

# DOS par complexité

Philippe PRADOS

[site@philippe.prados.name](mailto:site@philippe.prados.name)



*Préservez l'environnement,  
N'imprimez pas ce document*

## TABLE DES MATIERES

1.	Table de hash.....	3
1.1	Comment empêcher l'attaque d'une table de hash ?.....	5
2.	Expression régulière.....	8
2.1	Comment empêcher l'attaque d'expression régulière ?.....	9
3.	Autres attaques.....	10

## Avant-propos

*Des algorithmes ont été inventés depuis de nombreuses années, afin d'améliorer l'efficacité des traitements. A partir d'analyses statistiques sur les données manipulées et les traitements associés, différentes propositions permettent d'améliorer considérablement les vitesses d'exécutions. Les algorithmes sont prévus pour être efficaces dans le cas moyen, mais pas pour le pire. Un pirate peut volontairement exploiter les implémentations pour les placer dans les pires conditions. Avec un petit volume de données, il peut alors avoir un impact très important sur l'application, la ralentissant considérablement. Certaines attaques permettent de neutraliser un détecteur d'intrusion, afin de camoufler une attaque plus importante. D'autres pirates ont comme objectif d'interdire l'utilisation de l'application publiée sur Internet. Nous allons étudier quelques algorithmes vulnérables à ces attaques.*

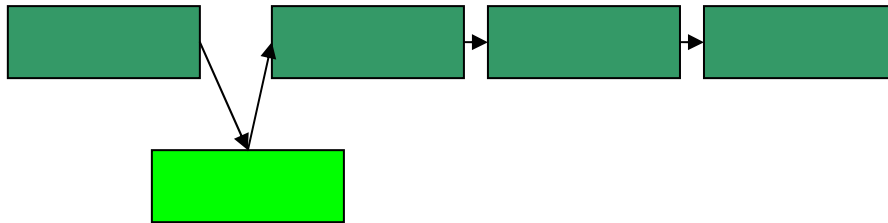
Pour calculer la complexité d'un algorithme, différentes techniques sont utilisées. Généralement, les algorithmes sont différenciés par le temps moyen d'exécution par rapport au nombre d'entrée. Par exemple, l'insertion dans une table de hash de  $n$  éléments prend un temps moyen en  $O(n)$ . Mais dans le cas limite où la table de hash est dégénérée, le temps moyen d'insertion est alors en  $O(n^2)$ .

## 1. TABLE DE HASH

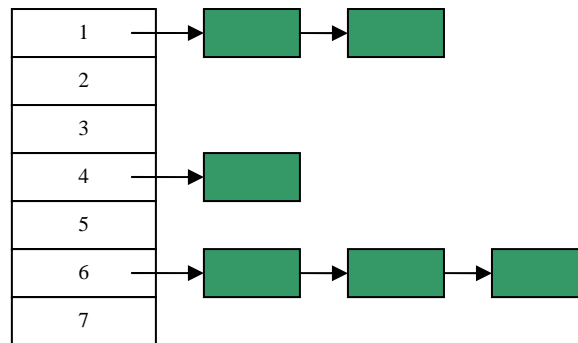
Pour bien comprendre, regardons comment fonctionne une table de hash.

L'objectif est de mémoriser des objets dans un conteneur, et de pouvoir les retrouver très rapidement. La première approche consiste à construire un tableau suffisamment grand pour contenir tous les objets, et de consommer les places disponibles lors des insertions. Pour rechercher un élément, il faut alors parcourir toute la table. Il y a une chance sur deux de trouver l'objet avant d'avoir atteint la moitié de la table. Cette approche présente un inconvénient, il faut estimer correctement la taille du tableau. S'il est trop grand, il y a une perte de place mémoire. Si le tableau est trop petit, il faut en construire un autre plus grand, transférer les objets d'un tableau à l'autre, et enfin, détruire le premier tableau. Cela consomme des ressources mémoires et CPU.

Pour améliorer cela, les développeurs utilisent généralement des listes. Chaque élément du tableau est porté par un nœud de la liste. Des pointeurs permettent de référencer les nœuds suivant et précédant. Avec cette approche, ajouter un objet n'est pas très coûteux. Par contre, pour rechercher un élément, il faut encore parcourir la liste. Si elle est triée, il faut  $O(n^2)$  étapes en moyenne, pour insérer les éléments.



Pour améliorer cela, les tables de hash ont été inventées. L'idée est la suivante. Il serait sympathique de pouvoir estimer à partir d'où l'objet recherché se trouve. Pour pouvoir faire cela, un calcul est effectué sur l'objet à insérer. On appelle cela, un calcul de hash. Le but de ce calcul est d'obtenir un nombre représentatif de l'objet. Il faut que le calcul soit choisi judicieusement pour produire une grande variabilité de valeurs, suivant les objets insérés. Plusieurs objets peuvent avoir le même résultat. Il faut faire en sorte que ces collisions soient rares. Un tableau peut alors référencer des listes courtes pour chaque valeur du calcul.



Pour rechercher un objet, le même algorithme est utilisé pour estimer la liste où il est possible de trouver l'objet. Cette liste est alors extraite du conteneur de liste, et l'objet est recherché linéairement dans une liste réduite.

Pour pouvoir faire cela, un tableau dont la taille est un nombre premier est alloué en mémoire. Ce tableau permettra de référencer les différentes sous-listes. Un algorithme de calcul de hash est sélectionné. Pour chaque objet à insérer, la valeur de hash est calculée. Le reste de la division du résultat par la taille du tableau est alors calculé. Cela correspond à la position de la liste à consulter pour trouver l'objet. Si les listes deviennent trop importantes, ou si la saturation du tableau de liste dépasse les 70%, un tableau de liste plus grand est alloué. Sa taille est toujours un nombre premier, car cela réduit les risques de collisions. Puis, tous les objets sont répartis dans le nouveau tableau. Ce traitement est long et doit être rare. En estimant correctement la taille initiale du tableau de liste, on évite généralement ces traitements.

Avec cette approche, en moyenne, il faut  $O(n)$  étape pour insérer un objet. Mais, dans le cas le pire, si les objets à insérer sont judicieusement choisis, les collisions sont systématiques. L'algorithme devient alors équivalent à une simple liste chaînée. L'insertion et la recherche dégénèrent en  $O(n^2)$ .

Que va faire le pirate pour réussir un déni de service d'une application ? Il doit être capable de générer des données insérées dans une table de hash. Cela est relativement facile. Les champs d'un formulaire, les en-têtes des pages, les adresses IP, etc. sont très souvent placés dans des

tables de hash, afin d'accélérer leurs consultations. Il doit pouvoir insérer un volume conséquent pour que le ralentissement d'exécution se fasse sentir. C'est généralement le cas des tableaux de hash, partagés par les utilisateurs (liste noire, caches partagées, etc.)

En connaissant l'algorithme utilisé pour le calcul de hash, et l'algorithme permettant de faire progresser la taille du tableau en cas de débordement, il est possible de produire des valeurs entraînant systématiquement des collisions.

Les données ne pouvant être en double dans un tableau de hash, il faut inverser le calcul du hash pour trouver différentes valeurs arrivant au même résultat.

Comme la valeur de hash subit un modulo suivant la taille du tableau, il existe alors d'autres valeurs entraînant le même résultat pour une taille donnée du tableau de hash. Sont candidates toutes les chaînes de caractères dont le reste de la division entière de la valeur de hash par la taille du conteneur donnent la valeur recherchées.

Par exemple, l'objet `String` de Java est très souvent utilisé comme clef d'une table de hash. La méthode de production du hash du JDK5 est la suivante :

```
class String
{
    ...
    public int hashCode()
    {
        int h = hash;
        if (h == 0)
        {
            int off = offset;
            char val[] = value;
            int len = count;

            for (int i = 0; i < len; i++)
            {
                h = 31*h + val[off++];
            }
            hash = h;
        }
        return h;
    }
}
```

Tous les caractères sont multiplié par 31 et ajouté au suivant. Un cache est maintenant mémorisé dans l'instance. Ce n'était pas le cas des premières versions. Le résultat est placé dans un entier.

Comment obtenir des collisions ? Avec trois caractères A, B et C, de valeurs ASCII respectives 65, 66, 67, le calcul est le suivant :

Etape 1 :  $0*31+65=65$

Etape 2 :  $65*31+66=2081$

Etape 3 :  $2081*31+67=64578$

Il existe une relation entre les caractères. Une unité d'un caractère correspond à -31 du suivant. Si j'ajoute 1 à 65, je dois enlever 31 à 66.

Avec des jeux de chaînes de caractères donnant le même résultat de hash, il est possible de les combiner indéfiniment pour construire des chaînes plus importantes dont tous les résultats du calcul sont identiques.

Par exemple, les chaînes de caractères dont les valeurs sont : [65,66,67], [66,35,67] et [65,67,36] ont les mêmes valeurs de hash, quelques soit la taille du tableau de hash. Dans le conteneur, ces trois valeurs sont mémorisées dans une liste chaînée à cause des collisions.

Il est donc très facile de produire une multitude de chaînes de caractères entraînant le même résultat. Pour cela, il faut sélectionner des couples de caractères donnant le même résultat, et les combiner pour générer des chaînes de caractères plus ou moins longues en alternant les membres du couple. Pour avoir des caractères compris entre '0' et 'z' afin de respecter les éventuelles filtres sur les caractères, comme il faut pouvoir soustraire 31 en restant dans ce jeu de caractères, il faut travailler avec les caractères compris entre '0' et 'z'-31. Choisissons le couple « PP » et « Q1 ». Les deux chaînes ont la même valeur de hash et fonds partis des caractères valides. Ensuite, avec un nombre traité comme une valeur binaire, il est aisé d'afficher l'une ou l'autre valeurs, suivant la valeur de chaque bit. Pour une génération à partir de 16 bit, il y aura  $16^2$  chaînes de  $16*2=32$  caractères. Toutes auront la même valeur de hash et composées uniquement de caractères alphanumérique. Pour trois bits, les valeurs sont :

```
PPPPPP
Q1PPPP
PPQ1PP
Q1Q1PP
PPPPQ1
Q1PPQ1
PPQ1Q1
Q1Q1Q1
```

Le programme de génération de chaînes de caractères est le suivant.

```
public static void generate()
{
    final int nbBit=31;
    final int maxValue=(0xFFFFFFFF << nbBit) ^ 0xFFFFFFFF;
    StringBuffer buf=new StringBuffer(nbBit*2);
```

```

for (int pattern=0;pattern<maxValue;++pattern)
{
    buf.setLength(0);
    for (int i=0;i<nbBit;++i)
        buf.append(((pattern & (1<<i))==0) ? "PP" : "Q1");
    String key=buf.toString();
    ... // Injection de la clef dans la table de hash
}
}
}

```

Un nombre de bit permet d'indiquer la taille des chaînes produites.

Si l'algorithme est plus complexe, il existe une autre approche. Parmi les résultats de l'algorithme utilisé, il est possible de trouver la valeur zéro. Cela consiste à ré-initialiser l'algorithme. Imaginez un programme capable de produire des chaînes de caractères dont le résultat du calcul de hash donne zéro. Il est alors possible de combiner toutes ces chaînes pour en produire de plus grandes par combinatoire. Le résultat global de toutes les variantes donne zéro.

Pour rechercher les sous chaînes donnant la valeur zéro, une analyse en force brute peut être pratiquée. Cela permet de produire une table de chaînes une fois pour toute, dont les valeurs de hash donnent zéro, et de procéder à des combinaisons pour attaquer une table de hash. Suivant la complexité de l'algorithme de hash, cette étape de recherche peut être importante. Mais, celle-ci étant franchie, le même résultat est exploitable indéfiniment, tant que l'algorithme de calcul n'est pas modifié. Une recherche distribuée peut être entreprises pour identifier les données générant une valeur de hash de zéro. Le reste est un jeu d'enfant.

### 1.1 Comment empêcher l'attaque d'une table de hash ?

Trois failles sont exploitables.

- L'algorithme utilisé permet de facilement trouver des collisions. C'est le cas de la génération d'une valeur de hash pour les chaînes de caractères de java. Un algorithme plus complexe peut corriger cela. L'utilisation d'un calcul cryptographique est préférable.
- L'algorithme utilisé peut générer une valeur de résultat à zéro. Cela peut être corrigés en interdisant cette valeur ou en utilisant une valeur aléatoire par application, pour initialiser les algorithmes de hash.
- L'algorithme utilisé étant réduit à l'aide du calcul du reste d'une division, le nombre de collision est amplifié par la taille du tableau de hash. Cela se corrige en ajoutant un aléa aux valeurs calculées. Un nombre aléatoire est tiré lors du lancement du programme. Il est systématiquement agrégé à la valeur de hash calculé, avant de référencer la table des collisions. L'agrégation peut s'effectuer par un ou exclusif et une addition, afin de maintenir le spectre des valeurs, tout en cassant la prédictibilité.

Les langages de développement proposent des algorithmes de manipulation d'objet, mais aucun n'est résistant à un déni de service. Des attaques ont été publiées pour Java, Perl, Python, TCL...

Pour améliorer les classes de Java, il faut utiliser une classe dont l'algorithme utilisé pour le calcul du hash respecte les contraintes énoncées. L'algorithme est complexe ; la valeur d'initialisation n'est pas prédictible ; ainsi que les collisions dues à l'application d'un module sur la taille du conteneur. La classe `SecureHashString` répond à ces impératifs.

```

package name.prados.philippe.securikit;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

public final class SecureHashString
{
    private static final int rnd_ = new SecureRandom(SecureRandom.getSeed(20))
        .nextInt();

    private final String str_;

    public SecureHashString(String str)
    {
        str_ = str;
    }

    public boolean equals(Object obj)
    {
        try
        {
            return str_.equals(((SecureHashString) obj).str_);
        }
        catch (ClassCastException x)
        {
            return false;
        }
    }

    public int hashCode()
    {

```

```

try
{
    byte[] digest = MessageDigest.getInstance(
        "SHA").digest(
        str_.getBytes());
    int hash = rnd_;
    for (int i = 0; i < digest.length; i += 4)
    {
        hash ^= digest[i] | (digest[i + 1] << 8)
            | (digest[i + 2] << 16) | (digest[i + 3] << 24);
    }
    return (hash == rnd_) ? hash + 1 : hash;
}
catch (NoSuchAlgorithmException e)
{
    return rnd_;
}

}

public String toString()
{
    return str_;
}
}

```

### Source 1

Cette classe est utilisée pour alimenter les clefs.

```

map.put(new SecureHashKeyWrapper("abc"),obj);
map.find(new SecureHashKeyWrapper("abc"));

```

Quelle est la réalité de la menace ? En alimentant une table de hash avec 300.000 objets ayant la même valeur de hash, le temps d'insertion est tombé à une demi seconde sur processeur Intel à 1,50 Ghz. Pour alimenter une table de cette taille, il faut plusieurs heures dans une JVM. A partir du réseau il faut utiliser une attaque de déni de service répartie pour obtenir des résultats convainquant. Le volume réseau est très bruyant.

Une autre solution pour éviter ce problème, est de limiter la taille maximum de la table de hash. Le classe [LimitedMap](#) permet cela.

```

package name.prados.philippe.securikit;

import java.util.Collection;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class LimitedMap implements Map
{
    public interface Overflow
    {
        public Object overflow(LimitedMap map, int number);
    }

    public static final Overflow Nothing = new Overflow()
    {
        public Object overflow(LimitedMap map, int number)
        {
            return null;
        }
    };

    public static final Overflow Exception = new Overflow()
    {
        public Object overflow(LimitedMap map, int number)
        {
            throw new ArrayStoreException();
        }
    };

    public static final Overflow RemoveRandom = new Overflow()
    {
        public Object overflow(LimitedMap map, int number)
        {
            for (Iterator i = map.entrySet().iterator();
                i.hasNext() && number > 0; --number)
            {

```

```

        i.next();
        i.remove();
    }
    return null;
}
};

private final Map next_;

private final int maxsize_;

private final Overflow overflow_;

public LimitedMap(Map next, int maxsize)
{
    this(next, maxsize, Exception);
}

public LimitedMap(Map next, int maxsize, Overflow overflow)
{
    next_ = next;
    maxsize_ = maxsize;
    overflow_ = overflow;
}

public int size()
{
    return next_.size();
}

public boolean isEmpty()
{
    return next_.isEmpty();
}

public boolean containsKey(Object key)
{
    return next_.containsKey(key);
}

public boolean containsValue(Object value)
{
    return next_.containsValue(value);
}

public Object get(Object key)
{
    return next_.get(key);
}

public Object put(Object key, Object value)
{
    if (next_.size() >= maxsize_)
        return overflow_.overflow(
            this, 1);
    return next_.put(
        key, value);
}

public Object remove(Object key)
{
    return next_.remove(key);
}

public void putAll(Map t)
{
    if (next_.size() + t.size() >= maxsize_)
        overflow_.overflow(
            this, t.size());
    next_.putAll(t);
}

public void clear()
{
    next_.clear();
}

```

```

public Set keySet()
{
    return next_.keySet();
}

public Collection values()
{
    return next_.values();
}

public Set entrySet()
{
    return next_.entrySet();
}
}

```

Une instance permet de paramétrer le comportement de la classe si trop d'objets sont présent. Trois stratégies sont proposées : ignorer les nouvelles entrées, générer une exception, supprimer certains objets au hasard avant d'insérer les nouveaux.

```
Map map = new LimitedMap(new java.util.Hashtable(), 3, LimitedMap.RemoveRandom);
```

## 2. EXPRESSION REGULIERE

D'autres algorithmes sont plus sensibles aux attaques exploitant les complexités algorithmique.

Pour analyser des expressions régulières, plusieurs familles de moteurs peuvent être utilisés : Les NFA ou les DFA. Il s'agit d'une différence d'approche dans l'analyse de l'expression. Un moteur NFA analyse tous les chemins possibles de l'expression, et retourne en arrière lors d'un échec. Un moteur DFA maintient une liste des candidats encore possible, et l'élague au fur et à mesure des caractères rencontrés. Chaque caractère n'est analysé qu'une seule fois. Un moteur DFA est plus complexe à compiler, mais plus rapide à l'exécution. Un moteur DFA est plus rapide à compiler mais moins rapide à l'exécution. Il est fortement dépendant de la rédaction de l'expression régulière, alors qu'un moteur DFA en est indifférent.

Par exemple, une expression (`int|info`) peut être optimisée en `in(t|fo)` pour un moteur NFA. Ainsi, en cas de retour arrière, les deux premières lettres peuvent être considérées comme possibles. Il n'est pas nécessaire d'analyser une nouvelle fois l'ensemble des caractères.

Prenons l'exemple d'une application qui utilise un moteur NFA, sensible à la syntaxe de l'expression régulière. Suivant les cas, le temps d'analyse d'un paramètre peut être linéaire par rapport à la taille de la chaîne  $O(n)$ , quadratique  $O(n^2)$ , cubique  $O(n^3)$  ou exponentielle  $O(n^n)$ .

Exemple	Type
<code>a*</code>	Linéaire
<code>a*[ab]*0</code>	Quadratique
<code>a*[ab]*[ac]*0</code>	Cubique
<code>(a aa)*0</code>	Exponentielle

Un déni de service peut être obtenu en envoyant, en grande quantité, des valeurs suffisamment longues pour entraîner un travail excessif du serveur. Cela occasionne de nombreux retours arrière. Par exemple, dans le cas d'une expression exponentielle, une chaîne de trente caractères peut être analysée en une minute en consommant 100% de la CPU.

Pour vous en convaincre, consultez la démonstration ici : <http://jakarta.apache.org/oro/demo.html> .

Indiquez l'expression AWK (moteur DFA) `(a|aa)*0`, demandez `matches()`, indiquez la valeur `aaaaaaaaaaaaaaaaaaaaaZ` et lancez le traitement. Vous obtenez rapidement un résultat. Modifiez alors l'algorithme utilisé en Perl5 (moteur NFA), et lancez à nouveau le traitement. Suivant la taille de la chaîne, le résultat peut mettre un temps très important pour analyser l'expression. Notez que le moteur DFA proposé par ORO ne gère pas les caractères uni-codes.

Pourquoi cette expression pose problème ? Lors de l'analyse de la chaîne de caractère, un premier 'a' est consommé. L'étoile indique que la règle est à nouveau applicable. Les 'a' suivant sont également consommés, jusqu'à la rencontre du 'Z'. La règle prend fin et la suivante est vérifiée. Elle indique qu'elle attend un 'O'. Comme ce n'est pas le cas, la règle précédente recule d'une étape. Le dernier 'a' est réintroduit dans le moteur. La deuxième règle est alors testée, celle attendant deux 'a'. Le premier caractère est valide, mais pas le second. La règle est alors rejeté. Un deuxième 'a' est réinjecté dans le moteur. Puis la première règle est testée à nouveau. Et ainsi de suite. Pour chaque 'a', toutes les combinaisons possibles possédant un 'a' et deux 'a' sont testés, jusqu'à découvrir l'échec de l'analyse lors de la rencontre du 'Z'. Le nombre de combinaison est monstrueux. Il faut un temps supérieur à l'âge de l'univers pour analyser une chaîne suffisamment grande.

```

(a)...(a)(a)(a)(a)(a)Z
(a)...(a)(a)(a)(aa)Z
(a)...(a)(a)(aa)(a)Z
(a)...(a)(aa)(a)(a)Z
(a)...(a)(aa)(aa)Z
(a)...(aa)(a)(a)(a)Z
(a)...(aa)(a)(aa)Z
(a)...(aa)(aa)(a)Z
(a)...

```

## 2.1 Comment empêcher l'attaque d'expression régulière ?

Comment supprimer ce risque ? Utilisez de préférence un moteur d'expression régulière de type DFA. Malheureusement, ils sont généralement moins riches en fonctionnalités. Sinon, il faut analyser finement les expressions régulières utilisées pour éviter au maximum les retours arrière lors de l'analyse d'une chaîne. Il ne faut pas permettre à un utilisateur de saisir une expression régulière dans l'application. Si cela est vraiment nécessaire, les recherches de chaînes doivent s'exécuter dans une tâche spécifique, de priorité plus faible que les tâches du serveur. Un timeout s'occupera de tuer le processus s'il dure trop longtemps. L'extrait de la classe `SecureMatcher` indique comment proposer une protection pour l'utilisation des expressions régulières de Java.

```
public class SecureMatcher
{
    private final Matcher matcher_;
    private final long defaultTimeout_;

    private abstract class BooleanAsync implements Runnable
    {
        boolean result;
    }

    private boolean startBooleanAsync(BooleanAsync async, long timeout)
        throws InterruptedException
    {
        Thread thread = new Thread(async);
        thread.setDaemon(true);
        thread.setPriority(Thread.NORM_PRIORITY / 2);
        thread.start();
        try
        {
            thread.join(timeout);
        }
        finally
        {
            if (thread.isAlive())
            {
                thread.interrupt();
                throw new InterruptedException();
            }
        }
        return async.result;
    }

    public boolean find(long timeout) throws InterruptedException
    {
        return startBooleanAsync(
            new BooleanAsync()
            {
                public void run()
                {
                    result = matcher_.find();
                }
            }, timeout);
    }
    ...
}
```

Il faut encapsuler une instance `Matcher` par la version sécurisés. Les méthodes proposées simulent les méthodes originales. Elles acceptent éventuellement une valeur de timeout et retournent une exception en cas de timeout.

```
public static void main(String[] args)
{
    Pattern p = Pattern.compile("(a|aa)*B");
    try
    {
        SecureMatcher sm = new SecureMatcher(p
            .matcher("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaC"));

        System.out.println(sm.matches(1000) ? "match" : "not match");
    }
    catch (InterruptedException e)
    {
        System.err.println("TIMEOUT !");
    }
}
```

Une version complète de cette classes est présente sur mon site : [www.philippe.prados.name](http://www.philippe.prados.name) .

Les outils de détection d'intrusion permettent généralement à l'utilisateur de saisir des expressions régulières pour rechercher des modèles d'attaques. Les administrateurs doivent être très prudents dans leurs rédactions. Sinon, un pirate peut faire tomber un site si l'outil est placé en façade ou bien le neutraliser s'il est passif et écoute les communications.

Attention également aux frameworks qui exploitent les mêmes expressions régulières sur le client et le serveur. Par exemple, un framework qui génère des scripts permettant de qualifier les champs des formulaires peut révéler les expressions utilisées sur le serveur. Un pirate analysant la page peut découvrir des expressions exponentielles et générer des valeurs entraînant un déni de service.

### 3. AUTRES ATTAQUES

Ces attaques peuvent s'effectuer sur différentes technologies, lorsque que les algorithmes sont quadratiques ou exponentiel, ou qu'il existe des données permettant d'effectuer des boucles sans fin.

Des fichiers XML peuvent inclure des ressources sans fin, provoquant la saturation de la mémoire.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE hack [
<!ENTITY include SYSTEM "file:///dev/random">
]>
<hack>
&include;
</hack>
```

Des scripts XSL peuvent partir dans des boucles infinies.

```
<xsl:template match="foo">
  <xsl:apply-templates select="."/>
</xsl:template>
```

Des fichiers compressés peuvent avoir une taille raisonnable, et produire des fichiers très volumineux lors de la décompression. Par exemple, une boucle peut compresser une suite de zéro très longue. Lorsque le programme décompresse le fichier, pour y chercher un virus par exemple, il arrive qu'il explose ou que le disque sature.

Certaines versions du vérificateur de byte code de java sont également vulnérable à des classes judicieusement formées. La machine virtuelle peut partir en vrille, avant le lancement d'une classe, à cause de l'algorithme de vérification des classes. L'archive d'attaque est petite, car les classes sont très redondantes.

Des vulnérabilités ont été exploitées sur SSH, utilisant la difficulté de vérification des signatures DSA et la génération des clefs RSA. Des données aléatoires sont envoyées à la place de données à décrypter. Le serveur doit consommer une quantité importante de CPU pour se rendre compte de la supercherie. Une fois les données décryptées, il s'aperçoit qu'elles ne sont pas exploitables.

Résister à ces attaques algorithmiques est très difficile. Depuis les années 70, pratiquement personne ne s'est préoccupé de ces vulnérabilités. Les algorithmes proposés dans tous les langages sont optimisés pour une utilisation classique, mais sont également sensibles à ces attaques. Les applications sont de plus en plus exposées sur Internet. Les pirates peuvent alors analyser les outils et les langages utilisés sur le serveur, deviner où les tables de hash, les expressions régulières ou d'autres algorithmes identiques sont utilisées, et exploiter ainsi les vulnérabilité inhérente des langages.

Philippe PRADOS

IBM : Sécurité et technologies GRID

[press@philippe.prados.name](mailto:press@philippe.prados.name)

<http://www.hut.fi/~mkousa/ssh/ssh-dos.html>

[http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach\\_UsenixSec2003/](http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003/)

<http://hotwired.wired.com/packet/garfinkel/96/45/geek.html>

<http://www.cs.rice.edu/~scrosby/hash/slides/USENIX-RegexpWIP.2.ppt>

<http://www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-03-23.pdf>