perlsec

Table des matières

1 NAME/NOM

perlsec - Sécurité de Perl

2 DESCRIPTION

Perl est conçu pour faciliter une programmation sûre, même lorsqu'il tourne avec des privilèges spéciaux, comme pour les programmes setuid ou setgid. Contrairement à la plupart des shells de ligne de commande, qui sont basés sur de multiples passes de substitution pour chaque ligne du script, Perl utilise un procédé d'évaluation plus conventionnel contenant moins de pièges cachés. De plus, comme le langage a plus de fonctionnalités intégrées, il doit moins se reposer sur des programmes externes (et potentiellement peu sûrs) pour accomplir ses tâches.

Perl met en oeuvre automatiquement un ensemble de vérifications spécifiques à la sécurité, appelé *taint mode* (mode souillé, NDT), lorsqu'il détecte que son programme tourne avec des identifiants de groupe ou d'utilisateurs réel et effectif différents. Le bit setuid dans les permissions d'Unix est le mode 04000, le bit setgid est le mode 02000 ; ils peuvent être placés l'un ou l'autre, ou les deux à la fois. Vous pouvez aussi activer le taint mode explicitement en utilisant l'option de ligne de commande **-T**. Cette option est *fortement* conseillée pour les programmes serveurs et pour tout programme exécuté au nom de quelqu'un d'autre, comme un script CGI. Une fois que le taint mode est activé, il l'est pour tout le reste de votre script.

Lorsqu'il est dans ce mode, Perl prend des précautions spéciales appelées *taint checks* (vérification de pollution, NDT) pour éviter aussi bien les pièges évidents que les pièges subtils. Certaines de ces vérifications sont raisonnablement simples, comme vérifier que personne ne peut écrire dans les répertoires du path ; les programmeurs précautionneux ont toujours utilisé de telles méthodes. D'autres vérifications, toutefois, sont mieux supportées par le langage lui-même, et ce sont ces vérifications en particulier qui contribuent à rendre un programme Perl set-id plus sûr qu'un programme équivalent en C.

Vous ne pouvez pas utiliser des données provenant de l'extérieur de votre programme pour modifier quelque chose d'autre à l'extérieur – au moins pas par accident. Tous les arguments de ligne de commande, toutes les variables d'environnement, toutes les informations locales (voir *perllocale*), résultant de certains appels au système (`readdir()`, `readlink()`, la variable de `shmread()`, les messages renvoyés par `msgrcv()`, le mot de passe, les champs gecos et shell des appels `getpwxxx()`), et toutes les entrées par fichier sont marquées comme "souillées". Les données souillées ne peuvent pas être utilisées directement ou indirectement dans une commande qui invoque un sous-shell, ni dans toute commande qui modifie des fichiers, des répertoires ou des processus, **avec les exceptions suivantes** :

– Les fonctions `print` et `syswrite` **ne** vérifient **pas** si leurs arguments sont souillées ou non.
– La sûreté des valeurs utilisées comme méthodes symboliques

```
$obj->$method(@args);
```

ou comme références symboliques à des sous-programmes

```
&{$foo}(@args);
$foo->(@args);
```

n'est pas vérifiée. Cela nécessite une attention particulière à moins que vous acceptiez que des données externes affectent votre flux de contrôle. Si vous ne limitez pas très précisément ce que seront ces valeurs symboliques, les personnes les fournissant pourront appeler des fonctions **en dehors** de votre code Perl, telle que POSIX::system, et seront donc à même d'exécuter n'importe quel code externe.
– Les clés des tables de hachage ne sont **jamais** souillées.

Pour des raisons d'efficacité, Perl a un point de vue très conservateur sur ce qui est souillé ou non. Si une expression contient des données souillées, n'importe quelle sous-expression sera considérée comme souillée, même si la valeur de cette sous-expression n'est pas affectée par des données souillées.

Puisque la sûreté est associée à chaque valeur scalaire, certains éléments d'un tableau peuvent être souillés et d'autres pas. Les clés d'une table de hachage ne sont **jamais** souillées.

Par exemple :

```
$arg = shift;               # $arg est souillée
$hid = $arg, 'bar';         # $hid est aussi souillée
$line = <>;                 # Souillée
$line = <STDIN>;            # Souillée aussi
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;              # Encore souillée
$path = $ENV{'PATH'};       # Souillée, mais voir plus bas
$data = 'abc';              # Non souillée

system "echo $arg";         # Non sûr
system "/bin/echo", $arg;   # Considéré comme non sûr
                            # (Perl ne sait rien de /bin/echo)
system "echo $hid";         # Non sûr
system "echo $data";        # Non sûr jusqu'à ce que PATH soit fixé

$path = $ENV{'PATH'};       # $path est désormais souillée

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'};       # $path N'est maintenant PLUS souillée
system "echo $data";        # Est désormais sûr !

open(FOO, "< $arg");        # ok - fichier en lecture seule
open(FOO, "> $arg");        # Pas ok - tentative d'écriture

open(FOO,"echo $arg|");     # Pas ok
open(FOO,"-|")
  or exec 'echo', $arg;     # Pas ok non plus

$shout = `echo $arg`;       # Non sûr, $shout est maintenant souillée

unlink $data, $arg;         # Non sûr
umask $arg;                 # Non sûr

exec "echo $arg";           # Non sûr
exec "echo", $arg;          # Non sûr
exec "sh", '-c', $arg;      # Extrêmement non sûr !

@files = <*.c>;             # Non sûr (utilise readdir() ou équivalent)
@files = glob('*.c');       # Non sûr (utilise readdir() ou équivalent)
```

```
# Les versions de Perl antérieures à 5.6.0 utilisaient un programme
# externe pour trouver les noms de fichiers dans <*.c> ou glob('*.c').
# Mais dans tous les cas, le résultat est souillé puisque la liste des
# noms de fichiers provient de l'extérieur du programme.

$bad = ($arg, 23);          # $bad est souillé
$arg, `true`;               # Non sûr (même si ça ne l'est pas)
```

Si vous essayez de faire quelque chose qui n'est pas sûr, vous obtiendrez une erreur fatale disant quelque chose comme "Insecure dependency" ou "Insecure $ENV{PATH}".

L'exception à la règle "une valeur souillée souille l'ensemble de l'expression" est l'opérateur ternaire conditionnel `?:`. Puisque le code utilisant l'opérateur conditionnel :

```
$result = $valeur_souillee ? "Pas souillé" : "Pas souillé non plus";
```

est en fait :

```
if ($valeur_souillee) {
  $result = "Pas souillé";
} else {
  $result = "Pas souillé non plus";
}
```

cela n'aurait pas beaucoup de sens de souillé `$result`.

2.1 Blanchiement et détection des données souillées

Pour tester si une variable contient des données souillées, et quels usages provoqueraient ainsi un message "Insecure dependency", vous pouvez utiliser la fonction `tainted()` du module Scalar::Util, disponible sur CPAN ou inclus dans Perl depuis la version 5.8.0. Ou vous pouvez utiliser la fonction `is_tainted` suivante :

```
sub is_tainted {
  return ! eval { eval("#" . substr(join('', @_), 0, 0)); 1; };
}
```

Cette fonction utilise le fait que la présence de données souillées n'importe où dans une expression rend toute l'expression souillée. Il serait inefficace de tester la sûreté de tous les arguments pour tous les opérateurs. Au lieu de cela, l'approche légèrement plus efficace et conservatrice qui est utilisée est que si une valeur souillée a été accédée à l'intérieur d'une expression, alors la totalité de l'expression est considérée comme souillée.

Mais le test de pureté ne vous fournit rien d'autre. Parfois, vous devez juste rendre vos données propres. Une valeur peut être blanchie en l'utilisant comme clé de table de hachage. Sinon, la seule façon d'outrepasser le mécanisme de pollution est de référencer des sous-motifs depuis une expression régulière. Perl présume que si vous référencez une sous-chaîne en utilisant $1, $2, etc., c'est que vous saviez ce que vous étiez en train de faire lorsque vous rédigiez le motif. Cela implique un peu de réflexion – ne blachissez pas tout aveuglément, ou vous détruisez la totalité du mécanisme. Il est meilleur de vérifier que la variable ne contient que des bons caractères (pour certaines valeurs de "bon") plutôt que de vérifier qu'elle contient un quelconque mauvais caractère. C'est parce qu'il est beaucoup trop facile de manquer un mauvais caractère auquel vous n'avez jamais pensé.

Voici un test pour s'assurer que les données ne contiennent rien d'autre que des caractères de "mots" (alphabétiques, numériques et souligné), un tiret, une arobase, ou un point.

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;                     # $data est maintenant propre
} else {
    die "Bad data in '$data'";      # tracer cela quelque part
}
```

Ceci est assez sûr car /\w+/ ne correspond normalement pas aux métacaractères du shell, et les points, tirets ou arobases ne veulent rien dire de spécial pour le shell. L'usage de /.+/ n'aurait pas été sûr en théorie parce qu'il laisse tout passer, mais Perl ne vérifie pas cela. La leçon est que lorsque vous blanchissez, vous devez être excessivement précautionneux avec vos motifs. Le blanchiment des données à l'aide d'expressions régulières est le *seul* mécanisme pour nettoyer les données polluées, à moins que vous n'utilisiez la stratégie détaillée ci-dessous pour forker un fils ayant des privilèges plus faibles.

L'exemple ne nettoie pas $data si use locale est en cours d'utilisation, car les caractères auxquels correspond \w sont déterminés par la localisation. Perl considère que les définitions locales ne sont pas sûres car elles contiennent des données extérieures au programme. Si vous écrivez un programme conscient de la localisation, et si voulez blanchir des données avec une expression régulière contenant \w, mettez no locale avant l'expression dans le même bloc. Voir SÉCURITÉ in *perllocale* pour plus de précisions et des exemples.

2.2 Options sur la ligne "#!"

Quand vous rendez un script exécutable, de façon à pouvoir l'utiliser comme une commande, le système passera des options à perl à partir de la ligne #! du script. Perl vérifie que toutes les options de ligne de commande données à un script setuid (ou setgid) correspondent effectivement à celles placées sur la ligne #!. Certains Unix et environnements cousins imposent une limite d'une seule option sur la ligne #!, vous aurez donc peut-être besoin d'utiliser quelque chose comme -wU à la place de -w -U sous ces systèmes (ce problème ne devrait se poser qu'avec les Unix et les environnement proches qui supportent #! et les scripts setuid ou setgid).

2.3 Mode taint et @INC

Lorsque le mode "taint" (-T) est actif, le répertoire "." ne fait plus partie de @INC et les variables d'environnement PERL5LIB et PERLLIB sont ignorées par Perl. Vous pouvez encore modifier @INC depuis l'extérieur du programme en utilisant l'option -I de la ligne de commande comme expliqué dans *perlrun*. Les deux variables d'environnement sont ignorées parce qu'elles sont cachées et qu'un utilisateur qui lance un programme ne sait pas obligatoirement qu'elles existent alors que l'option -I est clairement visible et peut donc être tolérée.

Un autre moyen de modifier @INC sans modifier le programme consiste à utiliser la directive lib de la manière suivante :

```
perl -Mlib=/foo programme
```

L'avantage de -Mlib=/foo sur -I/foo est de gérer les éventuelles occurences multiples d'un même répertoire (le répertoire ne sera pas répété dans @INC).

Notez que si un chemin souillé est ajouté à @INC, le problème suivant sera détecté :

```
Insecure dependency in require while running with -T switch
```

2.4 Nettoyer votre PATH

Pour les messages "Insecure $ENV{PATH}", vous avez besoin de fixer $ENV{'PATH'} à une valeur connue, et chaque répertoire dans le PATH doit être spécifié sous la forme d'un chemin absolu et non modifiable par d'autres utilisateurs que son propriétaire et son groupe. Vous pourriez être surpris d'obtenir ce message alors même que le chemin vers votre exécutable est un chemin absolu. Ceci n'est *pas* généré parce que vous n'avez pas fourni un chemin complet au programme mais plutôt parce que vous n'avez jamais défini votre variable d'environnement PATH, ou vous ne l'avez pas défini comme quelque chose de sûr. Puisque Perl ne peut pas garantir que l'exécutable en question ne vas pas exécuter un autre programme qui dépend de votre PATH, il s'assure que vous avez défini le PATH.

Le PATH n'est pas la seule variable d'environnement qui peut poser des problèmes. Puisque certains shells peuvent utiliser les variables IFS, CDPATH, ENV, et BASH_ENV, Perl vérifie que celles-ci sont soit vides, soit propres, lorsqu'il démarre des sous- processus. Vous pourriez désirer ajouter quelque chose comme ceci à vos scripts setid et blanchisseurs.

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Rend %ENV plus sûr
```

Il est aussi possible de s'attirer des ennuis avec d'autres opérations qui ne se soucient pas si elles utilisent des valeurs souillées. Faites un usage judicieux des tests de fichiers quand vous avez affaire à un nom de fichier fournis par l'utilisateur. Lorsque c'est possible, réalisez les ouvertures et compagnie **après** avoir correctement abandonné les privilèges d'utilisateur (ou de groupe !) particuliers. Perl ne vous empêche pas d'ouvrir en lecture des noms de fichier souillés, alors faites attention à ce que vous imprimez. Le mécanisme de pollution est destiné à prévenir les erreurs stupides, pas à supprimer le besoin de réflexion.

Perl n'appelle pas de shell pour expanser les métacaractères (wildcards, NDT) quand vous passez des listes de paramètres explicites à `system` et `exec` au lieu de chaînes pouvant contenir des métacaractères. Malheureusement, les fonctions `open`, `glob`, et backtick (substitution de commande avec "`", NDT) ne fournissent pas de telles conventions d'appel alternatives, alors d'autres subterfuges sont requis.

Perl fournit une façon raisonnablement sûre d'ouvrir un fichier ou un tube depuis un programme setuid ou setgid : créez juste un processus fils ayant des privilèges réduits et qui fera le sale boulot pour vous. Tout d'abord, créez un fils en utilisant la syntaxe spéciale de `open` qui connecte le père et le fils par un tube. Puis le fils redéfinit son ensemble d'ID et tous les autres attributs dépendant du processus, comme les variables d'environnement, les umasks, les répertoires courants, pour retourner à des valeurs originelles ou connues comme sûres. Puis le processus fils, qui n'a plus la moindre permission spéciale, réalise le `open` ou d'autres appels système. Finalement, le fils passe à son père les données auxquelles il parvient à accéder. Puisque le fichier ou le tube ont été ouverts par le fils alors qu'il tournait avec des privilèges inférieurs à celui du père, il ne peut pas être trompé et faire quelque chose qu'il ne devrait pas.

Voici une façon de faire des substitutions de commande de façon raisonnablement sûre. Remarquez comment le `exec` n'est pas appelé avec une chaîne que le shell pourrait interpréter. C'est de loin la meilleure manière d'appeler quelque chose qui pourrait être sujet à des séquences d'échappements du shell : n'appelez tout simplement jamais le shell.

```
use English '-no_match_vars';
die "Can't fork: $!" unless defined($pid = open(KID, "-|"));
if ($pid) {           # parent
  while (<KID>) {
    # faire quelque chose
  }
  close KID;
} else {
  my @temp = ($EUID, $EGID);
  my $orig_uid = $UID;
  my $orig_gid = $GID;
  $EUID = $UID;
  $EGID = $GID;
  # suppression des privilèges
  $UID = $orig_uid;
  $GID = $orig_gid;
  # S'assurer que les privilèges sont réellement partis
  ($EUID, $EGID) = @temp;
  die "Can't drop privileges"
    unless $UID == $EUID  && $GID eq $EGID;
  $ENV{PATH} = "/bin:/usr/bin"; # PATH minimaliste
  exec 'myprog', 'arg1', 'arg2'
    or die "can't exec myprog: $!";
}
```

Une stratégie similaire marcherait pour l'expansion des métacaractères via `glob`, bien que vous pouvez utiliser `readdir` à la place.

La vérification de la sûreté est surtout utile lorsque, même si vous vous faites confiance de ne pas avoir écrit un programme ouvrant toutes les portes, vous ne faites pas nécessairement confiance à ceux qui finiront par l'utiliser et pourraient essayer de le tromper pour qu'il fasse de vilaines choses. C'est le genre de vérification de sécurité qui est utile pour les programmes set-id et les programmes qui sont lancés au nom de quelqu'un d'autre, comme les scripts CGI.

C'est très différent, toutefois, du cas où l'on ne fait même pas confiance à l'auteur du code de ne pas essayer de faire quelque chose de diabolique. Ceci est le type de confiance dont on a besoin quand quelqu'un vous tend un programme que vous n'avez jamais vu auparavant et vous dit : "Voilà, exécute ceci". Pour ce genre de sécurité, jetez un oeil au module Safe, inclu en standard dans la distribution de Perl. Ce module permet au programmeur de mettre en place des compartiments spéciaux dans lesquels toutes les opérations liées au système sont détournées et où l'accès à l'espace de noms est contrôlé avec soin.

2.5 Failles de sécurité

Au-delà des problèmes évidents qui découlent du fait de donner des privilèges spéciaux à des systèmes aussi flexibles que les scripts, sous de nombreuses versions d'Unix, les scripts set-id ne sont pas sûrs dès le départ de façon inhérente. Le problème est une race condition dans le noyau. Entre le moment où le noyau ouvre le fichier pour voir quel interpréteur il

doit exécuter et celui où l'interpréteur (maintenant set-id) se retourne et réouvre le fichier pour l'interpréter, le fichier en question peut avoir changé, en particulier si vous avez des liens symboliques dans votre système.

Heureusement, cette "caractéristique" du noyau peut parfois être invalidée. Malheureusement, il y a deux façons de l'invalider. Le système peut simplement déclarer hors-la-loi les scripts ayant un bit set-id mis, ce qui n'aide pas vraiment. Sinon, il peut simplement ignorer les bits set-id pour les scripts. Si ce dernier cas est vrai, Perl peut émuler le mécanisme setuid et setgid lorsqu'il remarque les bits setuid/gid bits, par ailleurs inutiles, sur des scripts Perl. Il le fait via un exécutable spécial appelé *suidperl* qui est appelé automatiquement pour vous si besoin est.

Toutefois, si la caractéristique du noyau pour les scripts set-id n'est pas invalidée, Perl se plaindra bruyamment que votre script set-id n'est pas sûr. Vous devrez soit invalider la caractéristique du noyau pour les scripts set-id, soit mettre un wrapper C autour du script. Un wrapper C est juste un programme compilé qui ne fait rien à part appeler votre programme Perl. Les programmes compilés ne sont pas sujets au bug du noyau qui tourmente les scripts set-id. Voici un wrapper simple, écrit en C :

```
#define REAL_PATH "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_PATH, av);
}
```

Compilez ce wrapper en un binaire exécutable, puis rendez-*it* setuid ou setgid à la place de votre script.

Ces dernières années, les vendeurs ont commencé à fournir des systèmes libérés de ce bug de sécurité inhérent. Sur de tels systèmes, lorsque le noyau passe le nom du script set-id à ouvrir à l'interpréteur, plutôt que d'utiliser un nom et un chemin sujets à l'ingérence, il passe */dev/fd/3*. C'est un fichier spécial déjà ouvert sur le script, de sorte qu'il ne peut plus y avoir de race condition que des scripts malins pourraient exploiter. Sur ces systèmes, Perl devrait être compilé avec l'option `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. Le programme *Configure* qui construit Perl essaye de trouver cela tout seul, vous ne devriez donc jamais avoir à spécifier cela vous-même. La plupart des versions modernes de SysVr4 et BSD 4.4 utilisent cette approche pour éviter la race condition du noyau.

Avant la version 5.6.1 de Perl, des bugs dans le code de **suidperl** pouvait introduire des failles de sécurité.

2.6 Protection de vos programmes

Il existe de nombreuses façons de cacher le source de vos programmes Perl, avec des niveaux variables de "sécurité".

Tout d'abord, toutefois, vous *ne* pouvez *pas* retirer la permission en lecture, car le code source doit être lisible de façon à être compilé et interprété (cela ne veut toutefois pas dire que le source d'un script CGI est lisible par n'importe qui sur le web). Vous devez donc laisser les permissions au niveau socialement amical de 0755. Ceci laisse voir votre source uniquement aux gens de votre système local.

Certaines personnes prennent par erreur ceci pour un problème de sécurité. Si votre programme fait des choses qui ne sont pas sûres, et s'appuie sur le fait que les gens ne savent pas comment exploiter ces failles, il n'est pas sûr. Il est souvent possible pour quelqu'un de déterminer les failles de sécurité et de les exploiter sans voir le source. La sécurité par l'obscurité, expression désignant le fait de cacher vos bugs au lieu de les corriger, est vraiment une faible sécurité.

Vous pouvez essayer d'utiliser le chiffrement via des filtres de sources (Filter::* sur CPAN ou Filter::Util::Call et Filter::Simple depuis Perl 5.8). Mais les craqueurs peuvent encore le déchiffrer. Vous pouvez essayer d'utiliser le compilateur de byte code et l'interpréteur décrit ci-dessous, mais les craqueurs peuvent encore le décompiler. Vous pouvez essayer d'utiliser le compilateur de code natif décrit ci-dessous, mais les craqueurs peuvent encore le désassembler. Ces solutions posent des degrés variés de difficulté aux gens voulant obtenir votre code, mais aucune ne peut définitivement le dissimuler (ceci est vrai pour tous les langages, pas uniquement Perl).

Si vous êtes inquiet que des gens profitent de votre code, alors le point crucial est que rien à part une licence restrictive ne vous donnera de sécurité légale. Licenciez votre logiciel et pimentez-le de phrases menaçantes comme "Ceci est un logiciel propriétaire non publié de la société XYZ. L'accès qui vous y est donné ne vous donne pas la permission de l'utiliser bla bla bla". Vous devriez voir un avocat pour être sur que votre vocabulaire tiendra au tribunal.

2.7 Unicode

Unicode est une technologie nouvelle et complexe qui peut facilement amener quelqu'un vers de nouvelles embûches liées à la sécurité. Voir *perluniintro* pour la présentation générale, *perlunicode* pour les détails et en particulier perl<perlunicode/"Implications d'Unicode sur la sécurité">.

2.8 Attaques par complexité algorithmique

Certains algorithmes internes utilisés dans l'implémentation de Perl peuvent être attaqués en fournissant des entrées choisies spécialement pour consommer soit beaucoup de temps, soit beaucoup de mémoire, soit les deux. C'est ce qu'on appelle des attaques par *Déni de service* (DoS - Denial of Service).

– La fonction de hachage - l'algorithme utilisé pour "ordonné" les clés des tables de hachage a changé plusieurs fois au cours du développement de Perl, principalement pour des raisons d'efficacité. En Perl 5.8.1, les aspects concernant la sécurité ont aussi été pris en compte.
Dans les versions antérieures à la 5.8.1, il était relativement facile de générer des données qui, en tant que clés de hachage, amenait Perl à consommer énormément de temps à cause d'une mauvaise gestion des structures internes de hachage. En Perl 5.8.1, la fonction de hachage est volontairement perturbée par un générateur pseudo-aléatoire afin de rendre très difficile la génération de mauvaises clés de hachage. Voir PERL_HASH_SEED in *perlrun* pour plus d'information.
La perturbation aléatoire est utilisée par défaut mais si quelqu'un souhaite simluer l'ancien comportement, il peut positionner la variable d'environnement PERL_HASH_SEED à zéro (ou toute autre valeur entière). Une raison pour vouloir retrouver cet ancien comportement est qu'avec le nouveau comportement, deux exécutions successives de Perl ordonnent les clés de hachage différemment ce qui peut perturber des applications (avec Data::Dumper, les sorties de deux exécutions sur les même données ne sons pas identiques).
Perl n'a jamais garanti l'ordre des clés des tables de hachage et cet ordre a déjà changé plusieurs fois depuis l'arrivée de Perl 5. De plus, l'ordre des clés a toujours été et continue d'être modifié par une insertion.
Notez aussi que bien que cet ordre puisse être considéré comme aléatoire, il ne devrait pas **pas** être utilisé pour du mélange aléatoire de liste (utilisez la fontion List::Util::shuffle() du module *List::Util* qui est un module standard depuis Perl 5.8.0 ou le module CPAN Algorithm::Numerical::Shuffle), pour générer des permutations (utilisez les modules CPAN Algorithm::Permute ou Algorithm::FastPermute) ou pour des applications de cryptographie.
– Les expressions rationnelles - Le moteur d'expression rationnelle de Perl est aussi appelé NFA (Non-deterministic Finite Automaton - Automate fini non déterministe) ce qui signifie, entre autres, qu'il peut facilement consommer énormément de temps et d'espace mémoire si une expression rationnelle est reconnaissable de nombreuses manières différentes. Des expressions rationnelles correctement conçues peuvent empêcher cela mais c'est loin d'être facile (nous vous recommandons la lecture du livre "Mastering Regular Expressions", voir aussi *perlfaq2*). L'utilisation d'espace mémoire trop important se manifeste par un dépassement de capacité de mémoire (out of memory) de Perl.
– Le tri - l'algorithme de tri rapide (quicksort) utilisé dans les versions de Perl antérieures à la version 5.8.0 pour implémenter la fonction sort() est facilement pertubable pour l'amener à consommer beaucoup de temps. Le simple tri d'une liste déjà triée suffit. Depuis la version 5.8.0 de Perl, un autre algorithme de tri est utilisé : le 'mergesort'. Ce nouvel algorithme est insensible à ses données d'entrée. Il ne peut donc plus être dupé.

Pour plus d'information voir http://www.cs.rice.edu/~scrosby/hash/ et n'importe quel livre d'informatique abordant la complexité algorithmique.

3 VOIR AUSSI

perlrun pour sa description du nettoyage des variables d'environnement.

4 TRADUCTION

4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

4.2 Traducteur

Tarduction initiale : Roland Trique <*roland.trique@free.fr*>. Mise à jour : Paul Gaborit <paul.gaborit at enstimac.fr>.

4.3 Relecture

Régis Julié <*regis.julie@cetelem.fr*>

5 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.