge des fichiers du profil par une arborescence créée dans */etc/portage*.

Ces possibilités de configuration sont largement mises en œuvre dans CLIP, d'une part pour activer les différentes options de durcissement[3], et d'autre part afin de minimiser la base installée, aussi bien en termes de complexité (utilisation d'un USE réduit et adapté pour limiter les dépendances et désactiver les options inutiles des logiciels installés) qu'en termes de taille (configuration fine de *FEATURES* afin de ne pas installer de documentation).

[2] Il est rappelé que de tels exécutables seraient inutilisables sans mesures particulières sur un système mettant en œuvre les protections *PaX,* comme c'est le cas pour CLIP.

[3] Un certain nombre de drapeaux *USE*, en particulier *hardened*, viennent compléter le choix d'un compilateur durci par l'application de patches spécifiques à certains paquetages, par exemple le code d'initialisation *SSP* ajouté au *runtime C* fourni par la *glibc*

2.2 Modifications de *portage*

Cette simple configuration est complétée dans le cadre de CLIP par la modification de certains aspects du fonctionnement de portage. Outre la génération de paquetages *Debian*, qui fait l'objet du paragraphe suivant, ces modifications sont principalement l'ajout de nouvelles *FEATURES*, le support d'un préfixe de configuration spécifique, et d'un ou plusieurs préfixes d'installation déportés par rapport au préfixe de configuration.

2.2.1 Ajout de FEATURES : noinclude, nostatlib

Deux drapeaux spécifiques à CLIP sont ajoutés à la variable *FEATURES*. Le drapeau *noinclude* permet de supprimer l'installation des fichiers *headers* *.*h* (typiquement dans */usr/include*), utiles uniquement pour la compilation, et qui n'ont donc pas de raison d'être présents sur le client final. De même, le drapeau *nostatlib* permet de supprimer avant installation les bibliothèques statiques (**.a*) et les fichiers de configuration *libtool* (**.la*) associés. L'utilisation de ces deux *FEATURES* permet de se rapprocher du fonctionnement d'un système de gestion de paquetages binaires, dans lequel les fichiers utiles uniquement au développement sont généralement installés par un paquetage *–dev* distinct. On notera cependant que les fichiers de configuration *libtool* (**.la*) sont nécessaires dans certains cas particuliers au chargement de bibliothèques dynamiques[4]. Dans un tel cas, la variable d'environnement *NOSTATLIB_KEEPLA* peut être définie à une valeur non nulle, avec pour conséquence la suppression des seules bibliothèques statiques, et non des fichiers de configuration *libtool*,par la *FEATURE nostatlib*.

L'implémentation de ces nouvelles fonctionnalités fait qu'elles ne sont mises en œuvre que lorsque la variable d'environnement *ROOT* (qui désigne la racine du système de fichier dans lequel les paquetages sont installés) contient un chemin non nul et différent de « */* ». En se fixant pour convention de compiler tous les paquetages destinés au client CLIP avec un *ROOT* spécifique, même s'il ne s'agit pas du véritable chemin d'installation, on peut ainsi bénéficier de ces fonctionnalités sans affecter les paquetages qui pourraient être installés au passage sur la machine de compilation elle-même (avec un *ROOT* automatiquement ramené à *« / »*) afin de satisfaire les dépendances de compilation. L'application des options *nostatlib* ou *noinclude* à ces dépendances de compilation rendrait en effet le poste de développement inutilisable.

2.2.2 Préfixe de configuration ajustable : CPREFIX

Les paquetages secondaires (destinés à pouvoir être mis à jour sans interruption du service) des différents compartiments logiciels (socle, cages) de CLIP sont en général installés avec un préfixe */usr/local*[5], au lieu du préfixe */usr* généralement utilisé par les distributions Linux. *Portage* est modifié de manière à supporter ce préfixe alternatif, à travers une variable d'environnement *CPREFIX,* qui permet

[4] Ces cas particuliers concernent notamment les exécutables qui chargent en cours d'exécution des bibliothèques dynamiques à l'aide des primitives *libtool* (de tels exécutables sont typiquement liés à la bibliothèque *libltdl.so* plutôt qu'à *libdl.so*), ainsi que les différents composants modulaires *(kparts, kio_slaves*) de KDE.

[5] Et ce y compris en ce qui concerne les fichiers de configuration, qui sont installés dans */usr/local/etc* plutôt que dans */etc*. */var* et */tmp* sont en revanche partagés entre le cœur et les paquetages secondaires de chaque compartiment. Cette organisation des fichiers est ainsi similaire à celle retenue pour les *ports* BSD (équivalents aux paquetages secondaires CLIP) vis-à-vis du *World* (équivalent au cœur d'un compartiment), à ceci près que les paquetages X11 sont aussi installés dans */usr/local*, plutôt que dans */usr/X11R7* comme c'est le cas sur la plupart des BSD.

de passer un préfixe arbitraire à utiliser au lieu de */usr*. Les modifications correspondantes sont réalisées à deux niveaux dans *portage* :

- Une modification de la fonction *econf* (*ebuild.sh*), qui est utilisée par la grande majorité des *ebuilds* dans *src_compile()* pour appeler le script *configure* d'un paquetage source. Les paramètres *--prefix, --mandir, --infodir* et *--sysconfdir* passés par cette fonction au script *configure* sont établis de manière à prendre en compte le préfixe CPREFIX s'il est défini. En revanche, les paramètres de configuration normalement associés à */var*, comme *–localstatedir*, sont laissés tels quels. Cette modification entraîne automatiquement l'installation dans le bon préfixe de tous les fichiers normalement installés par *(e)make install* dans *src_install()*.
- Une modification des utilitaires spécifiques à *portage* que les *ebuilds* mettent en œuvre afin de manipuler l'arborescence d'installation temporaire *${D}*. Ces utilitaires, qui sont définis soit comme des fonctions *bash* dans *ebuild.sh* (*into(), exeinto(),…*), soit comme des exécutables (scripts *python* ou *bash*) dans */usr/lib/portage/bin* (*dobin, dosbin, dopamd, doinitd*, *newexe*, etc. , cf. [DEVREL] pour une liste complète) sont typiquement utilisés pour compléter par des fichiers spécifiques à *Gentoo* l'arborescence crée par un *make install*. Ils utilisent différentes variables d'environnement (*INSDESTTREE, EXEDESTREE, etc…* modifiables par certaines fonctions comme *insinto()*, *exeinto()*) afin de déterminer le répertoire d'installation courant. Ces variables sont modifiées de manière à prendre en compte *CPREFIX*, aussi bien à leur initialisation par *ebuild.sh* que dans leur gestion par les différents utilitaires. Une exception spécifique est maintenue pour les chemins commençant par */var*, qui ne sont jamais modifiés. Cette modification est transparente pour les *ebuilds*, ainsi un *ebuild* réalisant par exemple les appels :

  ```
  insinto /usr/share/toto && doins ''${FILESDIR}''/tata
  ```

  installera *tata* dans */usr/share/toto* si *CPREFIX* est vide ou égal à */usr*, mais dans */usr/local/share/toto* si *CPREFIX* vaut */usr/local*, ou encore */opt/share/toto* si *CPREFIX* vaut */opt*.

Ces modifications sont limitées au code de *portage* lui-même et se veulent transparentes pour les *ebuilds*, qui n'ont pour l'essentiel pas besoin d'être modifiés en conséquence. Certaines modifications spécifiques sont cependant nécessaires pour les *ebuilds* qui manipulent directement l'arborescence *${D}* à l'aide des utilitaires UNIX standards plutôt qu'en utilisant les outils *portage*, et pour ceux des paquetages dont les sources n'utilisent pas pour leur compilation les outils *GNU autotools* et qui ne sont donc pas installés par la séquence *econf, emake, emake install* habituelle dans les *ebuilds.*

On notera que l'affectation de */usr/local* à la variable *CPREFIX* modifie uniquement le chemin de configuration et d'installation des paquetages, mais n'affecte pas le chemin de recherche de bibliothèques, qui conserve la valeur par défaut */lib:/usr/lib*. Afin de permettre l'édition dynamique de liens avec des bibliothèques installées dans */usr/local/lib* sans faire appel à un cache *ldconfig*[6], il est par ailleurs nécessaire de configurer explicitement le *RPATH* inscrit à l'édition de liens dans les bibliothèques et exécutables correspondants. *Portage* permet de réaliser une telle configuration à travers la variable d'environnement *LDFLAGS*, à laquelle il convient d'ajouter – dans le cas d'un

[6] Un tel cache n'est pas mis en œuvre dans CLIP pour des raisons de sécurité, car il facilite les attaques par interposition de bibliothèques. En particulier, il n'est pas possible de se satisfaire de droits discrétionnaires réservant l'accès à ce fichier à *root*, dans la mesure où des processus peuvent s'exécuter dans CLIP sous la même identité (y compris l'identité *root*), mais avec des niveaux de privilèges hétérogènes.

préfixe */usr/local* – l'option *"-Wl,-rpath,/usr/local/lib"*.

Enfin, la prise en compte de *CPREFIX* est complétée par la suppression, à l'aide de la fonction *gentoo preinst_mask*, de certains répertoires spécifiques à *gentoo* dans *${CPREFIX}/etc*, lorsque *CPREFIX* est non-nul. Ces répertoires sont principalement ceux liés au démarrage : *init.d/* et *conf.d*/. Dans la mesure où la procédure de démarrage de CLIP ne fait intervenir que des paquetages primaires, les informations contenues dans de tels répertoires dans un paquetage secondaire (*CPREFIX* non nul) seront forcément ignorées par CLIP, et sont donc supprimées afin d'éviter toute confusion.

Remarque 1 : Gestion des préfixes du poste de compilation

Le support de CPREFIX tel qu'il est implémenté à ce stade prend mal en compte les éventuelles différences de préfixes entre le client final et le système Gentoo *sur lequel les paquetages sont générés. Ainsi, le passage d'un* CPREFIX=/usr/local *peut entraîner à la compilation la recherche dans* /usr/local/include *de fichiers d'en-tête qui sont en fait installés dans* /usr/include *sur la machine de compilation. Deux solutions « workaround » peuvent être mises en œuvre : soit utiliser exactement les mêmes préfixes sur le poste de développement et sur le client, soit utiliser uniquement le préfixe par défaut* /usr *sur le poste de développement, en créant les liens symboliques suivants :*

```
/usr/local/lib          =>    /usr/lib
/usr/local/include      =>    /usr/include
/usr/local/qt           =>    /usr/qt
/usr/local/kde          =>    /usr/kde
/usr/local/ccsd         =>    /usr/ccsd
```

2.2.3 Préfixe d'installation déporté : CLIP_VROOTS

Une dernière modification permet de mettre en œuvre des préfixes d'installation déportés par rapport au préfixe de configuration. En effet, le principe de « vues » mis en œuvre dans CLIP impose pour certains paquetages une installation (par une vue UPDATE) dans un préfixe différent de celui qui sera « vu » à l'exécution, depuis une autre vue ou sous-vue. Les paquetages *sys-apps/busybox-X* qui fournissent les utilitaires de base des cages AUDIT et ADMIN constituent un exemple de tels paquetages : les utilitaires sont exécutés depuis le chemin */bin* dans les vues USER et ADMIN, mais les fichiers correspondants doivent être installés par la vue UPDATE dans */mounts/audit_root/bin* ou */mounts/admin_root/bin*. De ce fait, l'arborescence du paquetage (telle quelle serait contenue par exemple dans l'archive *data.tar.gz* d'un paquetage *Debian*) doit inclure le préfixe */mounts/audit_root* ou */mounts/admin_root*. En revanche, ce préfixe ne doit pas être passé à un éventuel script *configure*, afin de ne pas perturber la mise en œuvre des exécutables contenus dans le paquetage, qui doivent par exemple chercher leurs fichiers de configuration dans */etc* et non dans */mounts/audit_root/etc*. Il est ainsi nécessaire de mettre en place un mécanisme similaire à celui supporté nativement par *portage* à travers la variable *${ROOT}*, mais qui intervienne lors de la phase d'*install* plutôt qu'ultérieurement au moment du *merge*[7].

Un autre besoin spécifique à CLIP est celui du multiplexage des préfixes d'installation, c'est-à-dire

[7] L'utilisation de *${ROOT}* n'est pas appropriée dans ce cas dans la mesure où elle est liée à la fonction *merge*, qui n'est pas appelée dans le cadre de la génération de paquetage *Debian* pour CLIP (cf. 2.3). Même dans le cadre d'une utilisation directe de *portage* (sans génération de paquetage *Debian*), une telle approche serait problématique car elle conduirait à la mise à jour de la base de paquetages installés dans *${ROOT}/var/db/pkg* plutôt que dans */var/db/pkg* lui-même.

de l'installation des mêmes fichiers dans plusieurs arborescences. Le cas typique est celui des sous-vues visionneuses, dont la duplication (une sous-vue par cage) fait que les mêmes fichiers (par exemple */bin/vncviewer*) sont installés dans les deux arborescences symétriques construites au-dessus respectivement de */viewers/rm_h* ou */viewers/rm_b*. Une approche possible serait de dupliquer les paquetages correspondants (*net-misc/tightvnc-rmh* et *net-misc/tightvnc-rmb* au lieu de *net-misc/tightvnc*, de manière similaire à ce qui est fait pour *sys-apps/busybox*), mais au prix d'une plus grande complexité de gestion (le travail de maintenance des paquetages doit être dupliqué) et d'installation (le mécanisme de dépendances est perturbé), sans avantage supplémentaire dans la mesure où les fichiers des deux paquetages seraient strictement les mêmes. Il est plus simple dans ce cas d'inclure les deux arborescences dans un même paquetage.

Portage est modifié dans CLIP de manière à prendre en compte ces deux objectifs à travers une même variable *CLIP_VROOTS*. Celle-ci peut contenir une liste de répertoires d'installation « déportés » par rapport au préfixe de configuration. Si cette variable est non vide lors de l'appel à *src_install()*, la phase d'installation est appelée autant de fois qu'il y a de répertoires *${rep}* dans *CLIP_VROOTS,* en initialisant à chaque appel *${D}* à *${IMAGE}/${rep}*[8]. Ainsi, l'installation des paquetages propres à la cage AUDIT du socle CLIP est réalisée avec *CLIP_VROOTS="/mounts/audit_root"*, et celle des visionneuses *VNC* des sous-vues de USER avec *CLIP_VROOTS="/viewers/rm_h /viewers/rm_b"*.

Remarque 2 : Utilisation de liens durs

La prise en compte de racines virtuelles multiples par CLIP_VROOTS *se traduit pour l'heure par l'installation de copies des mêmes fichiers dans les différentes racines, ce qui entraîne un gaspillage d'espace disque, et une augmentation des débits de téléchargement de paquetages. Une solution plus optimale consisterait dans ce cas à installer une seule fois chaque fichier dans la première arborescence, et à créer ensuite des liens durs vers les originaux dans les autres arborescences.*

2.2.4 Bibliothèques dynamiques exécutables

La gestion des projections mémoire est modifiée dans CLIP par le LSM CLIP ([CLIP_1201]), de telle sorte que la projection en mémoire exécutable[9] d'un fichier n'est plus possible qu'à condition de posséder un droit (discrétionnaire) en exécution sur ce fichier. Afin de permettre l'utilisation des bibliothèques dynamiques (*.so*) présentes dans les paquetages CLIP, celles-ci sont systématiquement marquées avec des droits en exécution pour tous les utilisateurs. Cette opération est réalisée uniquement lors de la génération de paquetages binaires CLIP, par la fonction *clip_do_lib()*, appelée par *dyn_deb()* (cf. 2.3.1), qui réalise une recherche des bibliothèques dynamiques par application d'une expression régulière aux noms de fichiers (afin de prendre en compte non seulement **.so*, mais aussi **.so.X.Y.Z*, etc.) présents dans l'arborescence d'installation temporaire de l'*ebuild*.

---

[8] Dans le contexte de *portage*, *${IMAGE}* représente la racine de l'arborescence d'installation temporaire, utilisée après installation par la phase de *merge* ou de génération de paquetage, tandis *${D}* correspond à la racine de l'arborescence d'installation telle qu'elle est perçue par la phase *install* (qui prend typiquement la forme d'un *make DESTDIR=${D} install*). Ces deux variables coïncident dans le fonctionnement normal de *portage*. La prise en compte de *CLIP_VROOTS* réalise un déport (éventuellement répété pour différentes racines virtuelles) de *${D}* par rapport à *${IMAGE}*.

[9] Aussi bien par *mmap(...PROT_EXEC...fichier)* que par *mprotect(...PROT_EXEC...)* sur une projection associée à un fichier.

2.3 Génération de paquetages *debian*

2.3.1 Génération par portage

Outre ces modifications génériques de *portage*, un mécanisme spécifique est mis en place afin de générer des paquetages *Debian* à partir des *ebuilds portage*. Celui-ci s'articule autour d'une nouvelle étape *ebuild deb*, insérée entre les étapes *install* et *preinst*. Cette étape appelle une fonction *dyn_deb()* (définie dans */usr/lib/portage/bin/misc-functions.sh*), qui lance une commande *dpkg-deb –b* (paquetage *app-arch/dpkg*) afin de générer un paquetage *Debian* à partir de l'arborescence temporaire *${IMAGE}*, au préalable peuplée des fichiers à installer par *src_install()* et des méta-données correspondantes (contenues selon les conventions *dpkg-deb* dans le répertoire */DEBIAN* de l'arborescence à « empaqueter ») par *dyn_deb()*.

Avant d'appeler *dpkg-deb*, *dyn_deb* réalise les opérations suivantes :

- Appel de *clip_do_libs* (cf. 2.3.3 et 2.2.4).
- Appel de *preinst_mask*(), fonction de *portage* normalement appelée à l'étape *pkg_preinst*[10] afin de supprimer les fichiers masqués par les options *FEATURES* (par exemple, fichiers de *man* si le drapeau *noman* est inclus dans *FEATURES*).
- Suppression temporaire des fichiers *.keep* que *portage* utilise pour signaler à *preinst_mask* qu'un répertoire par ailleurs vide ne doit pas être supprimé. La liste de ces fichiers est sauvegardée dans une variable locale.
- Copie dans *${IMAGE}/DEBIAN* des fichiers de *${FILESDIR*[11]*}/_debian/*, si un tel répertoire est présent. Cette copie permet de définir statiquement des *maintainers scripts* spécifiques à un paquetage. On notera cependant que la génération dynamique à l'aide du *hook pkg_predeb()* mentionné ci-dessous est souvent mieux adaptée, dans la mesure où elle permet de prendre en compte la valeur spécifique lors de la compilation de certaines variables, comme *CPREFIX* ou *CLIP_VROOTS*, susceptibles de conditionner la forme des *maintainer scripts*, ou de réaliser des mesures (empreintes cryptographiques par exemple) sur les fichiers installés dans *${IMAGE}*. Par ailleurs, la copie de ces fichiers est réalisée de manière à éviter d'écraser des *maintainer scripts* déjà présent dans l'arborescence temporaire d'installation (par exemple créés directement par les *Makefiles* du paquetage). Dans un tel cas, le contenu de *${FILESDIR}/_debian/* est simplement concaténé aux scripts préexistants.
- Traitement des fichiers de configuration par *clip_conf_files()* (cf. 2.4).
- Copie du *ChangeLog* et de l'éventuel *ClipChangeLog*[12]
- Appel de la fonction *pkg_predeb()* si l'*ebuild* (ou une *eclass* qu'il hérite) définit une telle

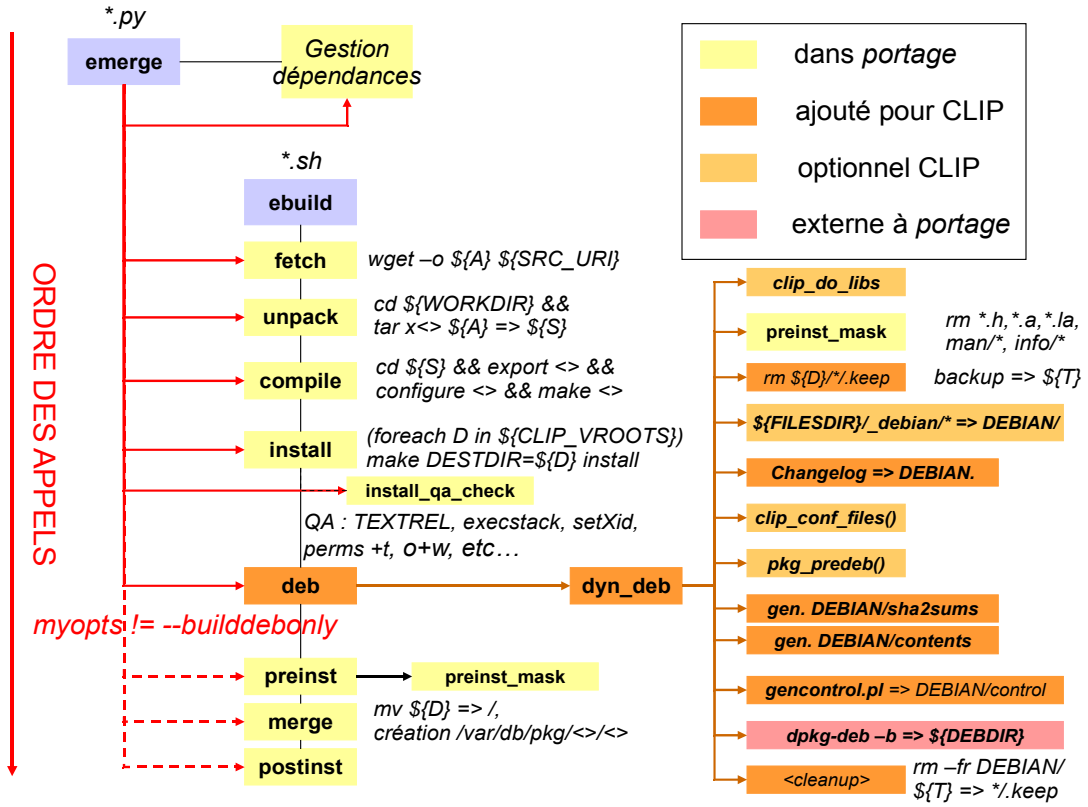
[10] Cette fonction est donc rappelée, inutilement mais sans impact néfaste, lors de la phase de *preinst* dans le cas où la génération de paquetage est suivie d'un *merge* sur le poste de compilation (option *–builddeb* plutôt que *–builddebonly*).

[11] *${FILESDIR}* représente le sous-répertoire *files/* du répertoire *${PORTDIR}/${CATEGORY}/${PN}* où est stocké l'*ebuild* lui même.

[12] Pour un *ebuild Gentoo* utilisé sans modification, ou un *ebuild* spécifique à CLIP, seul le fichier *ChangeLog* est présent. Dans le cas d'un *ebuild Gentoo* modifié pour CLIP, le fichier *ChangeLog* trace l'historique des modifications *gentoo*, tandis qu'un deuxième fichier, *ClipChangeLog*, est dédié à l'historique des modifications CLIP.

fonction. Ce *hook* facultatif est ajouté à l'*API ebuild*, de manière à permettre à un *ebuild* d'opérer un traitement spécifique à la génération de paquetages *.deb*, par exemple génération dynamique d'un script de post-installation dans *${IMAGE}/DEBIAN/postinst*. On notera que la résistance de ce mécanisme aux collisions (avec des *maintainer scripts* statiques, installés par la méthode ci-dessus ou par le paquetage source lui-même) repose sur l'utilisation de la fonction *init_maintainer* définie dans *deb.eclass* (cf. 4.2)

- Génération de *${IMAGE}/DEBIAN/sha2sums* en calculant les empreintes SHA256 de tous les fichiers réguliers de *${IMAGE}*, sous-répertoire *DEBIAN/* exclu.
- Génération similaire de *${IMAGE}/DEBIAN/contents*
- Appel d'un script *perl*, *gencontrol.pl* (installé par le paquetage *clip-dev/clip-deb*), afin de générer le fichier de méta-données principal, *${IMAGE}/DEBIAN/control*, à partir des méta-données du paquetage *Gentoo*, et de l'environnement de *portage*. Le script *gencontrol.pl* est décrit plus en détail en 2.3.2.



Après appel de *dpkg-deb* et stockage du paquetage *Debian* dans *${DEBDIR}*, l'arborescence

temporaire *${IMAGE}* est « remise en état » par suppression du répertoire DEBIAN et rétablissement des fichiers *.keep* sauvegardés dans une variable locale, afin de ne pas perturber un éventuel *merge*. L'ensemble des opérations est résumé dans la Figure 2.

Ces modifications de la couche bas-niveau (*ebuild*) de *portage* sont rendues accessibles à la couche de haut-niveau par l'ajout de deux options, *--builddeb* et *--builddebonly*, à la commande *emerge.* De manière similaire aux options natives *--buildpkg* et *--buildpkgonly*, la différence entre ces deux options consiste en la réalisation (*--builddeb*) ou non (*--builddebonly*) de la phase de *merge* après création du paquetage *Debian*.

2.3.2 Script gencontrol.pl

Le script *perl gencontrol.pl* est appelé par *portage* après la compilation (*src_compile()*) d'un *ebuild* et son installation dans un répertoire temporaire (*src_install()*). Il génère un fichier *control* à inclure dans les méta-données d'un paquetage *Debian* correspondant à cet *ebuild*, en exploitant les méta-données *portage* présentes dans le répertoire *${PORTAGE_BUILDDIR}/build-info*, complétées par des informations passées par la ligne de commande du script ou par son environnement. Outre l'agrégation de ces différentes informations et la génération du fichier *control* proprement dit, le principal rôle de *gencontrol.pl* est d'assurer la conversion du nom, de la version et des dépendances du paquetage depuis les conventions *Gentoo* vers les conventions *Debian.*

Conversion de nom et de version

Une version de paquetage *Gentoo* (*${PV}*) prend la forme suivante :

```
<version>{_<suffixe>{#}}{-r#}
```

(où # dénote un nombre) tandis qu'une version *Debian* prend la forme :

```
[<époque>:]<révision_principale>[-<révision_debian>]
```

La conversion de numéro de version échoue lorsque *suffixe* vaut *_alpha*, *_beta*, *_pre* ou *_rc*[13]. Dans les autres cas, la conversion réalisée est la suivante :

```
<époque> := (vide)
<révision_principale> := <version>{.<suffixe>{#}}
<révision_debian> := {-r#}
```

Par ailleurs, le nom de paquetage *Debian* correspond à celui du paquetage *Gentoo*, dans lequel les tirets bas **'_'**, interdits dans les noms de paquetages *Debian*, sont remplacés par des tirets hauts **'-'**. Le noms de paquetage et le numéro de version *gentoo* sont automatiquement récupérés dans les *build-info* par *gencontrol.pl,* de même que l'architecture de machine cible.

[13] C'est-à-dire toutes les valeurs autorisées de *suffixe* signifiant une version antérieure à la version stable *version*. Le rejet de telles versions n'est a priori pas gênant pour CLIP, dans la mesure où il n'est pas prévu de déployer des paquetages en version instable. La seule valeur de *suffixe* qui est autorisée à la fois par les conventions *Gentoo* et par *gencontrol.pl* est '*p*', qui dénote une version postérieure à la version stable principale. Dans ce cas, les conventions de conversion assure que la version *Debian* est bien postérieure à la version stable.

Conversion de dépendances

La génération des dépendances *Debian* se fait en plusieurs phases. Dans un premier temps, les dépendances *Gentoo* sont récupérées dans le fichier *RDEPEND* des *build-info*. Les dépendances ainsi obtenues sont alors encore exprimées selon les conventions *Gentoo*. En particulier, elles comportent des éléments dynamique, conditionnés par le contenu de la variable *USE*, et des dépendances « négatives », de la forme *!<nom de paquetage>*, qui constituent en fait des « conflits » au sens *Debian*. La transformation passe ensuite par les étapes suivantes :

- Les éléments conditionnels sont dans un premier temps réduits par comparaison avec le contenu de la variable *USE* lors de la compilation (qui est obtenu par lecture du fichier *USE* des *build-info*). Les composants des dépendances conditionnelles sont soit conservés sans condition, soit éliminés, selon qu'ils correspondent ou non à une condition *USE* satisfaite à la compilation.
- Chaque atome de dépendance est ensuite individuellement converti au format *Debian*. Le nom et la version de l'atome sont transformés selon les règles évoqués plus haut[14], la catégorie est éliminée, et la contrainte éventuelle est transformée en la contrainte *Debian* la plus proche[15]. Ces différents constituants sont ensuite réassemblés en un atome de dépendance *Debian*, selon le schéma suivant :

```
(Gentoo) [<contrainte>]<catégorie>/<nom>[-<version>]
                    =>
(Debian) <nom'> [( [<contrainte'>] <version'> )]
```

- Puis les agrégats de dépendances *Gentoo* (conditions « ou » ou « et » factorisées) sont transformés par associativité en sommes de facteurs. Par exemple :

```
(Gentoo) || ( atome1 atome2 atome3)
                    =>
(Debian) atome1'|atome2'|atome3'
```

ou encore :

```
(Gentoo) || ( atome1 ( atome2 atome3 ) )
                    =>
(Debian) atome1'|atome2', atome1'|atome3'
```

(On notera que dans cette dernière transformation, *atome1* est par ailleurs en général en conflit avec *atome2* et *atome3*.)

- Enfin, les éléments de la liste de dépendances ainsi normalisée sont répartis entre deux listes,

[14] A ceci prêt que les suffixes *alpha, beta, pre* et *rc* ne déclenchent pas une erreur, mais sont simplement éliminés. Les contraintes portant sur une version instable sont ainsi remplacées par des contraintes sur la version stable correspondante, qui est de toutes manière la seule à pouvoir être présente dans CLIP. On notera qu'une telle transformation est invalide dans le cas où la contrainte concerne un version *maximale* (dernière version instable satisfaisant une dépendance que la version stable ne satisfait plus). Cependant, ce cas est suffisamment rare dans *Gentoo* pour faire l'objet d'un traitement manuel (modification de la déclaration de dépendance dans l'*ebuild*).

[15] Les contraintes >= et <= sont laissées telles quelles, la contrainte = est supprimée (l'égalité de version est implicite par défaut dans l'expression de dépendances *Debian*), les > et < sont remplacés par >> et << respectivement. La contrainte *Gentoo* d'égalité partielle, exprimée à l'aide d'un ~, ou de manière équivalente d'une égalité à une version partielle suivie d'un *, est traduite de manière approximative par un >= *Debian*. Cette approximation est cohérente avec le fait que le mécanisme *Gentoo* des *SLOTs* (auquel sont associées ces contraintes partielles) demande dans CLIP une gestion manuelle spécifique.

une de dépendances « réelles », et une de conflits, selon que les atomes constituent ou non des dépendances négatives au sens *Gentoo*.

En cas de présence d'un fichier *PDEPEND* dans les *build-info*, exprimant des dépendances *a posteriori* qui n'ont pas d'équivalent *Debian*, le même traitement est appliqué à ce fichier, et les dépendances et conflits qui en résultent sont par approximation ajoutés aux dépendances et conflits déjà calculés à partir du *RDEPEND*. Ces deux listes peuvent ensuite être reportées dans les champs *Depends:* et *Conflicts:* du fichier *control* en cours de génération. Un traitement similaire, est appliqué aux dépendances virtuelles fournies par le paquetage et décrites par le fichier *PROVIDES*, si un tel fichier existe. Ces dépendances viennent peupler une troisième liste, qui est reportée dans le champ *Provides:* du fichier *control*. Enfin, les atomes apparaissant à la fois dans les listes *Conflicts:* et *Provides:* sont automatiquement ajoutés à une quatrième liste, qui est reportée dans le champ *Replaces:* du fichier *control*.

Informations complémentaires

En plus des informations automatiquement extraites des fichiers de méta-données *portage*, *gencontrol.pl* peut intégrer au fichier *control* qu'il produit des informations supplémentaires, qui lui sont passées soit par des options de la ligne de commande (passées par *portage* en fonction de l'*ebuild* ou de l'environnement), soit directement par des variables d'environnement. Les options de la ligne de commande sont de la forme -<*nom*> <*valeur*>. De manière classique pour *perl*, <*nom*> peut ici être remplacé par toute forme tronquée qui n'introduit pas d'ambigüité par rapport aux autres options possibles. Les options supportées sont répertoriées dans le Tableau 1. On notera que les champs correspondants ne figurent pas dans le fichier *control* généré si ces options ne sont pas explicitement définies.

Nom de l'option	Champ affecté dans le fichier *control*
–date	**Build-Date:**
–reldate	**Release-Date:**
–builder	**Built-By:**
–maintainer	**Maintainer:**
–size	**Installed-Size:**
–overlay	**Overlay:**
–distdir	**Distdir:**
–priority	**Priority:**
–urgency	**Urgency:**
–impact	**Impact:**

Tableau 1: Options de la ligne de commande de gencontrol.pl.

Le script est par ailleurs directement affecté par les variables d'environnement suivantes :

- **DEB_ESSENTIAL** : le champ **Essential:** est positionné à **yes** si cette variable est non-nulle.
- **DEB_DISTRIBUTION** : défini le contenu du champ **Distribution:**, qui est omis si la variable n'est pas définie.
- **DEB_NAME_SUFFIX** : le contenu de cette variable est ajouté au nom du paquetage (champ **Name:**). Il est ainsi possible de gérer des installations multiples faisant intervenir les *SLOTS Gentoo*. Par exemple, *gtk+-1.2** pourra être installé comme le paquetage *Debian gtk+1*, tandis que *gtk+-2** restera *gtk+*.

2.3.3 Prise en compte de *veriexec*

La gestion de *veriexec* ([CLIP_1201]) est intégrée aux paquetages *Debian* générés pour CLIP. Cette intégration se matérialise sous deux formes. D'une part, les fichiers de description d'entrées *veriexec*, utilisables par *vericti* (répertoires *vericti.d*), pour tous les fichiers binaires installés par un paquetage donné qui font l'objet d'une telle entrée dans la représentation, sont eux-mêmes inclus dans l'arborescence installée par ce paquetage (*data.tar.gz*). D'autre part, des *maintainers-scripts* inclus dans un tel paquetage assurent la mise à jour de la représentation *veriexec* de ces fichiers, par la suppression des anciennes entrées, si elles existent, et le chargement des nouvelles.

La procédure de génération de paquetage binaire fournit deux moyens complémentaires de produire un fichier de configuration *veriexec* dans un paquetage binaire. Les deux méthodes conduisent à l'ajout d'entrées dans un fichier de description d'empreintes *veriexec* unique (un par paquetage). Elles peuvent être utilisées aussi bien simultanément que séparément par n'importe que paquetage. Le fichier de description d'entrées de chaque paquetage est installé par le paquetage sous le chemin :

```
${CPREFIX}/etc/vericti.d/<nom du paquetage>
```

avec *CPREFIX* la valeur de la variable *CPREFIX* (cf. 2.2.2) telle qu'elle était définie lors de la génération du paquetage, et <*nom du paquetage*> le nom complet du paquetage, combinant le nom de base et un éventuel suffixe *DEB_NAME_SUFFIX* (cf. 2.3.2), mais pas le numéro de version.

La première méthode permet d'ajouter au fichier de descriptions les entrées associées à toutes les bibliothèques partagées installées par le paquetage. Elle est implémentée directement dans les fonctions de base de *portage* (*/usr/lib/portage/bin/misc-functions.sh,* fonction *clip_do_libs*, cf. 2.2.4 et Figure 2), et activée dès lors que la variable d'environnement *VERICTL_DOSHLIBS* est positionnée à une valeur non-nulle lors de l'appel *ebuild deb*. Dans ce cas, une entrée *veriexec* est créée pour chaque bibliothèque[16] présente dans l'arborescence *${IMAGE}*, après calcul de l'empreinte cryptographique correspondant en utilisant les fonctions de hachage CCSD fournies par l'utilitaire *ccsd-hash* (paquetage *clip-dev/ccsd-utils*). Chaque entrée est définie avec des masques de capacités nuls et le drapeau *VRX_FLAG_LIB*. L'identifiant de contexte des entrées est positionné à *${VERIEXEC_LIB_CTX}* si cette variable est définie, ou par défaut à –1, valeur qui est assimilée par *veriexec* au *xid* du processus réalisant l'opération de chargement sur */dev/veriexec*.

La création d'un tel fichier s'accompagne automatiquement de la création de deux *maintainer-scripts*, *postins* et *prerm,* afin de réaliser respectivement le chargement par *vericti* des entrées correspondantes après l'installation du paquetage, et leur suppression avant sa désinstallation. Ainsi,

[16] Les fichiers de l'arborescence correspondant à des bibliothèques sont déterminés par une expression régulière appliquée aux noms de fichiers.

lors d'une mise à jour du paquetage, les entrées associées aux anciennes versions de fichiers sont remplacées par les versions à jour de ces entrées. Lorsqu'un script *prerm* ou *postinst* est déjà présent dans le paquetage, les commandes correspondantes sont simplement ajoutées à ce dernier.

Par ailleurs, une *eclass portage* spécifique à CLIP fournit une deuxième méthode de génération de descriptions *veriexec*, permettant de produire de telles entrées pour des exécutables spécifiques, en leur attribuant cette fois des privilèges. Cette *eclass*, *verictl.eclass*, est décrite en 4.2.

2.4 Gestion des fichiers de configuration

Au sein d'un système CLIP, un certain nombre de fichiers de configuration, initialement installés par des paquetages, sont éditables par l'administrateur local. Il en va ainsi par exemple des fichiers de configuration réseau, pour lesquels le paquetage *app-clip/clip-net* (ou tout autre paquetage fournissant *virtual/clip-netbase*) fournit une version par défaut, mais qui ont vocation à être immédiatement personnalisés par l'administrateur local. Les fichiers localement modifiés ne doivent ensuite plus être écrasés par les mises à jour des paquetages qui les ont initialement installés. En revanche, il peut être nécessaire en de telles occasions de signaler à l'administrateur local qu'une mise à jour de ces fichiers est nécessaire au fonctionnement optimal de la nouvelle version du paquetage, par exemple si une adresse IP locale supplémentaire doit être définie.

Une modification de *portage* spécifique à CLIP permet de prendre en compte ces fichiers de configuration. Un *ebuild* peut définir une liste de fichiers de configuration qu'il installe, dans la variable *CLIP_CONF_FILES*. Chaque fichier contenu dans cette liste est décrit par son chemin complet, par rapport à la racine de l'arborescence temporaire d'installation *${D}*[17]. Pour chacun de ces fichiers, les opérations suivantes sont réalisées lors de la génération du paquetage *debian* (entre la copie des *maintainers scripts* statiques et l'appel à *pkg_predeb*, cf. Figure 2) :

- Renommage des fichiers de configuration, de manière à ne pas écraser les fichiers existant sur le système cible. La convention de nommage est la suivante :

```
/chemin/vers/fichier => /chemin/vers/.fichier.confnew
```

- Génération (ou ajout à des scripts préexistants, le cas échéant) de scripts *Debian postinst* et *prerm*, assurant les traitements décrits plus bas.

Le script *postinst* généré à cette occasion réalisera lors de l'installation les opérations suivantes :

- Si aucune version locale du fichier (avec son nom original, sans *.confnew*) n'existe sur le système, le fichier fournit par le paquetage est renommé de manière à devenir cette version locale. Ce cas est normalement limité à l'installation initiale, ou à l'ajout d'un fichier de configuration supplémentaire par le paquetage.
- Si une version locale du fichier existe, le fichier fourni par le paquetage est comparé avec la version locale (par appel à *cmp*). Si les deux fichiers sont identiques, la version fournie par le paquetage est supprimée. Dans le cas contraire, elle est laissée en place (avec son nom modifié, en *.confnew*).
- Le nom du fichier (sans modification) est dans tous les cas ajouté à une liste

[17] Le cas où la variable *CLIP_VROOTS* (cf.) spécifie un où plusieurs répertoires d'installation déportés est géré de manière transparente, il n'est pas nécessaire d'adapter *CLIP_CONF_FILES* dans ce cas.

/etc/admin/clip_install/conffiles.list.

Le script *prerm* assure quant à lui la suppression du nom du fichier de la liste */etc/admin/clip_install/conffiles.list*. On notera que lors de la suppression, les fichiers de configuration effectivement installés par le paquetage (c'est-à-dire ceux dont le nom contient *.confnew*) sont supprimés automatiquement s'ils sont encore présents. En revanche, les versions locales de ces fichiers (sans *.confnew*) ne sont jamais supprimées, même lorsqu'elles ont été créées par le paquetage et n'ont fait l'objet d'aucune modification locale.

Le renommage des fichiers de configuration, et le traitement réalisé lors de la post-installation garantissent que chaque fichier de configuration est bien créé lors de l'installation initiale, et jamais écrasé après celle-ci, et que des mises à jour éventuelles d'un fichier de configuration sont clairement signalées à l'administrateur local[18], qui peut ensuite réaliser manuellement la fusion de cette mise à jour et de sa version locale. Il est en revanche de la responsabilité des mainteneurs de paquetages de s'assurer que le système restera fonctionnel – au besoin avec des fonctionnalités dégradées – entre l'application de la mise à jour et la modification locale des fichiers de configuration.

Par ailleurs, la création et le maintien à jour d'une liste globale de fichiers de configuration susceptibles d'avoir été modifiés localement permet aux mécanismes de mise à jour et de sauvegarde / restauration d'un système CLIP de déterminer facilement une liste de fichiers qui doivent être recopiés lors d'un basculement entre deux installations CLIP.

[18] On peut remarquer cependant que la méthode simpliste de comparaison entre fichiers de configuration mise en œuvre à ce stade génère de nombreuses fausses-positives. En effet, un fichier *.confnew* est laissé en place et nécessite une vérification manuelle par l'administrateur dès lors que le fichier local est celui du paquetage diffèrent d'une quelconque manière. En particulier, lorsqu'un fichier a été modifié localement, toute mise à jour du paquetage ayant initialement créé ce fichier obligera l'administrateur à vérifier l'absence de différences autres que sa personnalisation entre les fichiers.

3 Chaîne de compilation durcie

Les paquetages déployés au sein d'un système CLIP sont pour la plupart[19] recompilés à partir de sources en C ou C++, en utilisant une chaîne de compilation durcie, issue de la branche *Hardened* de la distribution *Gentoo*. Cette chaîne est basée sur un compilateur *gcc-3.4* et des *binutils* (*2.16.1*), patchée pour intégrer automatiquement trois mécanismes de sécurité principaux, *Propolice/SSP,* la génération automatique d'exécutables *PIE*, et un durcissement de l'édition de liens dynamique.

3.1 Propolice / SSP

Le mécanisme *Propolice*, aussi connu sous le nom de *SSP* (*Stack Smashing Protector*), repose sur une modification de *gcc*, complétée par un support spécifique à l'exécution au sein de la *glibc,* afin d'assurer une protection générique contre les dépassements de tampon (« *buffer overflows* ») dans la pile. Cette protection n'interdit pas de tels dépassements, mais empêche dans plupart des cas leur exploitation à des fins malveillantes, tout en assurant leur détection. *Propolice* est nativement supportée par *gcc* à partir de la version *4.1*, et par la *glibc* en version *2.5* ou ultérieure. Cependant, la chaîne de compilation *Gentoo Hardened* mise en oeuvre dans CLIP utilise encore à ce stade l'implémentation originale de *Propolice*, reposant sur un patch appliqué à *gcc* en version *3.4*, qui diffère par certains aspects de la version finalement intégrée à *gcc-4.1*. De ce fait, un support spécifique de cette ancienne implémentation est aussi ajouté par un patch *Gentoo* à la *glibc*.

Propolice ajoute plusieurs options à la ligne de commande *gcc* :

- *-fstack-protector* : permet d'activer les protections *Propolice* pour toutes les fonctions du fichier compilé qui stockent sur leur pile une variable locale de type tableau (et donc plus particulièrement susceptibles de contenir des dépassements de tampons).
- *-fstack-protector-all* : permet d'activer les protections *Propolice* pour toutes les fonctions sans exception.
- *-fno-stack-protector* / *-fno-stack-protector-all* : permettent de désactiver ces protections. *-fno-stack-protector-all* annule aussi bien les effets de *-fstack-protector* que ceux de *fstack-protector-all*, tandis que *-fno-stack-protector* n'annule que ceux de *-fstack-protector* (une compilation avec les options *-fstack-protector-all -fno-stack-protector* génère toutes les protections).

La *spec gcc* par défaut est ajustée dans *Gentoo Hardened* pour ajouter par défaut les options suivantes :

- *-fstack-protector*, sauf si *-fno-stack-protector* ou *-nostdlib*[20] sont passées explicitement,
- *-fstack-protector-all*, sauf si *-fno-stack-protector* ou *-nostdlib* ou *-fno-stack-protector-all* sont passées explicitement, ou pour une compilation de la *glibc* (identifiée par la définition de la variable *D_LIBC*).

[19] Les principales exceptions concernent les paquetages qui n'installent aucun exécutable, ou ceux installant des exécutables « interprétés », *shell* ou *java* par exemple. S'y ajoute quelques paquetages installant des exécutables non-libres, disponibles uniquement sous forme binaire. Ces derniers exécutables ne bénéficient généralement de protections équivalentes à celles apportées par la compilation CLIP.

[20] Passée notamment pour une compilation noyau, qui ne bénéficie donc pas des protections *Propolice*.

Certains *ebuilds* désactivent explicitement la protection par défaut, soit en la restreignant aux fonctions définissant des tableaux locaux, soit en la supprimant entièrement. Par ailleurs, la *glibc gentoo* est patchée de manière à pouvoir être compilée avec *-fstack-protector*.

Les protections ajoutées par *Propolice* a un étage de pile donné reposent sur deux actions complémentaires : la réorganisation des variables locales, et la mise en oeuvre de « canaris » pour détecter les dépassements.

Remarque 3 : **Protection du noyau par *Propolice***

La chaîne de compilation Gentoo Hardened *désactive* Propolice *lors de la compilation du noyau Linux. En effet, le noyau Linux supporte, dans ses versions récentes (2.6.20 ou ultérieure) la mise en oeuvre de* Propolice *pour la protection de ses fonctions, mais uniquement sous la forme implémentée dans* gcc-4.1 *et ultérieur, et non dans le* gcc-3.4 *de* Gentoo Hardened. *Il serait cependant souhaitable à terme de pouvoir mettre en oeuvre ces protections dans les noyaux CLIP, soit par une modification du support noyau, soit par une mise à jour de la chaîne de compilation.*

Réorganisation des variables locales

Propolice réalise, dans la mesure du possible, une réorganisation des variables locales dans chaque étage de pile, de manière à placer les variables de type tableau aux adresses les plus-hautes, au dessus des autres variables locales. Cette modification découle de l'observation du fait que la plupart des dépassements de tampon se produisent dans le sens des adresses croissantes (dépassement vers le « haut » de la pile). Dans un tel cas de figure, le dépassement (qui se produit a priori dans l'une des variables de type tableau) ne pourra écraser que le contenu d'éventuelles autres variables de type tableau qui se situeraient au dessus de la variable dans laquelle se produit le dépassement, mais pas les variables locales scalaires. L'écrasement des autres données situées au-dessus des variables locales reste possible, mais est quant à lui détecté par la présence de canaris.

Vérification de « canaris »

Toutes les données situées au-dessus des variables locales d'un étage de pile donné peuvent être protégées par *Propolice* par l'ajout d'un « canari » juste au dessus des variables locales. Ce canari s'apparente à fausse variable locale entière, placée au sommet de la zone des variables locales de l'étage de pile, et renseignée à l'entrée de la fonction (par le prologue de cette fonction) à partir d'une valeur prédéfinie. Cette valeur est vérifiée par l'épilogue de la fonction avant de retourner à la fonction appelante. Si un dépassement de tampon s'est produit durant le traitement de la fonction, qui ait conduit à l'écrasement de données au dessus des variables locales de la fonction, et sous l'hypothèse que ce dépassement ait pris la forme d'une écriture continue (sans laisser de « trous » non modifiés, ce qui est généralement le cas), il aura conduit à une écriture dans la variable canari, en en modifiant généralement la valeur (sauf coïncidence, ou connaissance a priori de la valeur canari par un attaquant à l'origine du dépassement). Dans ce cas, un traitement d'erreur est déclenché, permettant l'interruption du programme fautif avant l'utilisation des données (autres que locales à la fonction fautive) modifiées par le dépassement (en particulier, l'adresse de retour de la fonction fautive, ou le pointeur de pile sauvegardé).

Dans l'implémentation de *Propolice* mise en oeuvre au sein de CLIP, la valeur canari est stockée

dans une variable globale *__guard*[21] (normalement accédée par l'intermédiaire de la *GOT*) de chaque processus, dont le contenu est initialisé au lancement du processus (lors de l'initialisation *libc*, donc avant *main()*) par l'éditeur de liens dynamique, */lib/ld-linux.so*. Ce dernier lit une valeur aléatoire de taille entière sur */dev/urandom,* puis en remplace les deux derniers octets par *255* et *'\n'* respectivement[22], ce qui laisse 16 bits d'aléa pour une architecture 32 bits (contre 48 pour une architecture qui stockerait les entiers sur 64 bits). Cette même valeur est par la suite utilisée comme canari par tous les étages de pile protégés. Un nouveau canari est régénéré lors de tout nouvel appel *exec()*. On notera en revanche qu'un *fork()* ne régénère pas le canari, ce qui peut permettre à un attaquant d'un démon « forkant » (qui créé un fils par *fork()* sans *exec()* pour traiter toute nouvelle connexion, par exemple *apache* ou *sshd*) de deviner la valeur du canari par des tentatives répétées sur des copies successives du processus maître. Une telle attaque sera néanmoins aisément détectable dans les journaux du système.

En cas d'échec de la vérification d'un canari, le prologue d'une fonction protégée par *Propolice* placera la valeur erronée lue à l'adresse du canari de son étage de pile, ainsi qu'un pointeur vers le nom de la fonction au sommet de la pile, avant d'appeler un symbole *stack_smashing_handler* fournit par la *glibc* . Ce gestionnaire d'erreur est écrit entièrement en assembleur *inline*, de manière à éviter tout appel de fonction qui dépilerait automatiquement des valeurs potentiellement corrompues de la pile. Il réalise les traitements suivant, dans cet ordre :

- Création d'une *socket UNIX*, et tentative de connexion à */dev/log* (par des appels système *inline,* donc sans appel de fonction)
- Écriture sur *STDERR*, ainsi que sur la *socket* précédemment créée si la connexion à */dev/log* a réussi, de deux messages de journalisation de la violation, incluant le nom de l'exécutable et celui de la fonction fautive.
- Suppression d'éventuels gestionnaires spécifiques du signal *SIGABRT*, puis envoi de *SIGABRT* au processus courant.
- Envoi de *SIGKILL* au processus courant (uniquement en cas de non terminaison par *SIGABRT*). On notera qu'aucun gestionnaire de *SIGKILL* ne peut avoir été défini par le programme.
- Appel à *exit()*, uniquement en cas de non terminaison par *SIGABRT* et *SIGKILL*.
- Entrée dans une boucle vide infinie si aucune des méthodes précédentes n'a pu terminer le processus.

Les messages signalant les violations *Propolice* sont, sous CLIP, collectés par le démon *syslog-ng* de la cage CLIP, et stockés dans un fichier dédié, */var/log/ssp.log*.

Remarque 4 : Canaris Propolice et gestion des exceptions

Les canaris Propolice *sont systématiquement vérifiés lors d'un retour normal d'une fonction à son appelant. En revanche, aucune vérification n'est réalisée lors de la remontée de la pile d'appels dans le cadre du traitement d'une exception, dans les langages qui les supportent (C++ par exemple). Il est ainsi théoriquement possible de modifier par un dépassement de tampon les étages de pile des fonctions appelantes, puis de faire exploiter*

[21] Dans la *glibc-2.6* utilisée par CLIP, l'éditeur de liens initialise aussi une valeur canari par *thread*, dans les segments *thread local storage.* La valeur affectée lors d'un *exec()* est la même que celle placée dans *__guard*. D'autres valeurs peuvent être tirées lors de la création de nouveaux *threads*. Ces valeurs canaris par *threads* sont celles mises en oeuvre par l'implémentation de *Propolice* intégrée à *gcc-4.1*. Elles ne sont en particulier pas utilisées dans CLIP à ce stade.

[22] Ces valeurs sont particulièrement intéressantes pour un canari, dans la mesure où elle ne sont que rarement disponibles pour un dépassement de tampon (qui se produit souvent lors de la manipulation de chaînes de caractères ASCII). La valeur nulle, correspondant à la terminaison de telles chaînes, serait aussi appropriée.

ces étages modifiés par le traitement d'exception, sans que la modification du canari ne soit jamais détectée. Une telle attaque pourrait par exemple conduire à une redirection du flux de contrôle vers un faux gestionnaire d'exception. Il serait nécessaire, à terme, de modifier dans CLIP les routines de gestion d'exceptions du runtime C++ *au moins, afin de leur faire effectuer les vérifications de canaris à chaque étage de pile traité.*

3.2 Génération d'exécutables PIE

Les exécutables *PIE* (*Position Independent Executable*) sont des exécutables relocalisables, selon le même principe que des bibliothèques dynamiques, ce qui leur permet d'être chargés en mémoire et exécutés à une adresse de base (adresse de départ de la section *.text*) arbitraire. La chaîne de compilation GNU supporte nativement dans ses versions récentes la génération d'exécutables PIE, activée par le passage d'une option *-fPIE* pour la compilation, et *-pie* pour l'édition de liens. Le passage de *-fPIE* entraîne la génération de code relocalisable, similaire à celui produit pour des bibliothèques à l'aide de *-fPIC*, à quelques optimisations près (certains symboles de l'exécutable sont appelés à l'aide d'*offsets* statiques plutôt que par l'intermédiaire de la *GOT/PLT*). *-pie* déclenche une édition de liens adaptée à la génération de *PIE*, en remplaçant les fichiers classiques de support d'exécution (*crtbegin.so, crt1.o* et *crtend.o* pour un source C) par leurs versions *PIE* (*crtbeginS.o, Scrt1.o* et *crtendS.o*), pour produire un fichier *ELF* de type *ET_DYN* (comme une bibliothèque dynamique) plutôt que *ET_EXEC*.

La chaîne de compilation *Gentoo Hardened* ne modifie pas ce support natif, mais adapte le fichier *spec gcc* utilisé par défaut de manière à :

- Passer par défaut l'option *-fPIE* aux commandes de compilation et d'édition de liens, pour toute compilation autre que celle du noyau (identifiée par la définition de la variable préprocesseur *__KERNEL__*), d'une bibliothèque (identifiée par l'option *-shared* ou les options *-fpic / -fPIC*) ou d'un exécutable statique (option *-static*)[23], et dès lors qu'aucune des options *-fno-PIC, -fno-pic, -nostdlib, -nostartfiles, -fno-PIE, -fno-pie, -nopie* n'est passée explicitement.
- Passer par défaut l'option *-pie* aux commandes d'édition, à l'exception des commandes associées à une bibliothèque (option *-shared*) ou a un exécutable statique (*-static*), et dès lors qu'aucune des options *-nostdlib* (ce qui interdit en particulier le passage de *-pie* pour le noyau), *-nostartfile, -fno-PIE* ou *-fno-pie* n'est passée explicitement.

Ainsi, à l'exception de certains *ebuilds* qui ajoutent explicitement *-fPIE* aux options de compilation, l'ensemble des exécutables de CLIP produit à partir de sources C ou C++ le sont sous forme relocalisable *PIE*. De tels exécutables bénéficient d'une protection accrue par la randomisation mémoire *PAX* (*PAX_RANDMMAP*, cf. [CLIP_1203]), qui rendra automatiquement aléatoire leur adresse de base de chargement en mémoire.

[23] On notera que les options *-shared* ou *-static* ne sont que rarement passées aux commandes de compilation, dans la mesure où elles n'ont de sens que lors de l'édition de liens. Dans un tel cas de figure, il est possible que *gcc* produise par défaut du code *PIE* inadapté à la génération d'un exécutable statique par exemple. Ces cas spécifiques sont généralement traités par l'*ebuild* associé, qui se charge de passer explicitement *-fno-PIE* aux commandes de compilation.

3.3 Durcissement de l'édition de liens dynamique

La chaîne de compilation *Gentoo Hardened* définit par ailleurs automatiquement deux options de l'éditeur de liens, qui permette de durcir l'image mémoire des processus en protégeant la plupart des méta-données *ELF* contre les écritures.

La première de ces options est *-z relro*, crée un segment *PT_GNU_RELRO* dans les fichiers *ELF* produits par l'éditeur de liens, permettant de décrire les sections qui, bien initialement accessibles en écriture, peuvent être configurées en lecture seule par l'éditeur de liens dynamique, après la phase de relocalisation initiale au chargement de l'exécutable. En effet, plusieurs sections doivent être accessibles en écriture pour cet éditeur de liens, afin de renseigner les champs déterminés lors de la relocalisation (typiquement, les différentes tables de pointeurs vers des symboles résolus dynamiquement : *GOT* (*Global Offset Table*), *PLT* (*Procedure Linkage Table*), tables de constructeurs et de destructeurs). En revanche, aucun accès en écriture à ces sections n'est plus nécessaire (sauf *lazy binding,* décrit plus bas) une fois la résolution de symboles achevée. L'option *-z relro* permet non seulement de décrire ces sections dans un segment spécifique du fichier *ELF*, mais assure aussi que l'alignement de ces sections leur permettra d'être chargée sur des pages mémoires dédiées, dont les permissions pourront être ajustées sans affecter les autres sections accessibles en écriture. Lors du chargement d'un exécutable, l'éditeur de liens dynamique de la *glibc* détecte la présence de ce segment *PT_GNU_RELRO*, et appelle *mprotect()* après la résolution de symboles afin de supprimer l'accès en écriture aux pages concernées. Cette mesure assure la protection contre les écritures malveillantes des tables de pointeurs correspondantes, et en particuliers les tables de constructeurs et de destructeurs, qui contiennent des pointeurs de fonction dont la modification peut permettre un attaquant d'appeler du code arbitraire. Les sections normalement incluses dans le segment *PT_GNU_RELRO* sont : *.ctors, .dtors, .jcr, .data.rel.ro, .dynamic*.

En revanche, l'option *-z relro* seule ne suffit pas à assurer la protection contre les écritures de la *GOT* et de la *PLT* d'un programme, dans la mesure où celles-ci sont normalement renseignées au fur et à mesure de l'exécution du programme. En effet, la technique dite du *lazy binding,* mise en oeuvre par défaut par l'éditeur de liens dynamique de la *glibc*, fait qu'un symbole n'est recherché, et son adresse placée dans la *GOT* ou la *PLT*, que lorsque le programme l'appelle effectivement pour la première fois. De ce fait, la *GOT* et la *PLT* doivent normalement rester inscriptibles tout au long de l'exécution du programme, ce qui les expose aussi à une éventuelle modification malicieuse – là encore typiquement dans le but d'exécuter du code arbitraire par modification d'un pointeur de fonction de la *PLT.* Cependant, une deuxième option de l'éditeur de liens *binutils*, *-z now*, permet de désactiver complètement le *lazy binding* pour un exécutable, et de forcer au contraire une résolution immédiate de tous les symboles (et un renseignement de l'intégralité de la *GOT* et de la *PLT*) au chargement de l'exécutable. L'ajout de *-z now* à la ligne de commande de *ld* entraîne le placement d'un drapeau *BIND_NOW* dans la section *.dynamic* du fichier *ELF* généré. A l'exécution, l'éditeur de liens dynamique détecte ce drapeau, et force une résolution immédiate des symboles. Lorsqu'un segment *PT_GNU_RELRO* est de plus détecté dans l'exécutable, l'éditeur de liens ajoute la *GOT* et la *PLT* (section *.got*) à la liste des sections à protéger contre les accès en écriture avant de passer la main au programme. On notera qu'une telle résolution immédiate des symboles peut engendrer un délai, potentiellement important, au lancement de l'exécutable.

La *spec* par défaut utilisée par *gcc* est modifiée par *Gentoo Hardened* de manière à ajouter automatiquement *-z relro* et *-z now* aux commandes d'édition de liens, sauf lorsque *-norelro* ou *-nonow*, respectivement, sont passées explicitement sur la ligne de commande.

3.4 Génération automatique d'en-têtes PaX

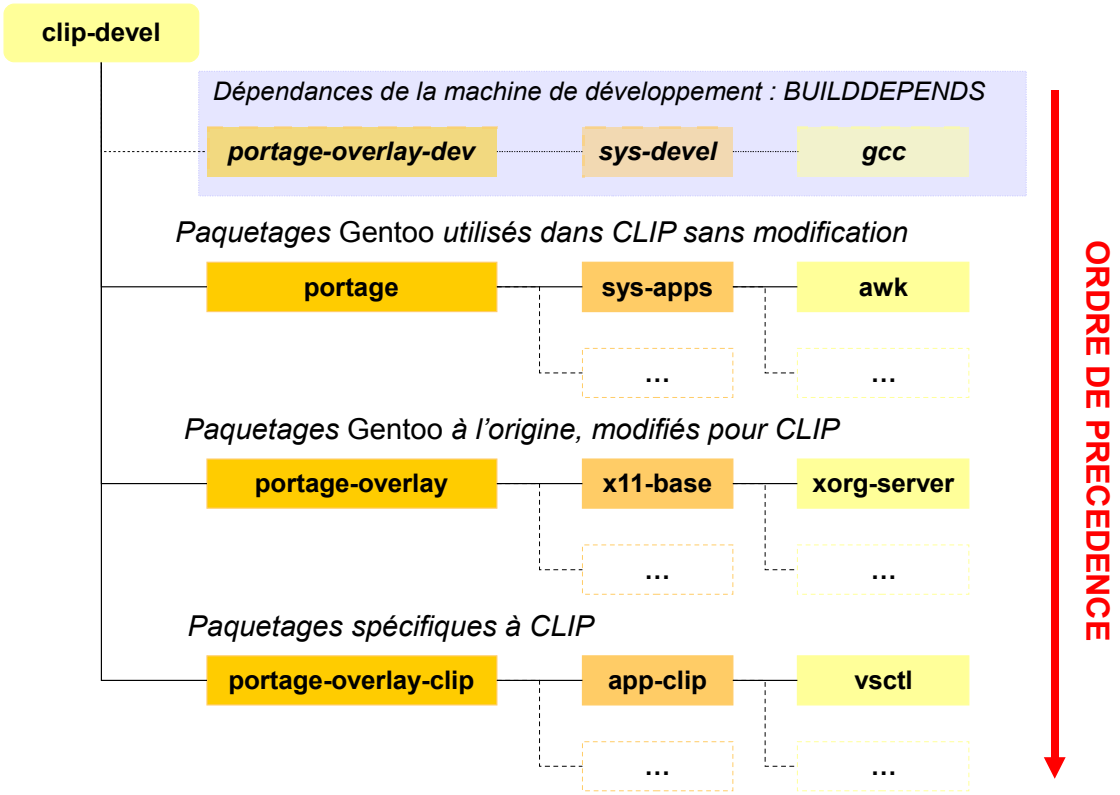
Enfin, *Gentoo Hardened* modifie l'éditeur de liens des *binutils* de manière à créer automatiquement un en-tête Pax *PT_PAX_FLAGS* dans chaque fichier *ELF* produit, de manière à supporter la modification des options PaX par exécutable (cf. [CLIP_1203]).

4 Organisation des ebuilds CLIP

4.1 Arborescences Portage

Les différents *ebuilds* utilisés pour la construction de CLIP sont organisés en utilisant une fonctionnalité supplémentaire de *portage* : la possibilité de définir des arborescences *overlay*. Un *overlay portage* est un répertoire contenant une arborescence *portage* partielle, similaire à l'arborescence principale généralement installée dans */usr/portage*. Il permet d'ajouter des *ebuilds* à l'arborescence *portage* officielle, ou de remplacer certains des *ebuilds* officiels par des versions modifiées. Plusieurs *overlays* peuvent être utilisés simultanément, avec un ordre de précédence bien défini afin de gérer les surcharges d'*ebuilds*. L'ensemble des *ebuilds* utilisés pour générer CLIP est regroupé dans quatre *overlays*, de telle sorte qu'aucun *ebuild* de l'arborescence principale ne soit utilisé directement. La séparation en quatre *overlays* plutôt qu'un seul permet de distinguer quatre catégories d'*ebuilds* :

- ***portage* :** (typiquement, *$HOME/portage* ou *$HOME/clip/portage* sur le poste de développement) contient les *ebuilds* et *eclasses* d'origine *Gentoo* utilisés dans le cadre de CLIP sans modification. La seule fonction de cet *overlay* est de fournir une extraction des *ebuilds* de l'arborescence *portage* officielle effectivement utilisés dans CLIP, et de se substituer à cette arborescence, afin de mieux suivre les *ebuilds* utilisés et d'éviter les importations non maîtrisées d'*ebuilds Gentoo*.
- ***portage-overlay* :** contient des *ebuilds* (et *eclasses*) d'origine *Gentoo*, mais modifiés pour l'utilisation dans CLIP, par ajout de patches ou de fichiers de configuration spécifiques, ou simplement afin de respecter entièrement les contraintes d'utilisation de *CPREFIX* (cf. 2.2.2). On notera que les *eclasses* définies dans cet *overlay* masquent automatiquement toutes *eclasses* de mêmes noms présentes dans l'*overlay* précédent.
- ***portage-overlay-clip* :** contient des *ebuilds* et *eclasses* spécifiques à CLIP, qu'il s'agisse de paquetages dont les sources sont disponibles publiquement mais qui ne sont pas intégrés à *Gentoo* (modules *pam*, outils *Debian apt,* etc.), ou bien de paquetages dont les sources ont été créées dans le cadre de CLIP (*jailmaster, vsctl*, etc.), ou encore de paquetages dérivés de paquetages *Gentoo* mais avec des noms modifiés (cas des *busybox-X*, dérivés de *busybox* mais avec des configurations spécifiques). Les *eclass* définies dans cet *overlay* sont utilisables dans tout paquetages des deux *overlays* précédents.
- ***portage-overlay-dev* :** contient des *ebuilds Gentoo* non modifiés, utilisés uniquement sur les postes de développement CLIP.



En plus des *ebuilds* / *eclasses*, trois autres types de fichiers sont inclus dans ces arborescences *overlay* :

- Un fichier de description de catégories, *portage-overlay-clip/profiles/categories*, qui définit les catégories d'*ebuilds* supplémentaires (en plus des catégories officielles de */usr/portage/profiles/categories*) utilisées dans le cadre de CLIP : *app-clip* (applications spécifiques à CLIP), *clip-dev* (outils de développement spécifiques, qui ne sont pas installés sur le client), *clip-layout* (paquetages créant les arborescences de répertoires et de *devices* des différentes vues) et *clip-libs* (bibliothèques)
- Deux fichiers de description de drapeaux USE, *portage-overlay/profiles/use.desc* et *portage-overlay/profiles/use.local.desc*, qui définissent les drapeaux USE spécifiques à CLIP. Le fichier *use.desc* décrit les drapeaux généraux (utilisés par un nombre indéterminé de paquetages), tandis que *use.local.desc* est réservé aux drapeaux locaux (utilisés uniquement par un ou quelques paquetages identifiés).
- Un fichier de descriptions de miroir tierce-partie, *portage-overlay-clip/profiles/thirdpartymirrors*, qui permet de définir un miroir '*clip*' de téléchargement de

sources, afin de rendre utilisable une construction de type

```
SRC_URI=mirror://clip/${P}.tar.bz2
```

valide pour exprimer la source d'un paquetage spécifique à CLIP. L'*URL* du miroir n'est pour l'heure pas utilisée directement.

4.2 Eclasses spécifiques à CLIP

Le répertoire *portage-overlay-clip/eclass* contient un certain nombre de classes *eclass* spécifiques à CLIP, qui peuvent être héritées par les *ebuilds*. Les fonctionnalités supplémentaires apportées par ces classes concernent notamment la génération de *maintainer-scripts* dans les paquetages *debian* produits par ces *ebuilds*.

deb.eclass

Cette classe très simple permet la création d'un en-tête de *maintainer-script* sans risque d'écraser un script préexistant. Elle définit une seule fonction, *init_maintainer*, typiquement appelée dans un *hook pkg_predeb*, de prototype :

```
init_maintainer <nom script>
```

Par exemple, l'appel à *init_maintainer postinst* créera un fichier *${IMAGE}/DEBIAN/postinst,* exécutable par *root*, et contenant l'en-tête suivant:

```
#!/bin/sh
set -e
```

sauf si un fichier portant ce nom existe déjà (par exemple, suite à sa copie depuis *${FILESDIR}/_DEBIAN/*). Le code du script peut ensuite dans tous les cas être ajouté à ce dernier par *echo* (ou *cat*) >> *${IMAGE}/DEBIAN/postinst*.

verictl.eclass

Cette *eclass* permet la création et la gestion d'entrées *veriexec* (cf. [CLIP_1201]) spécifiques pour des exécutables privilégiés. Elle définit une unique fonction *doverictld*, appelable dans un *hook pkg_predeb*, et qui accepte en argument le nom du fichier, les options et masques de capacités associés et la fonction de hachage à employer pour le calcul de l'empreinte cryptographique, selon le prototype :

```
doverictld <fichier> <flags> <cap_e> <cap_p> <cap_i> <algo> <privs>
```

avec :

- <*fichier*> le chemin du fichier pour lequel l'entrée doit être créée, relatif à la racine temporaire *${IMAGE}* (ce qui équivaut au chemin absolu du fichier tel qu'il sera installé par le paquetage). On notera que les cas où *CLIP_VROOTS* est défini (cf. 2.2.3) doivent être pris en compte explicitement : un appel à *doverictld* doit être réalisé pour chaque valeur de *CLIP_VROOTS*, en incluant cette valeur dans le chemin <*fichier*>.
- <*flags*> les options *veriexec* associées à cette entrée, sous la forme d'une concaténation de lettres-clés telle que supportées par *verictl(8)* (cf. [CLIP_1201]) .
- <*cap_e*>, <*cap_p*> et <*cap_i*> les masques de capacités effectif, permis et héritable associés à cette entrée, sous forme numérique.

- <*algo*> la fonction de hachage à utiliser, soit *md5*, *sha1*, *sha256* ou *ccsd*.
- <*privs*> les privilèges CLIP LSM associés à cette entrée, sous la forme d'une concaténation de lettres-clés telle que supportées par *verictl(8)* (cf. [CLIP_1201]). Cet argument peut être absent, auquel cas aucun privilège CLIP LSM n'est affecté à l'entrée.

L'entrée est créé dynamiquement en hachant le fichier présent dans *${IMAGE}*[24]. Le contexte *veriexec* associé à cette entrée peut être spécifié par la variable d'environnement *VERIEXEC_CTX*, ou laissé à sa valeur par défaut de –1. Chaque appel de cette fonction entraîne l'ajout d'une entrée unique dans le fichier de descriptions d'entrées du paquetage :

```
${CPREFIX}/etc/verictl.d/<nom du paquetage>
```

Comme pour les fichiers d'entrées *veriexec* associées aux bibliothèques (cf. 2.3.3), la création de ce fichier s'accompagne automatiquement (sauf lorsque ces commandes ont d'ores et déjà été ajoutées par *clip_do_libs()*) de l'ajout aux scripts *postinst* et *prerm* du paquetage des commandes *verictl* permettant respectivement de charger les entrées dans la base *verictl* à l'installation du paquetage, et de les supprimer à sa désinstallation. Si de tels scripts n'existent pas encore dans le paquetage, ils sont créés à cet occasion par un appel à la fonction *init_maintainer* décrite plus haut (*deb.eclass*).

screen-geom.eclass

Cette *eclass* facilite la prise en compte des différentes tailles d'écran envisageables sur les postes clients CLIP. En effet, les contraintes spécifiques à CLIP (interdiction des accès en écriture, limitation des logiciels installés au strict minimum) font qu'il n'est généralement pas possible d'adapter dynamiquement les fichiers de configuration des applications graphiques en fonction de la résolution de l'écran (comme cela est fait par exemple pour les fichiers de ressources X11 par un appel au préprocesseur C – qui n'est pas installé sur un poste CLIP). De ce fait, les fichiers de configuration doivent être ajustés à la résolution de l'écran lors de l'installation des paquetages auxquels ils appartiennent, par des scripts post-installation adaptés. A cette fin, les propriétés (dimensions X et Y uniquement à ce stade) de l'écran d'un poste donné sont normalement définies dans le fichier de configuration */etc/core/screen.geom*, qui est créé à l'installation (par un script *postinst* du paquetage *clip-layout/baselayout-clip*). La classe *screen-geom.eclass* fournit les fonctions détaillées ci-dessous afin de faciliter et d'abstraire dans les *ebuilds* les accès à ces informations.

```
gen_get_screen_geom <script>
```

La fonction *gen_get_screen_geom* ajoute au script <*script*> (fichier *${IMAGE}/DEBIAN/*<*script*>) la définition d'une fonction *get_screen_geom*, qui, appelée par ce script, définira dans l'environnement de ce dernier les variables suivantes :

- *CLIP_SCREEN_GEOM* : dimensions générales de l'écran, abscisses x ordonnées
- *CLIP_SCREEN_X* dimension en abscisses de l'écran
- *CLIP_SCREEN_Y* dimension en ordonnées de l'écran

[24] Il est de ce fait important de n'appeler cette fonction qu'après la génération de l'arborescence temporaire de fichiers installés, mais aussi après l'appel automatique à *strip* pour supprimer les symboles des exécutables, afin de calculer l'empreinte du fichier réellement installé. En particulier, le *hook pkg_predeb* satisfait ces conditions.

Ces informations sont lues dans le fichier */etc/core/screen.geom*, sauf lorsque la variable *CLIP_SCREEN_GEOM* est déjà définie lors de l'appel de la fonction, auquel cas cette définition est utilisée. Cette exception permet de définir explicitement les dimensions de l'écran lors de l'installation initiale du système.

Le script <*script*> doit avoir été initialisé préalablement à l'appel de cette fonction, par exemple par un appel à *init_maintainer* (*deb.eclass*).

```
gen_get_screen_matches  <var> <fonc> <script> <def> <val1-res1> <val2-res2> …
```

Cette fonction ajoute au script <*script*> la définition d'une fonction <*fonc*> qui écrit sur sa sortie standard <*res1*> si <*var*> vaut <*val1*>, <*res2*> si <*var*> vaut <*val2*>, etc. , et <*def*> par défaut si <*var*> ne vaut aucune des valeurs <*valX*>.

```
gen_get_x_matches <script> <def> <val1-res1>…

gen_get_y_matches <script> <def> <val1-res1>…
```

Fonctions équivalentes respectivement à :

gen_get_screen_matches CLIP_SCREEN_X get_x_matches <script> <def> <val1-res1> …

et

gen_get_screen_matches CLIP_SCREEN_Y get_y_matches <script> <def> <val1-res1> …

rootdisk.eclass

Cette *eclass* fournit une primitive facilitant la prise en compte des différents plan de partitionnement possibles pour un système CLIP. En effet, différents fichiers installés par des paquetages CLIP contiennent des définitions relatives aux *devices* de type disque montés dans CLIP (par exemple */etc/fstab* ou les fichiers de configuration du *bootloader*), et doivent être adaptés en post-installation afin de prendre en compte les spécificités de chaque poste CLIP (disque racine sur */dev/hda* ou */dev/sda* out */dev/md*, numéros de partition variables selon l'installation CLIP courante dans la mesure où deux systèmes CLIP sont installés sur le même disque, etc.). *rootdisk.eclass* fournit une fonction *gen_rootdisk()* qui, appelée dans la fonction *pkg_predeb()* d'un paquetage, ajoute au script *postinst* de ce paquetage (après l'avoir créé au besoin) le code permettant de définir un ensemble de variables d'environnement représentatives des partitions utilisées par le système CLIP courant :

- *ROOT_PARTITION* pour la partition montée à la racine, par exemple « */dev/sda5* »
- *HOME_PARTITION* pour la partition montée sur */home* dans le socle CLIP
- *LOG_PARTITION* pour la partition de stockage des journaux, montée sur */var/log*
- *SWAP_PARTITION* pour la partition dédiée au *swap*
- *BACKUP_DATA_PARTITION* et *BACKUP_APPS_PARTITION* pour les partitions dédiées au stockage des sauvegardes, respectivement des données utilisateur et du système
- *ROOT_DEVICE* pour le périphérique associé au disque racine, par exemple *« /dev/sda »*

- *ROOT_PARTNUM* le numéro de la partition montée à la racine, par exemple *« 5 »*

Ces différentes variables sont définies globalement dans le script *postinst*, à partir d'une lecture du fichier */etc/fstab* courant.

5 Pilotage de portage : clip-build

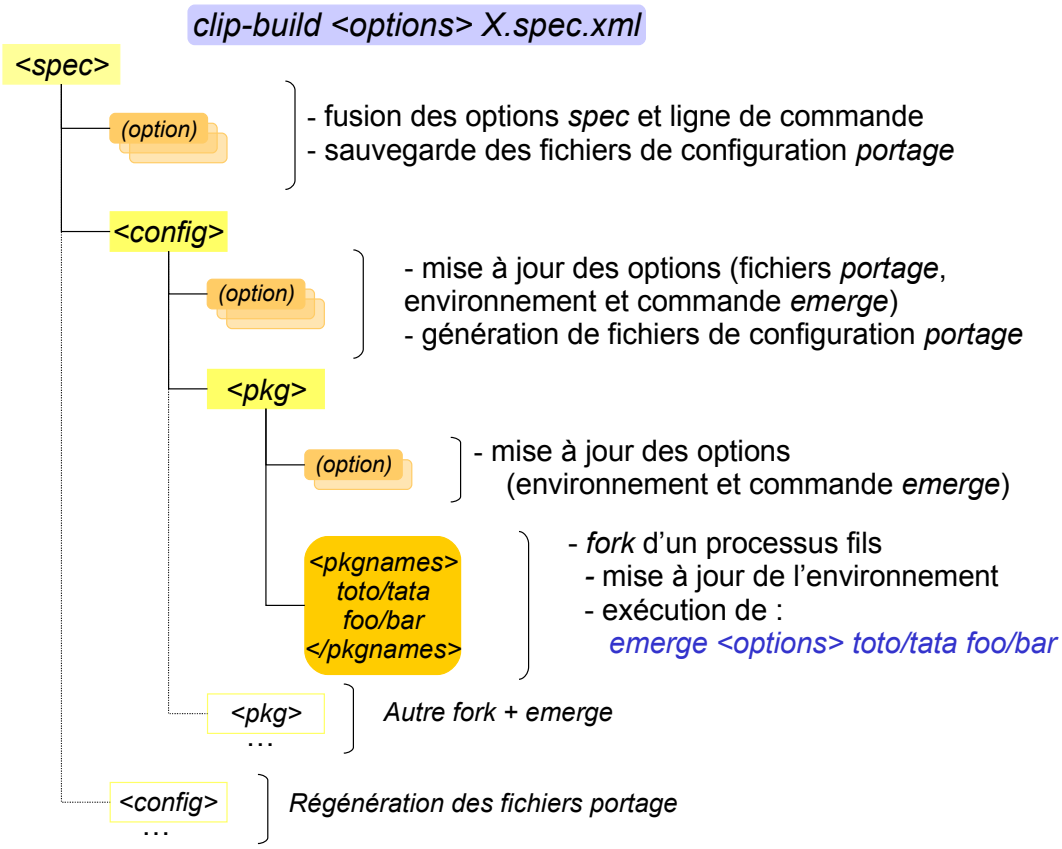
Afin de faciliter la génération maîtrisée de l'ensemble des paquetages d'un client CLIP, un outil spécifique a été développé : *clip-build* (paquetage *clip-dev/clip-build*). Celui-ci permet de configurer *portage* et de lancer une série de commandes *emerge* afin de générer l'ensemble des paquetages *.deb*[25] définis dans un fichier de configuration XML. Les fichiers de configuration *clip-build* donnent une représentation arborescente d'un ensemble de paquetages (typiquement tous les paquetages d'un compartiment, CLIP ou RM) : une <*spec*> est organisée en plusieurs <*config*>, qui contiennent elles-mêmes plusieurs <*pkg*>. Chaque élément <*pkg*> contient à son tour, dans un champ <*pkgnames*>, une liste de paquetages *Gentoo* (en général sous la forme <*catégorie*>/<*nom*>). Chacun des trois types de nœuds, <*spec*>, <*config*> et <*pkg*>, permet de définir un certain nombre d'options (options de configuration *portage*, variables d'environnement à définir, options à passer aux commandes *emerge*). Une même option peut-être définie « en cascade » à chacun des trois niveaux, ainsi éventuellement que sur la ligne de commande de *clip-build* lui-même, auquel cas des règles de fusion spécifiques à chaque type d'option sont appliquées (concaténation pour des variables comme *USE* ou *CFLAGS*, remplacement pour le profil *portage* ou les variables d'environnement). Pour chaque nœud <*pkg*>, *clip-build* crée un processus fils dont ils configure l'environnement en fonction des options courantes, avant de lui faire lancer une commande *emerge* avec comme argument tous les paquetages du nœud <*pkgnames*>.

Une limitation intrinsèque de ces possibilités de redéfinition d'options est liée au fait que certaines options de *portage* sont nécessairement associées à un fichier, et ne peuvent être passées à travers l'environnement ou les arguments d'une simple commande *emerge*. Par exemple, le profil *portage* est défini par un lien symbolique */etc/make.profile* pointant vers le sous-répertoire approprié de *${PORTDIR}/profiles*, tandis que le masquage/démasquage sélectif de paquetages de la branche instable *Gentoo*, ou encore l'association de paquetages à des paquetages virtuels, ne se font qu'en renseignant des fichiers spécifiques de */etc/portage*. Pour assurer la prise en compte de telles options, *clip-build* réalise à l'exécution une sauvegarde des fichiers de configuration *portage*[26], et régénère ceux-ci pour chaque nœud <*config*> qu'il lit, en fonction des options fusionnées de la ligne de commande, du nœud <*config*> courant, et du nœud <*spec*> parent. Les options correspondantes ne peuvent en revanche pas être définies au niveau d'un nœud <*pkg*>, pour lequel les fichiers de configuration *portage* ne peuvent pas être modifiés. A la terminaison de *clip-build* (normale ou suite à un signal), les fichiers de configuration *portage* initiaux sont rétablis.

Les options de la ligne de commande de *clip-build* (dont une liste exhaustive est donnée par *clip-build -help*) permettent en outre de définir les *overlays portage* à utiliser, le répertoire où stocker les paquetages, ainsi que de spécifier une <*config*> ou un <*pkg*> spécifiques (par leurs champs <*confname*> et <*pkgkey*>), voire un paquetage spécifique parmi la liste <*pkgnames*> d'un nœud <*pkg*>. On peut ainsi régénérer sélectivement certains paquetages de CLIP, tout en restant conforme à une configuration globale.

[25] *clip-build* peut aussi être utilisé pour installer directement (au sens d'un *merge Gentoo* plutôt que d'un *dpkg –i Debian*) les paquetages compilés, par exemple dans un *${ROOT}* correspondant au client. Cependant ce mode d'utilisation, antérieur à l'adoption d'un format de paquetage spécifique, est désormais peu mis en œuvre.

[26] */etc/make.conf*, */etc/make.profile* (lien symbolique) et le répertoire */etc/portage*.


clip-build <options> X.spec.xml
<spec>
(option)
- fusion des options spec et ligne de commande
- sauvegarde des fichiers de configuration portage
<config>
(option)
- mise à jour des options (fichiers portage, environnement et commande emerge)
- génération de fichiers de configuration portage
<pkg>
(option)
- mise à jour des options (environnement et commande emerge)
<pkgnames> toto/tata foo/bar </pkgnames>
- fork d'un processus fils
- mise à jour de l'environnement
- exécution de :
emerge <options> toto/tata foo/bar
<pkg>
…
Autre fork + emerge
<config>
…
Régénération des fichiers portage

```
<!ELEMENT spec (specname,
            use?,features?,cflags?,ldflags?,env?,profile?,
            config+)>
      <!ELEMENT config (id,confname,
                  rootdir?,pkgdir?,provided?,mask?,unmask?,
                  use?,features?,cflags?,ldflags?,profile?,env?,
                  pkg+)>
            <!ELEMENT provided CDATA #REQUIRED>
            <!ELEMENT virtuals CDATA #REQUIRED>
            <!ELEMENT mask CDATA #REQUIRED>
            <!ELEMENT unmask CDATA #REQUIRED>
            <!ELEMENT pkg (id,pkgkey,pkgnames,
                        options?,accept_keywords?,optoverride?,
                        use?,features?,cflags?,ldflags?,env?
                        )>
                  <!ELEMENT pkgkey CDATA #REQUIRED>
                  <!ELEMENT pkgnames CDATA #REQUIRED>
                  <!ELEMENT options CDATA #REQUIRED>
                  <!ELEMENT accept_keywords CDATA #REQUIRED>
                  <!ELEMENT optoverride CDATA #REQUIRED>
            <!ELEMENT id CDATA #REQUIRED>
            <!ELEMENT confname CDATA #REQUIRED>
            <!ELEMENT rootdir CDATA #REQUIRED>
            <!ELEMENT pkgdir CDATA #REQUIRED>
      <!ELEMENT specname CDATA #REQUIRED>
      <!ELEMENT use CDATA #REQUIRED>
      <!ELEMENT features CDATA #REQUIRED>
      <!ELEMENT cflags CDATA #REQUIRED>
      <!ELEMENT ldflags CDATA #REQUIRED>
      <!ELEMENT env CDATA #REQUIRED>
      <!ELEMENT profile CDATA #REQUIRED>
```

Définition du format XML attendu en entrée de clip-build.

Annexe A Références

[CLIP_1201] *Documentation CLIP – 1201 – Patch CLIP LSM*

[CLIP_1203] *Documentation CLIP – 1203 – Patch Grsecurity*

[CLIP_DCS_13018] *Format de Paquetage CLIP, CLIP-ST-13000-018-DCS Ed 0 Rev 2*

[PAX] *PaX, http://pax.grsecurity.net*

[DEVREL] *Gentoo Developer Handbook, http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml*

[TOOLCHAIN] *Gentoo Linux Documentation – The Gentoo Hardened Toolchain, http://www.gentoo.org/proj/en/hardened/hardened-toolchain.xml*

[DEBIAN] *Debian GNU/Linux, http://www.debian.org*

[DEBPOLICY] *Debian Policy Manual, http://www.debian.org/doc/debian-policy/*

[BUSYBOX] *BusyBox, http://www.busybox.net/about.html*

[DEBOOTSTRAP] *Debootstrap, http://packages.debian.org/stable/admin/debootstrap*

[SSP] *GCC extension for protecting applications from stack-smashing attacks, http://www.trl.ibm.com/projects/security/ssp*

Annexe B Liste des figures

Figure 1: Format de paquetage CLIP....5
Figure 2: Génération de paquetages .deb dans la séquence d'appels portage....14
Figure 3: Arborescence portage pour CLIP....28
Figure 4: Principe de fonctionnement de clip-build....34

Annexe C Liste des tableaux

Tableau 1: Options de la ligne de commande de gencontrol.pl....17

Annexe D Liste des remarques

Remarque 1 : Gestion des préfixes du poste de compilation....11
Remarque 2 : Utilisation de liens durs....12
Remarque 3 : Protection du noyau par Propolice....22
Remarque 4 : Canaris Propolice et gestion des exceptions....23