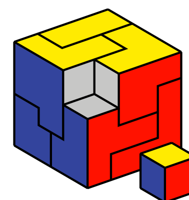
Prologin 2020 

Concours National d'Informatique

Correction des challenges

Correction des challenges

Gabriel Duque

16 janvier 2020

Table des matières

1 Strace-me **2**

2 LSB **5**

3 Call me maybe **9**

3.1 Une approche automatisée 9

3.2 Une approche plus amusante 11

1 Strace-me

Énoncé Voici le code qui a été compilé pour générer le binaire ‘chal1.elf‘.

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define FLAG_SIZE    14
#define FILENAME     "toto.txt"
#define MIN_FILESIZE 42

static char flag[FLAG_SIZE] = {
    0x77,
    0xff,
    0xfb,
    0x5d,
    0x64,
    0x2a,
    0x81,
    0xce,
    0xe5,
    0xb2,
    0xe7,
    0x66,
    0x50,
    0xff
};

static char mask[FLAG_SIZE] = {
    0x42,
    0xab,
    0x89,
    0x69,
    0x07,
    0x19,
    0xde,
    0xff,
    0x96,
    0xed,
    0x81,
    0x13,
    0x3e,
    0xff
};

static void print_flag(void)
{
    for (int i = 0; i < FLAG_SIZE; ++i)
        flag[i] ^= mask[i];
```

```c
    printf("PROLOGIN{%s}\n", flag);
}

int main(void)
{
    int fd;
    char buff[MIN_FILESIZE];

    /* Force them to create a file named toto.txt */
    if ((fd = open(FILENAME, O_RDONLY)) == -1)
        exit(EXIT_FAILURE);

    /* It must be at least MIN_FILESIZE characters long */
    if (read(fd, buff, MIN_FILESIZE) != MIN_FILESIZE) {
        close(fd);
        exit(EXIT_FAILURE);
    }

    close(fd);
    print_flag();

    return 0;
}
```

Résolution En l'état, le programme qui s'exécute ne semble rien faire de particulier à part quitter en renvoyant un 1. Afin de mieux comprendre son comportement, nous pouvons utiliser la commande 'strace'.

'strace' est un outil de débogage sous Linux pour surveiller les appels système utilisés par un programme, et tous les signaux qu'il reçoit.

Lorsque vous lancez le binaire avec 'strace' pour l'analyser, vous devriez obtenir une sortie ressemblant à ceci :

```
$ strace ./chal1.elf
execve("./chal1.elf", ["./chal1.elf"], 0x7ffca149f3b0 /* 39 vars */) = 0
open("/proc/self/exe", O_RDONLY)        = 3
[...]
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd2b714350) = -1 EINVAL (Invalid argument)
brk(NULL)                               = 0x15c1000
brk(0x15c21c0)                          = 0x15c21c0
arch_prctl(ARCH_SET_FS, 0x15c1880)      = 0
uname({sysname="Linux", nodename="ako", ...}) = 0
readlink("/proc/self/exe", "/home/zuh0/src/ctf/qcm2020/1-str"..., 4096) = 48
brk(0x15e31c0)                          = 0x15e31c0
brk(0x15e4000)                          = 0x15e4000
openat(AT_FDCWD, "toto.txt", O_RDONLY)  = -1 ENOENT (No such file or directory)
exit_group(1)                           = ?
+++ exited with 1 +++
```

Ca fait beaucoup d'informations mais elles ne sont pas toutes importantes pour notre challenge.

Dans ce genre de situation, il faut essayer de détecter les anomalies dans la sortie de 'strace'. On peut voir que le dernier appel système à avoir été un échec avant de quitter le programme était 'openat' qui a échoué avec une erreur 'ENOENT' car on a essayé d'ouvrir un fichier qui n'existait pas : 'toto.txt'.

Nous pouvons donc créer ce fichier en utlisant la commande 'touch' :

```
$ touch toto.txt
```

Avant de relancer ‘strace‘ sur notre programme :

```
$ strace ./chal1.elf
execve("./chal1.elf", ["./chal1.elf"], 0x7ffca149f3b0 /* 39 vars */) = 0
open("/proc/self/exe", O_RDONLY)        = 3
[...]
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd2b714350) = -1 EINVAL (Invalid argument)
brk(NULL)                               = 0x15c1000
brk(0x15c21c0)                          = 0x15c21c0
arch_prctl(ARCH_SET_FS, 0x15c1880)      = 0
uname({sysname="Linux", nodename="ako", ...}) = 0
readlink("/proc/self/exe", "/home/zuh0/src/ctf/qcm2020/1-str"..., 4096) = 48
brk(0x15e31c0)                          = 0x15e31c0
brk(0x15e4000)                          = 0x15e4000
openat(AT_FDCWD, "toto.txt", O_RDONLY)  = 3
read(3, "", 42)                         = 0
close(3)                                = 0
exit_group(1)                           = ?
+++ exited with 1 +++
```

Le programme va plus loin cette fois-ci ! Il réussit à ouvrir le fichier ‘toto.txt‘ et essaye de lire 42 caractères dans un tampon depuis le fichier ouvert.

Remplissons notre fichier ‘toto.txt‘ d’au moins 42 caractères.

```
$ python -c 'print("A" * 42)' > toto.txt
```

Maintenant que tout semble bon, relançons le binaire du CTF tout seul.

```
$ ./chal1.elf
PROLOGIN{5Tr4c3_1s_fun}
```

Nous obtenons le flag !

Misc Certains d’entre vous ont remarqué que le binaire était un peu "bizarre" et ont eu l’impression qu’il était auto-modifiant. Ce n’est pas exactement le cas mais si ça vous intéresse il avait été "packé" avec UPX.

Si vous avez essayé de le "unpacker" avec UPX et que cela n’a pas fonctionné, c’est parce que nous avons remplacé le "magic number" inséré par UPX avec ‘sed‘.

Correction proposée par Gabriel Duque (zuh0).

2 LSB

Énoncé L'image ne semble cacher aucune information visible à l'œil nu. Le flag a en fait été dissimulé en utilisant la méthode Least Significant Bit (LSB). Voici une implémentation en Python3 de l'algorithme qui a été utilisé pour dissimuler le flag dans l'image.

```
def embed(f, in_img, out_img):
    """Encode file in image using the Least Significant Bit method
    f as file to encode
    in_img as name of input image
    out_img as name of result image
    """
    with open(f, 'rb') as f_hndl:
        i_hndl = Image.open(in_img)
        o_hndl = Image.new('RGB', i_hndl.size)

        channels = []

        # Read all rgb channels
        for x in range(i_hndl.width):
            for y in range(i_hndl.height):
                channels.extend(list(i_hndl.getpixel((x, y))))

        # Embed text in channels
        index = 0
        for byte in f_hndl.read():
            for i in range(8):
                bit = bool((byte >> i) & 1)

                if bit:
                    channels[index] |= 1
                else:
                    channels[index] &= ~1

                index += 1

        # Write output image
        for x in range(i_hndl.width):
            for y in range(i_hndl.height):
                index = 3 * (x * i_hndl.height + y)
                rgb = tuple(channels[index:index + 3])
                o_hndl.putpixel((x, y), rgb)

        o_hndl.save(out_img, 'BMP')
```

La sténographie LSB consiste à dissimuler chaque bit de notre texte initial sur le bit de poids faible (représentant l'unité) des couleurs primaires (RGB) dans les bytes représentants nos pixels. L'avantage de cette technique est donc d'être invisible à l'œil nu, car modifier le bit de poids faible n'aura que peu d'incidence sur la valeur du byte (+1, -1 ou pas de changement par rapport à la valeur initiale du byte).

Résolution Pour extraire le flag de cette image, la méthode consiste à récupérer le bit de poids faible des couleurs de chaque pixel et de les reconstruire en texte. Différentes variantes de sténographie existent, parmi lesquelles ne dissimuler que sur certaines couleurs primaires du pixel ou dissimuler en

partant des derniers pixels. Dans notre cas, le texte est encodé sur RGB mais à la verticale. (cf code de dissimulation)

Voici un exemple de code pour extraire notre flag.

```
/* Extract the LSB hidden file */

#include <err.h>
#include <stdint.h>
#include <stdlib.h>

#include <SDL.h>
#include <SDL_image.h>

/* Load the file */
static SDL_Surface* load_image(char *path)
{
    SDL_Surface *img;
    img = IMG_Load(path);
    if (!img)
        errx(1, "can't load %s: %s", path, IMG_GetError());
    return img;
}

/* Get pointer to pixel data */
static inline uint8_t* pixelref(SDL_Surface *surf, uint32_t x, uint32_t y)
{
    int32_t bpp = surf->format->BytesPerPixel;
    return (uint8_t *)surf->pixels + y * surf->pitch + x * bpp;
}

/* Get pixel value */
static uint32_t getpixel(SDL_Surface *surface, unsigned x, unsigned y)
{
    uint8_t *p = pixelref(surface, x, y);
    switch (surface->format->BytesPerPixel) {
        case 1:
            return *p;
        case 2:
            return *(uint16_t *)p;
        case 3:
            if (SDL_BYTEORDER == SDL_BIG_ENDIAN)
                return p[0] << 16u | p[1] << 8u | p[2];
            else
                return p[0] | p[1] << 8u | p[2] << 16u;
        case 4:
            return *(uint32_t *)p;
    }
    return 0;
}

/* Actually fills the buffer with the flag and some noise */
static void extract_flag(SDL_Surface *img, uint8_t *buff)
{
    uint8_t r;
```

```c
    uint8_t g;
    uint8_t b;
    uint8_t byte_idx;
    uint32_t pixel;

    byte_idx = 0;

    /* Iterate over each pixel */
    for (int32_t x = 0; x < img->w; ++x) {
        for (int32_t y = 0; y < img->h; ++y) {
            pixel = getpixel(img, x, y);
            SDL_GetRGB(pixel, img->format, &r, &g, &b);

            /*
             * Get bit in r, g, and b.
             * Flag was calloced so no need to & 0, | 0 will not set bit
             */
            *buff |= (r & 0x1) << byte_idx;
            /* Should we pass to the next byte in buffer */
            if (byte_idx++ == 7) {
                byte_idx = 0;
                buff++;
            }

            *buff |= (g & 0x1) << byte_idx;
            if (byte_idx++ == 7) {
                byte_idx = 0;
                buff++;
            }

            *buff |= (b & 0x1) << byte_idx;
            if (byte_idx++ == 7) {
                byte_idx = 0;
                buff++;
            }
        }
    }

}

/* Main entrypoint */
int main(int argc, char **argv)
{
    uint8_t *flag;
    SDL_Surface *img;
    int32_t buff_size;

    if (argc != 2)
        errx(1, "usage: %s <filename>", argv[0]);

    img = load_image(argv[1]);

    /*
```

```
     * Get total size of buffer (total number of pixels * 3 / 8) + 2
     * + 2 to take into account the last bits and a null-terminator
     */
    buff_size = (img->h * img->w * 3) / 8 + 2;

    if ((flag = calloc(buff_size, sizeof(uint8_t))) == NULL)
        errx(1, "could not allocate memory");

    /* Call the extraction function */
    extract_flag(img, flag);

    /*
     * We know the flag is just the first line so we will print just that
     * but we could have printed the whole buffer resulting in a line with the
     * flag followed by lots of noise.
     */
    for (size_t i = 0; flag[i]; ++i) {
        if (flag[i] == '\n') {
            flag[i] = '\0';
            break;
        }
    }

    /* Print the flag */
    printf("%s\n", flag);

    /* Cleanup */
    free(flag);
    SDL_FreeSurface(img);
}
```

Il ne reste plus qu'à exécuter notre programme pour extraire notre flag.

```
$ ./extract chal2.bmp
PROLOGIN{5t3G4n0gr4phy_1s_b0r1n9}
```

Correction proposée par Gabriel Duque (zuh0) & Augustin Thiercelin (n1tsu).

3 Call me maybe

Énoncé Lorsque l'on appelle le numéro fourni dans le fichier audio, nous nous retrouvons dans un environnement où l'on a à notre disposition une pile et une série d'opérations que l'on peut utiliser pour modifier son état.

La liste des opérations disponibles ainsi que des exemples sont disponibles dans le manuel.

Le but final était d'écrire la chaîne de caractères 'AlLoCtF2020' dans la pile.

Résolution Nous allons voir deux solutions pour résoudre le problème.

La première solution est générique et fournit une suite de touches pour n'importe quelle chaîne de caractères.

La seconde solution est celle qui a été réalisée à la main par nos soins avant de publier le challenge.

3.1 Une approche automatisée

Nous pouvons remarquer que tout nombre écrit en binaire peut être vu comme plusieurs nombres écrits en binaire avec un seul bit à 1 auxquels on a appliqué l'opérateur binaire "OR".

Par exemple :

```
 42 =  0b101010

    =  0b100000
    OR 0b001000
    OR 0b000010
```

En partant de ce postulat, il ne nous reste plus qu'à trouver un moyen de générer les nombres avec un seul bit à 1.

Pour ce faire, on a besoin de seulement deux instructions : PUSH et SHL. En effet, pour set le bit d'index 'i' (les indices commencent à 0), il nous suffit de PUSH un 1 sur la pile et de lui appliquer un SHL 'i' fois.

Par exemple, pour obtenir 8 il faut PUSH 1 puis appliquer un SHL 3 fois.

En effet :

```
  8 =  0b1000

    =  0b0100 SHL 1
    =  0b0010 SHL 1 SHL 1
    =  0b0001 SHL 1 SHL 1 SHL 1
    =       1 SHL 1 SHL 1 SHL 1
```

Ainsi, pour obtenir un nombre ayant le bit d'index 'i' à 1, il suffit de PUSH un 1 'i + 1' fois puis SHL 'i' fois.

Une fois que les différents masques (nombres avec un seul bit à 1) sont sur la pile, il suffit de leur appliquer 'n - 1' fois l'opérateur OR.

Voici un script en Python3 qui prend en entrée une chaîne de caractères et qui génère une des séquences possibles pour écrire cette chaîne sur la pile.

```python
#!/usr/bin/env python3

import sys


def get_seq(c: chr) -> str:
    '''Generate the sequence for a specific character.

    Each character can be seen as the OR between multiple numbers having
    only one bit set. We can use this to generate each single-bit-set number
    with only PUSH and SHL and then OR then all together.

    5 is a PUSH 1
    3 is the SHL operator
    1 is the OR operator
    '''
    seq = str()
    cnt = 0
    ascii_code = ord(c)

    for i in range(8):
        if ascii_code & (1 << i):
            seq += '5' * (i + 1) + '3' * i
            cnt += 1
    seq += '1' * (cnt - 1)
    return seq

def main() -> None:
    '''Iterate over string backwards (top of stack must be first letter)
    and generate dial-tone sequences for each character of input string.
    '''
    if len(sys.argv) != 2:
        sys.stderr.write(f'usage: {sys.argv[0]} <str>\n')
        sys.exit(1)

    seq = str()
    for c in sys.argv[1][::-1]:
        seq += get_seq(c)

    print(seq)

if __name__ == '__main__':
    main()
```

On peut ensuite utiliser ce script pour générer la séquence pour obtenir notre flag :

```
$ ./chal3.py AlLoCtF2020
555553333555555333331553555553333555555333331155555333355555533333155355555553333
5555553333311553555335555555333333115553355555333355555533333555555533333311155
535555555333333115553555335555333555555333335555555333333111115553355553335555
5553333331155533555533355555533333555555533333311155555555333333331
```

Vous pouvez ensuite valider votre pile en appuyant sur la touche ‘#‘.

Après une version française de la magnifique chanson "Call me maybe" de Carly Rae Jepsen, le flag ‘PROLOGIN{@steriskm@sterace}‘ est dicté pour finir ce challenge.

3.2 Une approche plus amusante

Générer les séquences, c'est drôle, mais en faisant ça, on n'utilise que 3 des opérations disponibles.

Pour tester la machine à pile, nous avons écrit une solution qui utilise plus d'opérations (toutes sauf ROTL).

Nous n'allons pas rentrer dans les détails du fonctionnement de chacune des séquences mais nous allons toutes vous les donner.

On a donc :

```
A  l  L  o  C  t  F  2  0  2  0

41 6c 4c 6f 43 74 46 32 30 32 30

Solution complète:
    5553355351355555533335555333511355533553513555555333355553335113553555553333
    3555533351185535555533335555333555335111355355555333555333511855546555533153 2
    455358553553513155351553518553555351315535555553318

    A
    01000001:
        55355553318

    l
    01101100:
        553515535185535535131

    L
    01001100:
        553585535535131

    o
    01101111:
        5554655553315324

    C
    01000011:
        5535555333555335118

    t
    01110100:
        55355555333355553335553351113

    F
    01000110:
        553555553333555533335118

    2
    00110010:
        55555533335555333335113

    0
    00110000:
        55533553513
```

```
2
00110010:
    5555553333555533335113

0
00110000:
    55533553513
```

Si vous avez trouvé des solutions plus courtes ou un peu différentes, nous serons heureux des les lire si vous les envoyez à l'adresse suivante :

`info@prologin.org`

Correction proposée par Gabriel Duque (zuh0).

⁂

Nous espérons que la lecture de ces corrections vous a été aussi plaisainte qu'a été leur rédaction pour nous. Vive Prologin !