Éviter les failles de sécurité dès le développement d'une application 1ère Partie*

Cet article est le premier d'une série traitant des principales failles de sécurité susceptibles d'apparaître dans une application. Au cours des articles, nous présenterons les moyens de les éviter en modifiant quelque peu les habitudes de développement.

Introduction

Il s'écoule rarement plus d'une quinzaine de jours consécutifs sans qu'une application majeure fournie sur l'essentiel des distributions Linux ne révèle une faille de sécurité permettant, par exemple, à un utilisateur local de devenir *root*. Malgré la qualité indéniable de la plupart de ces logiciels, assurer la fiabilité d'un programme au niveau sécurité se révèle une tâche très compliquée : il ne faut pas qu'il permette à un utilisateur mal intentionné de tirer illégalement profit des ressources du système. La disponibilité des sources des applications est une excellente chose, que tous les programmeurs apprécient largement, mais le moindre défaut dans la cuirasse d'un logiciel se retrouve ainsi exposé aux yeux de tout le monde. De surcroît, la détection de tels défauts résulte souvent d'une démarche volontaire et les personnes s'y astreignant ne sont pas toutes bien intentionnées.

Du côté de l'administrateur du système, un travail quotidien doit être consacré à la lecture des listes de diffusion traitant des problèmes de sécurité, et à la mise à jour immédiate des paquetages incriminés. Du côté du programmeur en revanche, il est plus judicieux d'essayer d'anticiper les problèmes. Il est préférable d'éviter les failles éventuelles dès la conception ou la programmation du logiciel.Nous allons ici essayer de caractériser quelques comportements "classiquement" dangereux, et présenter des solutions pour réduire les risques tout en réalisant les mêmes fonctionnalités. Nous ne traiterons pas des problèmes de sécurité liés au réseau, car ils relèvent généralement d'erreurs de configuration (scripts CGI-bin dangereux, ...) ou de bogues situés dans le système lui-même et qui permettent des attaques de type DOS (*Denial Of Service*) afin de rendre la machine sourde à ses clients légitimes. Ces problèmes concernent l'administrateur du système ou les développeurs du noyau, mais aussi le programmeur applicatif s'il traite des données d'origine externe. Par exemple, **`pine`**, **`acroread`**, **`netscape`**, **`access`**,... dont certaines versions sous certaines conditions permettaient des accès distants ou des fuites d'information. La programmation *sécurisée* concerne finalement tout le monde.

Cette série d'articles présente des méthodes qui permettent de nuire à un système type Unix. Nous aurions pu uniquement les évoquer, ou en parler à mots couverts, mais nous préférons le faire ouvertement pour que chacun ait bien conscience des risques. Ainsi, lorsque vous corrigerez un programme ou que vous développerez le votre, vous serez à même d'éviter ou bien de corriger ces erreurs. Pour chaque faille présentée, nous adopterons une démarche identique. Nous commencerons par en détailler le fonctionnement. Ensuite, nous montrerons les précautions à prendre pour les éviter. A chaque fois, nous illustrerons nos propos avec des trous de sécurité présents dans des logiciels encore largement répandus.

Ce premier article présente les bases nécessaires à la compréhension des failles de sécurité, c'est à dire la notion de privilèges, et de bits *Set-UID* ou *Set-GID*. Nous nous consacrons ensuite à l'étude des failles les plus faciles à comprendre, fondées sur l'emploi de la fonction **`system()`**.

Nous utiliserons souvent de petits programmes en C pour illustrer nos propos. Toutefois, les démarches indiquées dans ces articles restent valables pour d'autres langages : perl, java, les shell scripts... Certaines failles dépendent d'un langage, mais ce n'est pas le cas de toutes, comme nous le verrons dès cet article avec **`system()`**.

---

* Cet article est paru dans le numéro 23 de *Linux Magazine France*, au mois de décembre 2000.

Privilèges

Les utilisateurs ne sont pas égaux sur un système Unix, pas plus d'ailleurs que ne le sont les applications. L'accès aux différents noeuds du système de fichiers - et de ce fait aux principaux périphériques de la machine - est soumis à un contrôle d'identité strict. De même, certains utilisateurs sont autorisés à entreprendre des opérations sensibles pour garantir le bon fonctionnement du système. L'identification des utilisateurs se fait par l'intermédiaire d'un numéro nommé UID (*User Identifier*). Par commodité, chaque numéro est associé à un nom d'utilisateur plus parlant, la correspondance s'établissant dans le fichier **`/etc/passwd`**.

L'utilisateur *root*, dont l'UID vaut 0 par définition, dispose de tous les droits sur le système. Il peut non seulement créer, modifier, ou supprimer n'importe quel noeud du système de fichiers, mais il peut aussi intervenir sur la configuration physique de la machine, en montant des partitions dans l'arborescence des fichiers, en activant des interfaces réseau et en modifiant leur configuration (adresse IP), ou simplement en invoquant des appels-système privilégiés comme **`mlock()`** qui agit sur la mémoire physique, ou **`sched_setscheduler()`** qui modifie le mécanisme d'ordonnancement. Nous étudierons dans un article ultérieur les capacités Posix.1e qui permettent de limiter un peu la toute puissance d'une application exécutée sous l'identité *root*, mais pour le moment nous considérerons que le super-utilisateur est tout-puissant sur la machine.

Les attaques que nous essaierons de prévenir dans nos articles sont internes, c'est à dire qu'un utilisateur dûment autorisé à se connecter sur la machine tente de s'approprier des privilèges ne lui appartenant pas. À l'opposé les attaques réseau sont généralement externes, en provenance de personnes cherchant à obtenir une connexion sur la station alors qu'elles n'y ont normalement pas d'accès légitime. S'approprier les privilèges d'un autre utilisateur signifie que l'opération à réaliser le sera sous couvert de son identité, de son UID, et non de la sienne propre. Naturellement, tout pirate tend à usurper l'identité *root*, mais d'autres comptes utilisateur s'avèrent également intéressants, soit parce qu'ils donnent accès à des informations système (*news*, *mail*, *lp*...) soit parce qu'ils permettent la lecture de données privées (courriers, fichiers personnels, etc) ou la dissimulation d'activités illégales comme les attaques dirigées vers d'autres sites.

Pour utiliser les privilèges réservés à un autre utilisateur, sans pour autant avoir la possibilité de se connecter directement sous son identité, il faut, au minimum, être en mesure de dialoguer avec une application s'exécutant sous l'UID de la victime. Lorsqu'une application - un processus - se déroule sous Linux, c'est avec une identité bien définie. Tout d'abord un programme est doté d'un attribut nommé RUID (*Real UID*) correspondant à l'identité de l'utilisateur l'ayant lancé. Cette donnée remplie par le noyau est normalement immuable. Un second attribut vient compléter cette information : le champ EUID (*Effective UID*) qui correspond à l'identité que le noyau prend effectivement en compte pour vérifier les autorisations d'accès pour les opérations nécessitant une identification (ouverture de fichier, appel-système réservé).

Pour qu'une application s'exécute avec un UID effectif (ses autorisations) différent de son UID réel (l'utilisateur qui l'a lancé), il faut fixer sur son fichier exécutable un bit de permission particulier nommé Set-UID. Ce bit se trouve dans l'attribut des permissions du fichier (comme les bits d'exécution, de lecture et d'écriture pour l'utilisateur, les membres du groupe ou le reste du monde), et correspond à la valeur octale 4000. Le bit Set-UID est représenté par un **`s`** lors de l'affichage des permissions avec la commande **`ls`** :

```
 ls -l /bin/su
-rwsr-xr-x  1 root root 14124 Aug 18 1999 /bin/su
```

La commande "**`find / -type f -perm +4000`**" fournit une liste des applications du système ayant leur bit Set-UID à 1. Lorsque le noyau lance une application dont le bit Set-UID du fichier exécutable vaut 1, il utilise l'identité du propriétaire du fichier comme EUID du processus. Le RUID, en revanche, ne varie pas, et correspond à la personne ayant lancé le programme. Dans le cas de **`/bin/su`** par exemple, chaque utilisateur dispose de cette commande, mais elle se déroule sous l'identité de son propriétaire (*root*), et possède donc tous les privilèges possibles sur le système. Inutile de préciser qu'il convient d'être particulièrement prudent lors de l'écriture d'un programme avec cet attribut.

Il existe symétriquement pour chaque processus un identificateur du groupe d'utilisateur effectif EGID, et un identificateur réel RGID. De même le bit Set-GID (2000 en octal) dans les permissions associées à un fichier exécutable demande au noyau de prendre comme EGID celui du groupe propriétaire du fichier, plutôt que le groupe de l'utilisateur ayant lancé le programme. Une combinaison curieuse, avec le bit Set-GID à 1 mais sans l'autorisation d'exécution pour le groupe, apparaît parfois. Il s'agit en réalité d'une convention n'ayant rien à voir avec les privilèges associés aux applications, mais indiquant seulement que le fichier est susceptible de faire l'objet de verrouillage strict avec la fonction **`fcntl(fd, F_SETLK, lock)`**. Une application tire rarement parti des possibilités offerte par le bit Set-GID, mais cela se produit parfois, par exemple certains jeux emploient ce mécanisme pour sauvegarder les meilleurs scores dans un répertoire système.

Types d'attaques et cibles potentielles

Il existe plusieurs types d'attaques à l'encontre de la sécurité d'un système. Les mécanismes que nous allons observer aujourd'hui reposent sur l'invocation d'une commande externe par une application, grâce à laquelle on

s'arrange pour démarrer un autre programme, souvent un shell, avec l'identité du propriétaire de l'application principale. Un deuxième type d'attaque, que nous étudierons dans les prochains articles, s'appuie sur la technique du débordement de buffer (*buffer overflow*) permettant à l'attaquant de faire exécuter à l'application des instructions de code personnelles. Enfin, le troisième type principal d'attaque est fondé sur une condition de concurrence (*race condition*), laps de temps entre deux instructions pendant lequel une modification d'un élément du système - souvent un fichier - intervient alors que l'application le considère immuable.

Les deux premiers types d'attaques cherchent souvent à exécuter un shell avec les privilèges du propriétaire de l'application, alors que le troisième est plutôt orienté vers l'accès, généralement en écriture, à des fichiers système protégés. L'accès en lecture est parfois considéré comme une atteinte à la sécurité du système (fichiers personnels, courriers électroniques, fichier des mots de passe **`/etc/shadow`**, et même pseudo-fichiers de configuration du noyau dans **`/proc`**).

Les cibles des attaques de sécurité sont pour l'essentiel les programmes dont le fichier exécutable possède un bit Set-UID (ou Set-GID). Toutefois cela concerne également toute application qui s'exécute sous une identité différente de celle de l'utilisateur avec lequel elle dialogue. Une partie de ces programmes est représentée par les démons système. Un démon est une application généralement démarrée dès l'initialisation du système, qui s'exécute à l'arrière-plan sans terminal de contrôle, et qui réalise des opérations privilégiées pour le compte d'utilisateur quelconque. Par exemple le démon **`lpd`** permet à tout utilisateur de transmettre des documents à l'imprimante, **`sendmail`** reçoit et aiguille des courriers électroniques, ou encore **`apmd`** interroge le Bios pour connaître le niveau de charge d'une batterie sur un ordinateur portable. Il existe également des démons chargés de communiquer avec des utilisateurs externes, par l'intermédiaire du réseau (services Ftp, Html, Telnet...). Ils sont gérés globalement par un serveur nommé **`inetd`** qui assure l'établissement de la connexion.

En définitive nous pouvons en conclure qu'un programme est susceptible de faire l'objet d'attaques s'il dialogue - même brièvement - avec un utilisateur différent de celui qui l'a fait démarrer. Si la conception d'une application fait apparaître une telle caractéristique, il est important de faire preuve de prudence lors du développement, et de garder à l'esprit les risques présentés par les fonctions que nous étudierons ici.

Modification du niveau de privilèges

Lorsqu'une application s'exécute avec un UID effectif différent de son UID réel, c'est dans le but de disposer par moment de privilèges auxquels son utilisateur n'a pas droit directement (accès à des fichiers, appels système réservés...). Toutefois ce besoin ne s'exprime en général que très ponctuellement, par exemple lors de l'ouverture d'un fichier, et le reste du temps l'application pourrait très bien se contenter du niveau de privilèges attribué à son utilisateur. Il est possible de modifier temporairement l'UID effectif d'une application grâce à l'appel-système :

```
int seteuid (uid_t uid);
```

Un processus peut toujours modifier la valeur de son UID effectif en lui celle de son UID réel. Dans ce cas l'ancien UID est mémorisé dans un champ de sauvegarde nommé SUID (*Saved UID*) à ne pas confondre avec le SID (*Session ID*) qui sert à la gestion du terminal de contrôle. Il est également toujours possible de reprendre la valeur de l'UID sauvé pour l'employer en UID effectif. Naturellement un programme ayant un UID effectif nul (*root*) peut modifier à volonté ses UID effectif et réel (c'est ainsi que fonctionne **`/bin/su`**).

Pour limiter les risques d'attaques, il est conseillé de modifier au plus vite l'UID effectif pour employer l'identité réelle de l'utilisateur (R-UID). Lorsqu'une portion de code nécessite des privilèges correspondants à ceux du propriétaire du fichier, on replace temporairement l'UID sauvé dans l'UID effectif. Voici un exemple de code :

```
uid_t e_uid_initial;
uid_t r_uid;

int
main (int argc, char * argv [])
{
  /* Sauvegarde des différents UIDs */
  e_uid_initial = geteuid ();
  r_uid = getuid ();

  /* limitation des droits à ceux du lanceur */
  seteuid (r_uid);
  ...
  fonction_privilegiee ();
  ...
}

void
fonction_privilegiee (void)
{
  /* Restitution des privilèges initiaux */
  seteuid (e_uid_initial);
  ...
  /* Portion nécessitant les privilèges */
  ...
  /* Retour aux droits du lanceur */
  seteuid (r_uid);
}
```

Cette manière de procéder est beaucoup plus sûre que l'inverse, trop souvent rencontrée, et qui consiste à fonctionner avec l'UID effectif initial en permanence, puis à réduire temporairement les privilèges juste avant d'effectuer une opération "à risque". Il faut noter malgré tout que cette réduction des privilèges n'est pas efficace contre les attaques par débordement de buffer. En effet, comme nous le verrons dans le prochain article, ces attaques visent à faire exécuter des instructions personnelles à l'application, et peuvent contenir les appels-système nécessaires pour remonter le niveau de privilège. Cette approche protège toutefois contre les exécutions de

commandes externes que nous allons examiner, mais également contre l'essentiel des conditions de concurrence.

Exécution de commandes externes

Il est fréquent qu'une application ait besoin de faire appel à un service système externe. L'exemple le plus classique est l'invocation de la commande **`mail`** pour transmettre un courrier électronique (rapport d'exécution, alarme, statistiques, etc.) sans avoir besoin de mettre en oeuvre un dialogue complexe avec le système de messagerie. La manière la plus facile de procéder est d'employer la fonction de bibliothèque :

```
int system (const char * commande)
```

Dangers de la fonction system()

Cette fonction présente un danger important : elle invoque le shell lui-même pour exécuter la commande transmise en argument. Or, le comportement du shell est sensible à des éléments que l'utilisateur peut configurer à sa guise. L'exemple le plus frappant provient de la variable d'environnement **`PATH`**. Supposons qu'une application appelle, à un moment ou à un autre, la fonction **`mail`**. Par exemple le programme suivant envoie son propre code-source à l'utilisateur qui l'a lancé :

```
/* system1.c */

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

int
main (void)
{
  if (system ("mail $USER < system1.c") != 0)
    perror ("system");
  return (0);
}
```

Nous supposons que ce programme est installé Set-UID *root* :

```
 cc system1.c -o system1
 su
Password:
[root] chown root.root system1
[root] chmod +s system1
[root] exit
 ls -l system1
-rwsrwsr-x 1 root root 11831 Oct 16 17:25 system1
```

Pour exécuter ce programme, le système lance un shell (avec le nom **`/bin/sh`**) et lui transmet, avec l'option **`-c`**, l'instruction à invoquer. Le shell parcourt alors l'ensemble des répertoires indiqués dans la variable d'environnement **`PATH`** à la recherche d'un exécutable nommé **`mail`**. L'utilisateur n'a qu'à modifier le contenu de cette variable avant de lancer l'application principale. Par exemple, le remplacement suivant :

```
export PATH=.
./system1
```

dirige la recherche de la commande **`mail`** uniquement dans le répertoire courant. Il suffit alors d'y créer un fichier exécutable quelconque (par exemple un script qui lance un nouveau shell) et de le nommer **`mail`** pour que ce programme soit automatiquement exécuté avec l'UID effectif du propriétaire de l'application principale ! Ici, notre script lance **`/bin/sh`**. Toutefois comme il est exécuté avec son entrée standard redirigée (comme la commande **`mail`** initiale), nous devons la rétablir sur le terminal. Ainsi nous créons le script :

```
#! /bin/sh
# Script "mail" fictif lançant un shell en
# rétablissant son entrée standard.
/bin/sh < /dev/tty
```

L'exécution est concluante :

```
 export PATH="."
 ./system1
bash# /usr/bin/whoami
  root
bash#
```

Bien sûr, une première solution consiste à donner le chemin d'accès complet du fichier exécutable visé, par exemple **`/bin/mail`**. Une nouvelle difficulté se dresse alors : l'application dépend de l'installation du système. Si **`/bin/mail`** est généralement disponible sur tous les systèmes, des problèmes vont commencer à se poser par exemple pour trouver GhostScript (se trouve-t il en **`/usr/bin`**, **`/usr/share/bin`**, **`/usr/local/bin`**). D'autre part, un deuxième type d'attaque est alors possible sur certains vieux shells : l'utilisation de la variable d'environnement **`IFS`**. Le shell l'emploie pour distinguer les différents mots de la ligne de commande. Cette variable contient les caractères de séparation à considérer. Par défaut, il s'agit de l'espace, la tabulation et le retour-chariot. Si un utilisateur y ajoute le caractère slash **`/`**, la commande "**`/bin/mail`**" est interprétée par le shell comme "**`bin mail`**". Un fichier exécutable nommé **`bin`** dans le répertoire en cours et une modification du **`PATH`**, comme nous l'avons vu précédemment permettent à ce programme d'être lancé avec l'UID effectif de l'application.

En réalité, sous Linux, la variable d'environnement **`IFS`** ne pose plus de problèmes, car bash la remplit automatiquement avec les caractères par défaut dès son démarrage (comme pdksh également). Mais pour conserver une certaine portabilité à l'application, il faut prévoir que d'autres systèmes peuvent être moins sûrs vis-à-vis de cette variable.

D'autres variables d'environnement posent parfois des problèmes inattendus. Par exemple l'application **`mail`** autorise l'utilisateur à lancer une commande lors de la

rédaction d'un message en employant une séquence d'échappement "**~!**". Si l'utilisateur entre en début de ligne la chaîne "**~!***commande*", la commande indiquée est invoquée. D'autre part l'application **/usr/bin/suidperl** qui sert à faire fonctionner des scripts Perl en version Set-UID appelle, lorsqu'elle détecte un problème, **/bin/mail** pour envoyer un message à *root*. L'application étant Set-UID *root*, l'appel de **/bin/mail** se fait bien entendu sous cette identité. Dans le message transmis à *root*, le nom du fichier posant un problème est présent. Un utilisateur peut donc créer un fichier dont le nom contient un retour-chariot suivi d'une séquence **~!commande** suivi à nouveau d'un retour-chariot. Si un script Perl qui appelle **suidperl** échoue sur un problème bas-niveau lié à ce fichier, un message est émis sous l'identité *root*, contenant la séquence d'échappement de l'application **mail**.

Ce problème ne devrait théoriquement pas exister car le programme **mail** n'accepte pas les séquences d'échappement lorsqu'il est invoqué de manière automatique (sans être piloté depuis un terminal). Malheureusement, une fonctionnalité non documentée de cette application (probablement un reste d'une option de débogage), veut que si la variable d'environnement **interactive** est définie, les séquences soient autorisées. Résultat ? Une faille de sécurité facilement exploitable (et largement exploitée) dans une application justement censée améliorer la sûreté du système. La faute en est partagée. D'abord **/bin/mail** contient une option non documentée et fondamentalement dangereuse puisqu'elle autorise l'exécution de code sous l'unique contrôle des *données* transmises, et qui, pour un utilitaire de courrier électronique, sont *a priori* suspectes. En second lieu, même si les développeurs de **/usr/bin/suidperl** ignoraient l'existence de la variable **interactive**, ils n'auraient jamais dû laisser l'environnement d'exécution intact au moment d'appeler une commande externe lors de l'écriture de ce programme Set-UID *root*.

En fait, Linux ignore totalement les bits Set-UID et Set-GID lors de l'exécution de scripts (voir pour cela **/usr/src/linux/fs/binfmt_script.c** et **/usr/src/linux/fs/exec.c**). Des artifices permettent toutefois de contourner cette règle, comme Perl le fait si bien avec ses propres scripts en passant par **/usr/bin/suidperl** pour honorer indirectement ces bits.

Solutions

Il n'est pas toujours facile de trouver un remplacement pour la fonction **system()**. La première possibilité est d'employer directement les appels-système **execl()** ou **execle()**. Toutefois, la sémantique diffère totalement puisque le programme externe n'est plus invoqué comme une sous-routine, mais la commande appelée remplace le processus en cours. Il est nécessaire de rajouter une duplication du processus, et de séparer nous-mêmes les arguments de la ligne de commande. Ainsi, le programme :

```
if (system ("/bin/lpr -Plisting stats.txt") != 0)
{
  perror ("Impression");
  return (-1);
}
```

devient :

```
pid_t pid;
int   status;

if ((pid = fork()) < 0) {
  perror("fork");
  return (-1);
}
if (pid == 0) {
  /* processus fils */
  execl    ("/bin/lpr", "   lpr", "-   Plisting",
"stats.txt", NULL);
  perror ("execl");
  exit (-1);
}
/* processus père */
waitpid (pid, & status, 0);
if ((! WIFEXITED (status))
 || (WEXITSTATUS (status) != 0)) {
  perror ("Impression");
  return (-1);
}
```

Le code s'alourdit quand même sensiblement ! Dans certaines situations, l'écriture est franchement complexe, par exemple lorsqu'il s'agit de rediriger l'entrée standard de l'application exécutée comme dans :

```
system ("mail root < stat.txt");
```

En effet, la redirection imposée par **<** est établie par le shell. Il est possible de réaliser la même opération, mais au prix d'une manipulation complexe autour d'une séquence **fork()**, **open()**, **dup2()**, **execl()**, etc. Dans ce cas, une solution acceptable est d'employer quand même la fonction **system()**, mais en configurant entièrement l'environnement.

Sous Linux, les variables d'environnement sont stockées sous forme d'un tableau de pointers sur des caractères : **char ** environ**. Ce tableau se termine par NULL. Les chaînes de caractères sont de la forme "**NOM=valeur**".

On commence par effacer tout le contenu de l'environnement en utilisant l'extension Gnu :

```
int clearenv (void);
```

ou en forçant le pointeur

```
extern char ** environ;
```

à prendre la valeur NULL. Puis les variables d'environnement importantes sont initialisées, en utilisant des valeurs bien contrôlées, à l'aide des fonctions

```
int setenv (const char * nom,
            const char * valeur, int ecrase)

int putenv(const char *string)
```

avant d'appeler la fonction **`system()`**. Par exemple :

```
clearenv ();
setenv ("PATH", "/bin:/usr/bin", 1);
setenv ("IFS", " \t\n", 1);
system ("mail root < /tmp/msg.txt");
```

Au besoin, on récupère le contenu de certaines variables utiles avant d'effacer l'environnement (**`HOME`**, **`LANG`**, **`TERM`**, **`TZ`**, etc.). Une vérification stricte du contenu, de la forme, la taille de ces variables s'impose nécessairement. Insistons encore une fois sur l'effacement impératif de tout l'environnement avant de reconstruire les variables indispensables. La faille de sécurité **`suidperl`** n'aurait jamais existé si l'environnement avait été bien effacé.

Par analogie, la protection d'une machine sur un réseau passe dans un premier temps par un refus systématique de toutes les connexions qui lui sont destinées. Ensuite, seuls les services nécessaires à son bon fonctionnement ou utiles pour le réseau sont activés. De la même manière, lors de la programmation d'une application Set-UID, l'environnement doit être vidé puis rempli uniquement avec les variables indispensables.

Dans la continuité, la vérification du format d'un paramètre se fait en comparant la valeur candidate aux formats autorisés. Si la comparaison réussit, le paramètre est validé. Dans le cas contraire, il est immédiatement rejeté. Si le test est fait en utilisant une liste d'expressions invalides du format, le risque de laisser passer une valeur malformée s'accroît, ce qui peut avoir des conséquences désastreuses sur le système.

Il faut remarquer que ce qui présente des dangers avec **`system()`** en présente tout autant avec certaines fonctions dérivées comme **`popen()`**, ou même avec les appels-systèmes **`execlp()`** ou **`execvp()`** qui prennent en charge la variable **`PATH`**.

Exécution indirecte de commandes

Pour améliorer l'ergonomie d'un programme, il est pratique de laisser l'utilisateur libre de configurer une partie importante du comportement du logiciel par le biais de macros par exemple. Pour manipuler des variables et des motifs génériques comme le shell le fait habituellement, il existe une fonction très puissante nommée **`wordexp()`**. Elle nécessite toutefois une grande prudence, car la transmission d'une chaîne du type **`$(`***`commande`***`)`** permet l'exécution de la commande externe mentionnée. Il suffit de lui passer la chaîne "**`$(/bin/sh)`**" pour disposer instantanément d'un shell Set-UID. Afin d'éviter cela **`wordexp()`** dispose d'un attribut nommé **`WRDE_NOCMD`** qui désactive l'interprétation des séquences **`$( )`**.

Il faut également prendre garde, lors de l'invocation de commandes externes telles que nous les avons vues plus haut, à ne pas appeler un utilitaire qui à son tour offre un mécanisme d'échappement vers un shell (comme les séquences **`:!`***`commande`* de vi par exemple). Il est difficile d'en faire la liste, certaines applications sont évidentes (éditeurs de texte, gestionnaire de fichiers...) d'autres s'avèrent plus difficile à déceler (comme nous l'avons vu avec **`/bin/mail`**) ou disposer de modes de débogage dangereux.

Conclusion

Cet article illustre plusieurs préceptes :

- tout ce qui est externe à un programme Set-UID *root* doit être validé ! Ceci concerne aussi bien les variables d'environnement que les paramètres fournis au programme (ligne de commande, fichier de configuration...) ;
- les privilèges doivent être réduits dès le début de l'exécution pour n'être augmentés que le plus brièvement possible et uniquement lorsqu'il n'y a pas d'autres solutions ;
- La "*sécurité en profondeur*" reste toujours aussi essentielle : chaque mesure de protection prise diminue le nombre total de personnes susceptible de venir à bout des défenses.

Le prochain article traitera de la mémoire, de son organisation, des appels de fonctions... tout cela pour préparer le terrain aux *débordements de buffer* (*buffer overflows*). Nous verrons également comment construire un *shellcode*.

Christophe BLAESS - ccb@club-internet.fr
Christophe GRENIER - grenier@nef.esiea.fr
Frédéreric RAYNAL - pappy@users.sourceforge.net