Séminaire de détection des intrusions
Le déni de service (DoS)

Boniver Christophe
Herbiet Laurence
Joseph Benoit
Seronveaux Xavier

Le Denial of Service (DoS)

Les attaques par Denial Of service (souvent abrégé DoS, en français Déni de service) consistent à paralyser temporairement (rendre inactif pendant un temps donné) des serveurs afin qu'ils ne puissent être utilisés et consultés. Elles sont un fléau touchant tout serveurs (Lan, Wan...) mais aussi tous particuliers reliés à l'internet via les protocoles de la suite TCP/IP. Le but d'une telle attaque n'est pas de récupérer ou d'altérer des données, mais de nuire à des sociétés dont l'activité repose sur un système d'information en l'empêchant de fonctionner.

Ces attaques n'exploitent pas les failles d'un système d'exploitation particulier, mais celles de l'architecture TCP/IP. Les attaques par deni de service consistent souvent en un envoi de paquets IP en quantité très importante, ce qui a pour cause la saturation de la machine victime, qui ne peut plus assurer les services réseaux qu'elle propose (d'où le terme déni de service).

Sous-sections

Effets - Causes

Le déni de service se distingue par différents effets :

- Manque de ressources
- Verrouillage de fichiers ouverts
- Surcharge de ressources
- Inondation (Flooding) de reséaux
- Disponibilité des ressources
- Crash de serveur web

Ces différents effets sont la conséquence de :

- l'épuisement de processus, de mémoire disque ou de mémoire virtuelle
- le verrouillage de ressources
- l'exploitation de failles sur les systèmes (buffer overflow, mauvaises procédures de validations, ...)
- l'exploitation de failles sur les applications (failles dans les browsers, les outils mails, ...)
- les défauts de la pile IP (fragmentation, SYN flooding, ...)
- la saturation de la bande passante (ICMP flooding, requetes HTTP, ...)
- les interférences entre LAN (spoof des réponses DHCP, DNS et WINS, détournements (hijacking) TCP, désynchronisation de numéro de séquence TCP, ...
- les attaques sur le routage ou les serveurs DNS (spoof, ...)

Ce type d'attaque se traduit par différents facteurs :

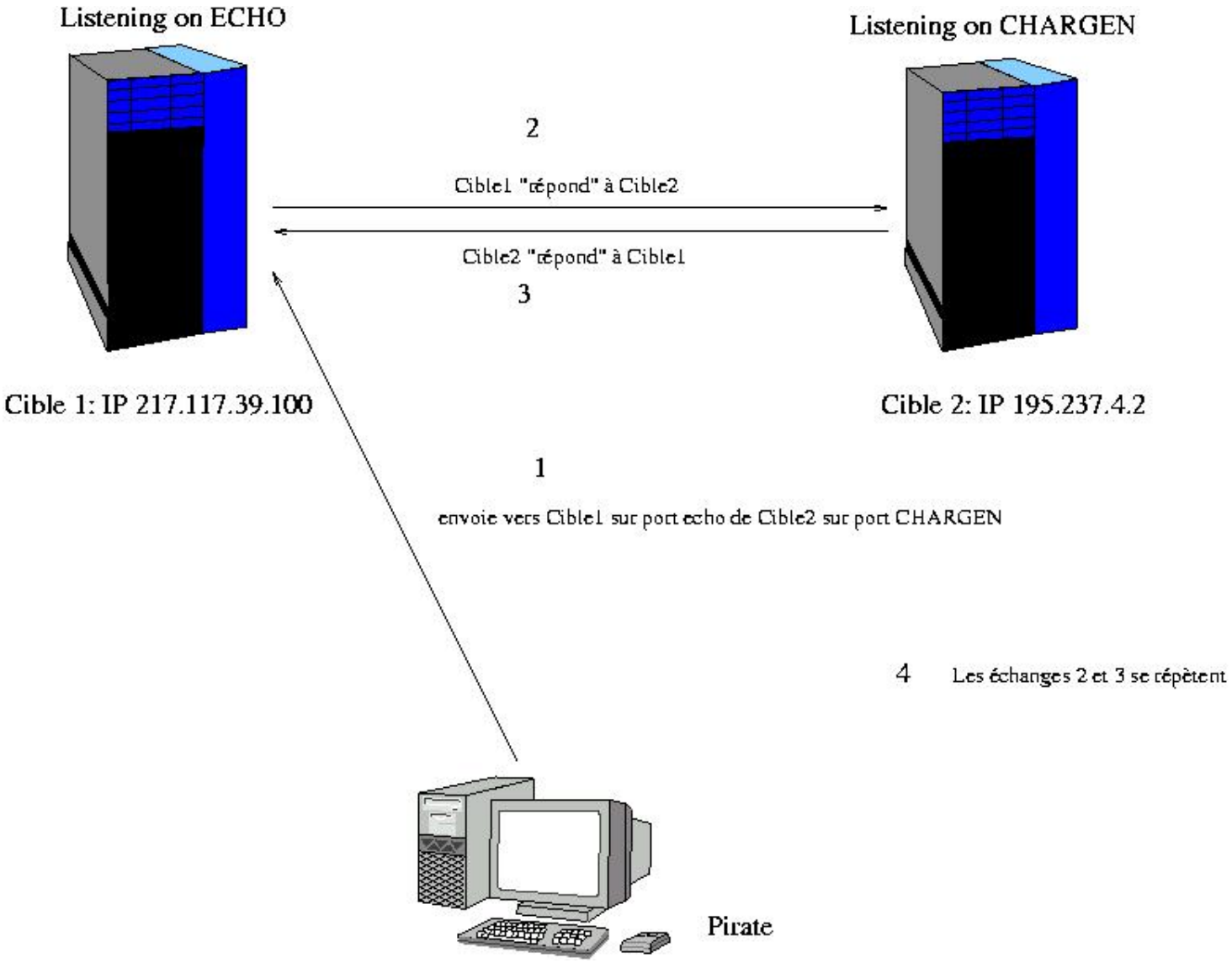
Performances :
se dégradent et passent sous certaines limites fixées;
Disponibilités :
Certaines applications/systèmes sont indisponibles;
Anomalies :
boucles de processus, manque de mémoire, apparition de fichiers inconnus, modification non prévue de fichiers,...

Pour se prémunir, il s'agit donc de vérifier les adresses sources suspectes (loopback, broadcast, adresses réservées, ...) et et les paquets anormaux (options inhabituelles, paquets trop grands, les Echos ICMP, ...)

Suivant : Exemple **Monter :** Le Denial of Service **Précédent :** Le Denial of Service

Exemple de DoS

Nous allons présenter un type d'attaque par refus de service.



Fonctionement de l'attaque
Le pirate envoi un message à la *Cible 1* sur le port ECHO en se faisant passer pour la *Cible 2*
Le message est du type:

Src : 195.237.4.2 CHARGEN
Dst : 217.117.39.100 ECHO

La *Cible 1* recevant ce message voit que c'est la *Cible 2* qui lui a envoyé le paquet.
En conséquence, le système envoie à son tour un paquet vers la *Cible 2* sur le port CHARGEN.
Le message est du type :

Src : 217.117.39.100 ECHO
Dst : 195.237.4.2 CHARGEN

La *Cible 2* reçoit ce paquet, CHARGEN génère un chaîne de caratère et la renvoit à la *Cible 1* sur le port ECHO.
On arrive ainsi à une boucle infinie. La machine qui "crashera" en premier est celle qui possède le moins de ressources.

L'attaque

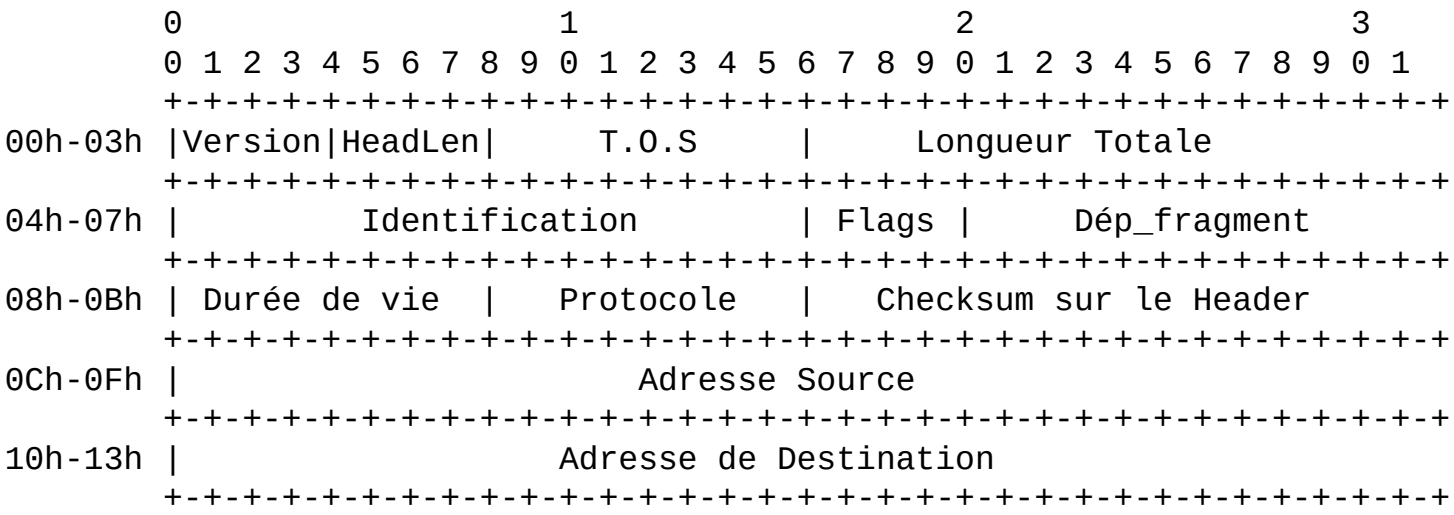
Type d'attaque :
Création d'un déni de service sur l'outil Snort (v 1.8.3) au moyen de paquets ICMP envoyés par **ping**

Commande utilisée :
Ping -c 1 -s 1 [Host]

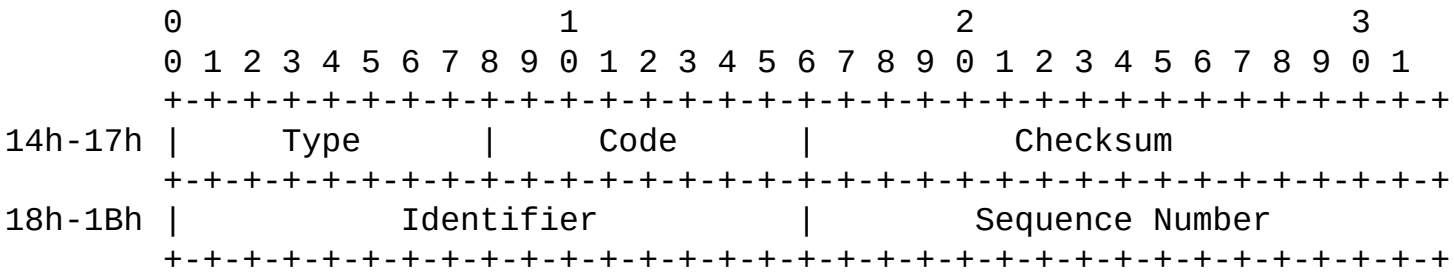
Sous-sections

Représentation de la structure du paquet ICMP envoyé

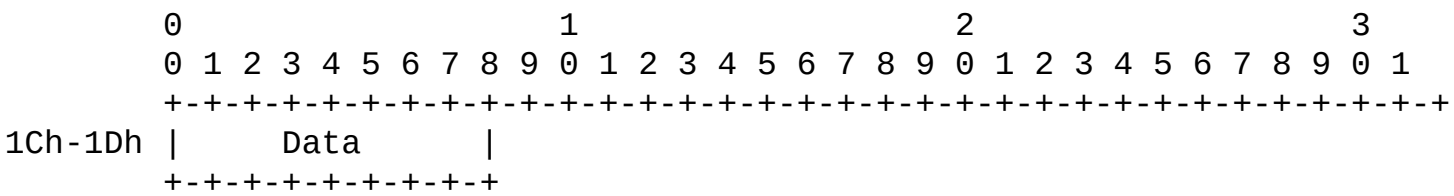
En-tête IP:



En-tête ICMP :



Données :



Taille totale du paquet: 29 bytes (28 bytes d'en-tête, 1 byte de données).

Résultat de la commande (obtenu par TCPDump)

Exemples avec "tcpdump" (pour deux pings successifs):

```
23:22:16.069711 localhost > localhost: icmp: echo request

4500 001d 01e2 0000 4001 7afc 7f00 0001 7f00 0001 0800 76fe 8101
0000 00

23:22:17.067666 localhost > localhost: icmp: echo request

4500 001d 01e4 0000 4001 7afa 7f00 0001 7f00 0001 0800 75fe 8101
0100 00
```

Lorsqu'on analyse ces paquets, on voit que les données correspondent bien aux champs décrits ci-dessus:

```
Offset:
             0    2    4    6    8    A    C    E

        0    4500 001d 01e2 0000 4001 7afc 7f00 0001

        1    7f00 0001 0800 76fe 8101 0000 00
```

En effet, on remarque le pattern "7f00 0001" (c'est à dire "127.0.0.1") à l'offset 0Ch et 10h, ce qui correspond bien aux champs d'adresse source et destination.

On remarque également la valeur 08h à l'offset 14h, qui correspond au type de message, c'est à dire: Echo Request (voir decode.h).

Enfin, on remarque à l'offset 1Ah que dans le premier paquet, la valeur est 00h et dans le second, la valeur est 01h, et on peut donc supposer qu'il s'agit du numéro de séquence.

Et le bug de Snort dans tout ca?

On constate donc qu'on a bien d'abord l'en-tête IP (20 bytes), puis l'en-tête ICMP (8 bytes), puis les données (1 bytes).

Si maintenant on regarde le code des fonctions "decode.h" et "decode.c" définie dans snort, (voir le zip avec les fichiers), on constate :

I. Lorsque Snort reçoit le paquet ICMP, il appelle d'abord la fonction "DecodeIP" en lui passant comme arguments:

- pkt => un pointeur vers le paquet reçu.
- len => longueur du paquet.
- p => un pointeur vers une structure facilitant le décodage.

II. Après plein de tests sur les checksum, fragments, ... il appelle la fonction "DecodeICMP" en lui passant comme args:

- pkt + hlen => pointeur vers le paquet ICMP (= adresse IP-Paquet + IP-Header-Len).
- ip-len => longueur du contenu du paquet IP, c'est a dire la longueur du paquet ICMP.
- p => la structure de décodage...

III. DecodeICMP refait des tests sur la longueur du paquet ICMP, checksum, ... Ensuite, il définit la longueur des données comme la taille du paquet ICMP moins la taille du Header, et définit un pointeur vers les données, c'est à dire, le pointeur du paquet ICMP plus la taille du Header.

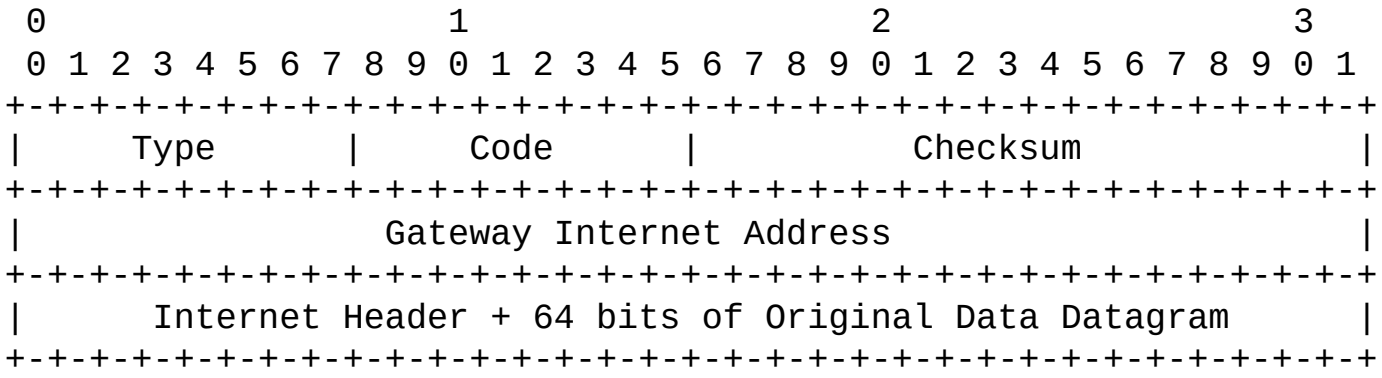
Ensuite survient le bug, ...

Il teste le type de message, et dans le cas d'un "echo", ou "echo reply", il décale de nouveau le pointeur vers les données 4 bytes plus loin (la structure "idseq" représente la ligne <18h-1Bh> de l'en-tête ci-dessus), et réduit la taille des données de 4 bytes.

Or dans l'exemple précédent, la taille du paquet ICMP etait de 9 bytes ==> si on saute le Header, il reste 1 byte, et si on saute encore 4 bytes, on est hors du buffer ==> Segmentation fault, ou core dumped, ...

En fait, il considère que dans certains cas, les 4 derniers bytes du header ICMP doivent figurer dans les données et dans d'autre cas, comme un "echo", non.

Par exemple, pour le paquet ICMP "Redirect Message", le header est constitué comme suit:



Si une attaque était réalisée avec des messages de ce type, il peut effectivement être intéressant d'avoir dans les données le champ "Gateway Internet Address".

Mais pour faire cela, il faut considérer que l'en-tête ICMP fait 4 bytes, (pour prendre en compte les 4 suivants dans les données) et dans le cas d'un "echo", il faut sauter les 4 bytes suivants. Cela revient tout simplement à changer la taille par défaut du Header ICMP à 4 bytes, et c'est bien ce que fait le patch proposé par Martin Roesch et implémenté dans la nouvelle version de Snort.

Suivant: Et la signature fut... **Monter:** Le Denial of Service **Précédent:** Résultat de la commande

Et la signature fut...

Lorsque Snort décode une trame ICMP de n (<4) bytes, il indique une payload supérieure à 65535-n bytes. Cette valeur peut-être obtenue dans les règles à l'aide de dsize.

La règle verifira si on a bien une trame ICMP de type ECHO REQUEST (type=8) et que la payload est supérieure à 65531 bytes.

- **alert icmp $EXT_NET any -> $HOME_NET any (msg:"ICMP less four bytes";itype:8;dsize:>655531;)**

- **alert icmp $EXT_NET any -> $HOME_NET any (msg:"ICMP less four bytes";itype:8;dsize:>655531;)**

Maintenant, pour ce qui est de détecter cette attaque dans un environement non vulnérable, il suffit de vérifier si la payload est inférieure à 4 bytes :

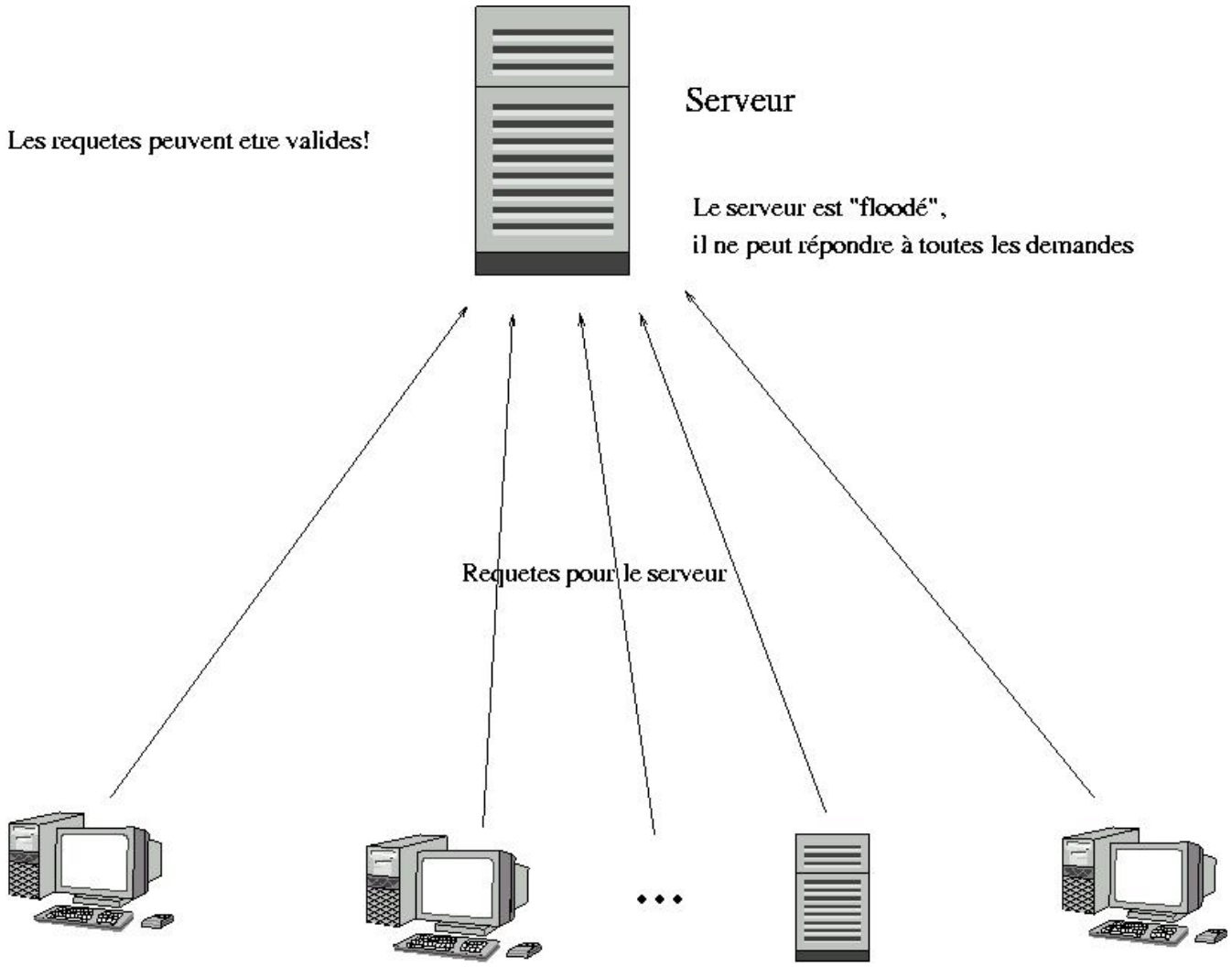
- **alert icmp $EXT_NET any -> $HOME_NET any (msg:"ICMP less four bytes";itype:8;dsize:<4;)**

Le Distributed Denial of Service (DDoS)

Sous-sections

Principe

Le DoS distribué à les mêmes effets que le DoS traditionnel excepté qu'ici ce n'est plus une seule machine qui attaque les autres mais plutôt une multitude de machines zombies contrôlées par un maître unique, ou encore une multitude de machine groupées (organisations), qui s'attaque à une cible.



Bien souvent le maître et ses esclaves communiquent par le biais des protocoles ICMP, TCP, SSH, Telnet, UDP, etc. et lancent des commandes qui ne requièrent pas de validations particulières (droits, mot de passe, authentification, ...). Ces commandes sont identiques à celle des Dos, à savoir, les inondations UDP, TCP, ICMP, SYN et les attaques de types smurf (broadcast dirigé). Mais également les attaques à l'aide de paquets malformés.

Cependant, il subsiste de nombreuses attaques DDoS qu'on n'arrive pas à contrer. C'est dû au fait qu'il est difficile de tracer les communications jusqu'à leur origine, de couper toutes les communications qui arrivent car elles proviennent aussi bien de personnes fiables que non fiables, de savoir que les machines zombies ont été attaquées au préalable.

Parmi les attaques DDoS très populaires, on connait l'attaque sur les sites internets Yahoo, CNN et EBay qui ont subit une inondation de leur réseau.

Suivant: L'attaque sur le World **Monter:** Le Denial of Service **Précédent:** Le Denial of Service

L'attaque sur le World Economic Forum (WEF)

Cette attaque a eu lieu au début du mois de février 2002 et elle se distingue par les éléments suivants :

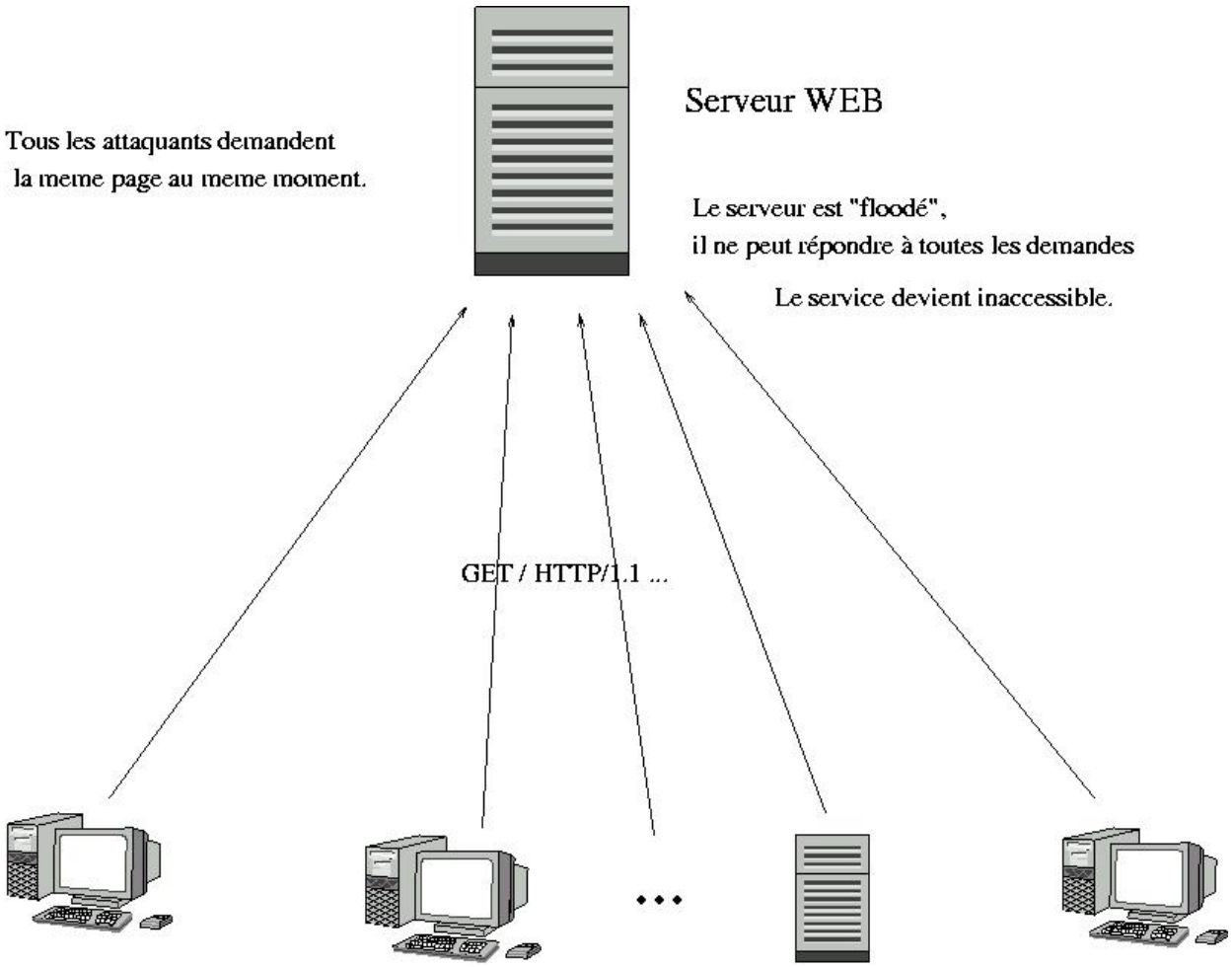
Contexte

Cette attaque a été lancée par une organisation hacktiviste. Leur principe est d'instaurer un 'sit-in' virtuel sur un site, c'est-à-dire d'en interdire l'accès d'une manière ou d'une autre. Il s'agit ici d'une attaque lancée par l'organisation Electronic Civil Disobedience qui s'est déjà illustrée par l'attaque de nombreux sites à tendance néo-libéraliste dans les pays latins et américains. Elle s'est également illustrée dans l'attaque du site d'un grand distributeur américain de jouets et contre la Maison Blanche.

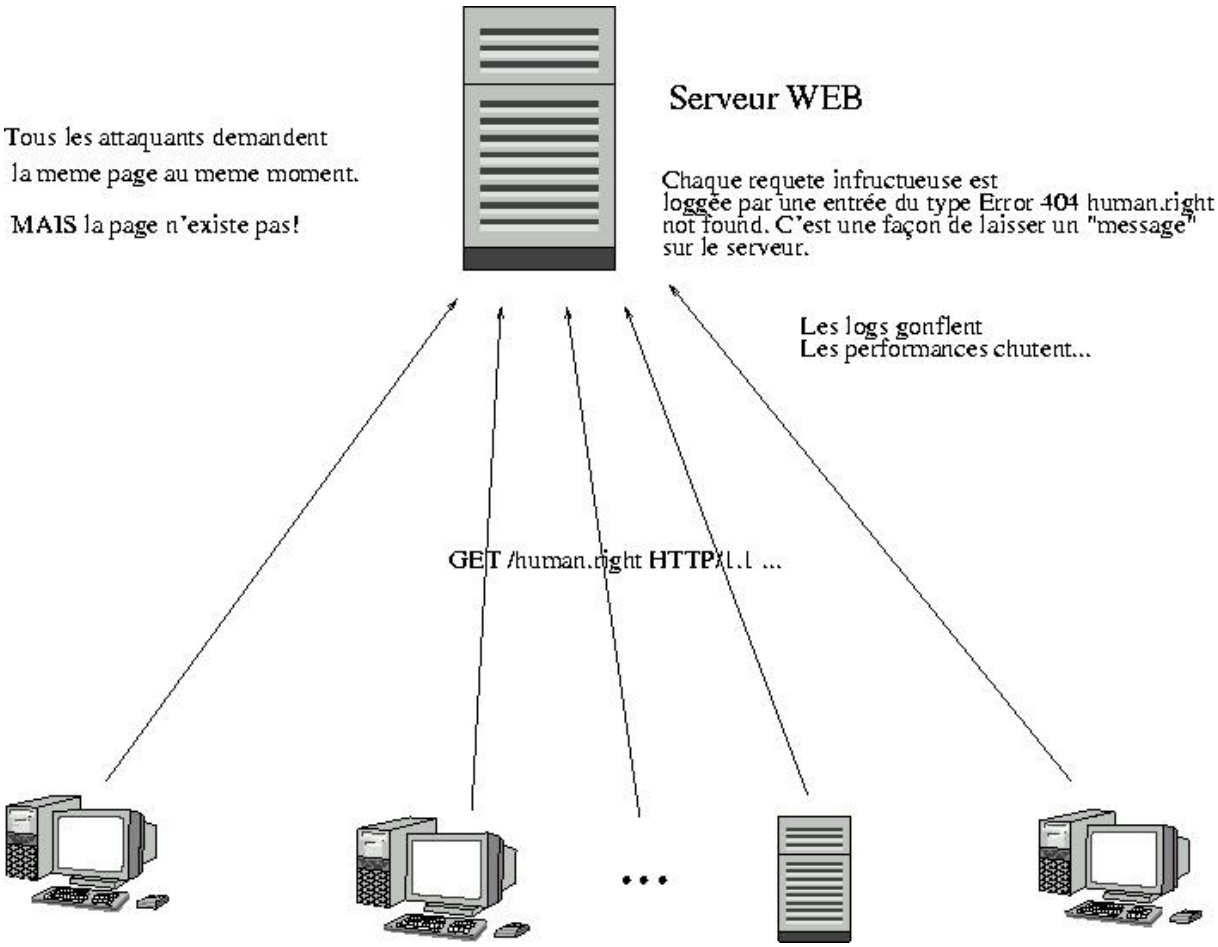
Principe de l'attaque

L'applet que nous avons pu récupérer permet d'attaquer en deux phases :

1. FLOOD : l'applet recharge une même page un grand nombre de fois par minutes



2. SPAM : l'applet permet également de laisser un message sur le serveur cible (via des requêtes erronées).
   Cela est possible car certains serveurs journalisent les requêtes et plus particulièrement celles retournant un code d'erreur 404 ("page not found").



L'applet a également tendance à multiplier le nombre de fenêtre de navigation afin de multiplier le nombre de demande. De plus, afin d'optimiser le nombre de requêtes sur le site, il est conseillé de neutraliser les javascripts et de refuser l'utilisation d'un cache pendant la navigation. Ainsi, l'afflux de requêtes a pour effet de saturer le serveur web qui finit par s'écrouler.

Mise en oeuvre de l'attaque

A chaque fois, ils procèdent de la même façon : une fois décidé le jour de l'attaque, ils préviennent toutes les personnes membres de leur réseau et les invitent à télécharger un utilitaire (une applet) et à la lancer à une date prévue. Dès que l'attaque a réussi, ils coupent tous les liens qui permettaient de charger l'utilitaire. C'est la raison pourquoi, faute d'avoir trouvé l'utilitaire confectionné dans le cadre du forum, nous en étudierons une version antérieure.

A chaque attaque, il ne s'agit jamais que d'une amélioration de leur prototype.

Les hacktivist n'ont donc même pas eu le problème de prendre le contrôle d'un parc de machines. L'attaque est lancée par l'applet qu'on active en chargeant une certaine page HTML

avec n'importe quel navigateur supportant Java. L'utilisateur arrive alors sur une page où on lui explique la marche à suivre et les options du navigateur à désactiver (javascript, cache, ...) pour que l'attaque soit optimale.

C'est également là que l'on peut rentrer un message personnel qui sera utilisé pour spammer le serveur.

Remarquons qu'il faut un grand nombre d'utilisateurs pour arriver à mettre un serveur sur les genoux. Plus le serveur disposera de ressources et plus il faudra d'utilisateurs. Dans le cas de l'attaque contre le WEF, ils étaient plus de 100.000.

Synchronisation des attaquants

La méthode utilisée ici est très simple : l'utilitaire n'est mis à disposition qu'aux alentours de la date prévue pour l'attaque et une notice indique quel jour l'attaque devra être lancée. Libre ensuite à qui le souhaite de s'y joindre.

Suivant: Essais-Erreurs **Monter:** Le DoS **Précédent:** Principe

La détection de l'attaque - Essais/Erreurs

Pour détecter cette attaque, nous avons envisagé deux directions différentes :

1. Utiliser des compteurs
2. Détecter l'applet

Plusieurs solutions se sont ainsi distinguées au fur et à mesure de nos différents essais. Certaines sont des améliorations de ce qui avait été proposé plus tôt dans nos réflexion, d'autres sont plutôt des remaniements complets.

Sous-sections

Détection par compteur

Notre attaque se base sur des inondations si bien que la première idée qui nous est venue à l'esprit était d'utiliser des compteurs.

Comme le premier symptôme de cette attaque était une grande quantité d'erreurs 404, nous avons regardé s'il n'était pas possible de faire quelque chose avec cela.

- Du coté du serveur, il s'agirait de compter le nombre d'erreurs 404 qui sont retournées par client pendant un laps de temps. Cette solution a l'avantage de réduire le nombre de paquets comptabilisés. De plus, si on est certain que tous les liens du serveur sont corrects, il est assez facile de fixer un seuil au-delà duquel il y a un risque de DDos.
- De même, du coté du client, il s'agirait de compter le nombre d'erreurs 404 reçues par utilisateurs. Ici aussi, l'évaluation du nombre d'erreurs acceptées est facile à évaluer.

Ce mode de détection fonctionne correctement. Cependant, cette solution peut rapidement être contournée si le hacker décide d'envoyer des requêtes vers des pages valides.

Dans ce cas nous avons du envisager une nouvelle solution qui était de compter le nombre de requêtes effectuées par URL. Cette solution est parfaitement réalisable mais très lourde de conséquences pour l'IDS.

- En effet, si on se trouve du coté du serveur et que celui-ci comporte beaucoup de ressources (un grand nombre d'url ou d'éléments référencés par les pages html), on aura un grand nombre de compteurs à gérer. De plus, il est difficile de savoir si ces requêtes proviennent d'un seul utilisateur ou de plusieurs. De même pour un site comme le WEF qui n'est ouvert qu'à l'occasion de certains événements, il n'est pas possible de savoir à l'avance combien de fois une page sera demandée pendant un certain intervalle.

  Donc quel que soit la situation, il est difficile de fixer un seuil au-delà duquel on peut affirmer que l'on est attaqué.
- Si on se trouve du coté d'un client et que l'organisation dispose d'un grand nombre de machines susceptibles d'envoyer des requêtes, logger toutes les urls contactées par ses utilisateurs peut se révéler lourd de conséquence pour l'IDS. En effet, le stockage et l'analyse de ceux-ci sont susceptibles de causer un DoS de l'ids lui-même.

Bien que cette solution nous permette de détecter un DDoS par requêtes HTTP d'une façon plus générale que la détection d'erreurs 404, elle n'est pas très performante. De plus, elle risque d' engendrer un DoS sur l'IDS lui-même ce qui n'est pas vraiment ce que nous souhaitons.

Afin de contourner ce problème, nous nous sommes demandé s'il ne fallait pas plutôt comptabiliser le nombre de requêtes effectuées (vers des fichiers .html, .php, .jsp, ...) pendant un certain laps de temps par un utilisateur (sur base des adresses IP)

- Du coté client, il s'agirait de compter le nombre de requêtes lancées par un utilisateur pendant un intervalle particulier. Sachant que les demandes sont temporisées (une requête toutes les 2 secondes dans le cas de notre applet), il doit être possible d'évaluer un nombre limite de requêtes effectuées. On peut dès lors limiter les intervalles de temps et les bornes supérieurs ce qui réduit la charge des logs par rapport à la solution précédente.
- Du coté serveur, il s'agirait de compter le nombre de fois qu'un client contacte le serveur pendant un certain laps de temps. Connaissant la temporisation de l'applet, on peut envisager de limiter l'intervalle de temps. Comme on distingue les utilisateurs, on n'a plus le problème de devoir évaluer le nombre de personnes qui risquent de contacter celui-ci. De plus, dès que l'un deux dépasse le nombre limite de requêtes permises pendant l'intervalle de temps, une alerte est lancée et le nombre de logs effectués est limité.

  Cependant, on conserve l'inconvénient de la gestion des compteurs.

Il est donc clair que la détection par compteur peut être très pénalisante si on ne paramètre pas bien les éléments à comptabiliser. Cependant, tant que les requêtes sont fausses, la première solution est très efficace.

Suivant: Détection sur l'applet **Monter:** Essais-Erreurs **Précédent:** Essais-Erreurs

Détection sur l'applet

La première solution que nous avons envisagée était de regarder si l'applet initialisait certains champs particuliers de l'en-tête http. Mais il semblerait que la requête soit parfaitement transparente et qu'aucun champ ne soit généré par l'applet.

Nous n'avons donc pas pu exploiter cette piste.

Nous avons alors tenté de nous fixer sur l'applet elle-même.
Tout d'abord, nous avons envisagé de détecter sa présence dans les pages HTML. En effet, il semblerait que d'évolution en évolution, l'applet conserve le même nom ou un nom fort semblable. Cette solution ne pourra être appliquée que du coté client.
Il s'agirait donc de détecter une balise <APPLET> et dans cette balise de retrouver des mots clés tels que "spam" ou "flood" que l'on retrouve fréquemment. Bien que très facile à détecter, cette solution s'est révélée très fragile car il suffit que l'on modifie le nom de l'applet et qu'on la recompile pour que la détection soit caduque.

La solution suivante que nous avons envisagée était de trouver dans le fichier ".class" de l'applet téléchargée le nom de l'applet ou certains mots suspects tels que "flood" ou "spam" que l'on retrouve souvent dans le code de celle-ci. Ce n'est pas très difficile à détecter car la signature de l'applet est stockée dans les 4 premiers bytes du fichier ".class". De plus, le nom de l'applet ainsi que les imports apparaissent en clair dans l'applet.

On remarque cependant assez vite que cette solution souffre des mêmes lacunes que la proposition précédente. De plus, elle nécessite un chargement à distance de l'applet.

Nous nous sommes alors penché sur l'origine de l'applet. Nous avons ainsi décidé que que toutes les applets provenant de serveurs non-fiables tels que hacktivist, nethack,... devraient être loggées. Pour réaliser cela, il faudra regarder les requêtes dont l'IP dest correspond à celles d'un de ces sites et dont l'uri contient ".class".

Solutions retenues et résultats obtenus

Sous-sections

La piste des compteurs

Nous avons tout d'abord décidé de comptabiliser les erreurs 404 comme expliqué ci-dessus. Cependant, Snort ne propose pas de compteur par défaut, il a donc fallu développer un préprocesseur dédicacé.

Notre préprocesseur a été conçu de façon à pouvoir s'adapter aussi bien à un client qu'à un serveur. C'est pourquoi, il nécessite certains paramètres lors de son initialisation :

Initialisation

Lorsque le préprocesseur est déclaré dans "snort.conf", il analyse les arguments qui lui sont passés et reconnait:

- *src*: s'il doit se fier à l'adresse IP source.
- *dest*: s'il doit se fier à l'adresse IP de destination.
- *to:[xx]* avec [xx] définissant combien de temps il garde en mémoire une IP inactive.
- *nb:[xx]* avec [xx] définissant le nombre d'erreurs tolérées.
- *ti:[xx]* avec [xx] définissant l'interval de temps durant lequel les [nb] erreurs sont tolérées.

La ligne de snort.conf ressemble donc à ceci:

preprocessor http_404: -src -to:60 -nb:10 -ti:60

Remarques :

i. Les options -src et -dest ne peuvent êtres utilisées en même temps;
ii. Pour détecter les floods côté client et surtout serveur, il est préférable d'utiliser -dest. L'option -src était en fait surtout utile pour tester plusieurs adresses IP avec un seule machine.
iii. [to] et [ti] sont exprimés en secondes.
iv. Les paramètres [nb] et [ti] sont des ordres de grandeurs. Etant donné la nature des compteurs, il est impossible de garantir que [nb] erreurs en [ti]+1 secondes ne seront pas alertées.

Détection

Le préprocesseur commence par vérifier qu'il s'agit d'un paquet de traffic TCP (donc que la session TCP est ouverte). Si ce n'est pas le cas, il quitte directement.

Il vérifie ensuite qu'il s'agit d'une réponse HTTP, c'est-à-dire que les 4 premiers bytes de données du paquet sont 'HTTP'. Etant donné que 4 'char' = 1 'int', il suffit de considérer que les données sont un tableau de 'int' et de vérifier que le premier est 'PTTH' (car la machine sur laquelle nous avons testé est little endian).

Le début d'un paquet de réponse HTTP ressemble toujours à ceci (car c'est le serveur qui l'envoie) :

HTTP/1.1 404 [...]

Si on le découpe en 'int', on obtient:

|HTTP| |/1.1| | 404| [...]

Il suffit donc pour détecter une erreur 404 de comparer le 3eme int à '404_' (c'est à dire '_404' inversé). La vérification qu'un paquet TCP est bien une erreur HTTP 404 est donc très rapide.

Remarque :
On ne peut pas optimiser la détection de cette manière si on teste les 'GET' sur les pages 'html' car dans ce cas, le 'GET' est envoyé par l'attaquant, et peut donc être 'gEt' si ca l'amuse.

Comptage

Une fois l'erreur détectée, il faut l'ajouter à un compteur pour ne lancer une alerte que quand un certain seuil est dépassé. De plus, pour que la détection soit efficace, il faut qu'il y ait un compteur par adresse IP.

Pour cela, le préprocesseur insère chaque compteur dans une liste doublement liée circulaire ordonnée par adresse IP croissante. La liste des champs d'une cellule de la liste est :

- *unsigned int IP_Addr;* // L'adresse IP
- *double Time;* // La date et l'heure de la dernière erreur 404.
- *float Count;* // Le compteur.
- *void *Prev;* // Un pointeur vers la cellule précédente.
- *void *Next;* // Un pointeur vers la cellule suivante.

Pour simplifier la gestion de la liste, on y insère par défaut une cellule avec l'adresse IP égale à 0 de sorte que la liste ne soit jamais vide.
Il y a aussi un pointeur global qui pointe vers la cellule courante dans la liste.

Lorsque l'erreur 404 est détectée, le préprocesseur parcours la liste depuis la cellule courante jusqu'à ce qu'il tombe sur la bonne adresse IP, ou qu'il dépasse l'endroit où elle aurait du se trouver (ce qui est faisable vu que la liste est ordonnée par IP croissante). Vu que la liste est doublement liée, on peut choisir le sens de parcours pour minimiser la distance entre l'adresse IP recherchée et celle se trouvant dans la cellule courante.

Lors de cette recherche, si le préprocesseur tombe sur une cellule dont l'adresse IP ne correspond pas et dont le champ 'Time' est inférieur à l'heure actuelle - [to], il retire la cellule de la liste (sauf s'il s'agit de l'adresse IP 0). Cela permet d'élaguer la liste pour ne garder en mémoire que les adresses IP actives, sans pour autant faire un tour complet de la liste à chaque fois.

Une fois la cellule trouvée (ou créée), le préprocesseur met à jour la valeur du compteur par la formule suivante :

$$Count=Count*exp(K1*(Now-Time))+K2;$$

Ensuite, il la compare à K3 et si elle plus grande, il lance une alerte et retire la cellule de la liste.

Les valeurs de K1,K2 et K3 sont calculées à partir des paramètres [nb] et [ti].
Le préprocesseur retire la cellule de la liste pour limiter le nombre d'alertes lancées. On pourrait toutefois envisager une autre stratégie, du style remettre le compteur à 0, ou désactiver cette adresse pendant un certain temps, ...

Remarque :
Le calcul de K1,K2 et K3 est basé sur le fait qu'on peut calculer K1 et K2 pour que C ne dépasse jamais une certaine constante si (Now-Time) est constant. En calculant ces valeurs avec Cmax=[ti] et (Now-Time) = [ti]/[nb], on arrive a faire un compteur qui ne dépassera [ti] que si le nombre d'erreurs moyen sur un temps [ti] est plus grand que [nb].
Le problème, c'est que si le nombre d'erreurs moyen est légèrement supérieur à [nb], le nombre d'erreurs nécessaires pour dépasser [ti] peut être très grand. C'est pour cela que pour calculer K1 et K2, on utilise un certain pourcentage de [nb].

Suivant: La piste des applets **Monter:** Solutions et résultats

La piste des applets

Nous avons finalement trouvé trois façons de détecter l'applet. La première solution est de détecter la balise applet et le nom de l'applet. Grâce au mot clé '*nocase;*', on rend le nom et les balises insensibles à la casse.

La deuxième consiste à rechercher les paramètres destinés à l'applet. Dans notre cas, il s'agit de *targetUrl* et de *evade*.

La troisième intercepte les requêtes sortantes contenant la chaîne ".class" à destination de certains hôtes non sûrs. Dans notre cas, l'hôte est knuth.

Limites :
La première règle est assez fragile car il suffit de changer le nom de la classe et de recompiler pour que l'applet ne soit plus détectée.

La deuxième fait intervenir des détails de l'implémentation de l'applet. Pour pouvoir changer les paramètres dans la page html, il faudrait changer le code qui récupère les paramètres ce qui est un rien moins simple que de changer le nom.

La troisième méthode est beaucoup plus générale car elle s'attaque aux .class en provenance d'un serveur bien particulier. Il est évident qu'il faudra une règle par serveur que l'on veut surveiller.

Les Règles :

Les règles sont:

- ```
  alert tcp any 80 -> $HOME_NET any (msg:"Flooding Applet"; content:"<applet";
  nocase; content:"zapsfloodnetpublic1.class"; nocase;)
  ```

- ```
  alert tcp any 80 -> $HOME_NET any (msg:"Flooding param"; content: " targetUrl ";
   nocase; content:"evade";  nocase;)
  ```

- ```
  alert tcp $HOME_NET any -> 192.168.1.44/32 80 (msg:"flooding server"; uricontent:
  ".class"; nocase;)
  ```

  ```

Table des matières