

# TP AISE — Runtime

## Exercice : Objet “String” en C

Implémentez votre propre **bibliothèque** C de gestion d'un type “chaîne de caractères”. On supportera l'ensemble des appels de fonctions suivants:

- `string* str_create(const char * str);`
- `void str_destroy(string*);`
- `void str_copy(string* s1, const string* s2);`
- `int str_compare(const string* s1, const string* s2);`
- `void str_append(string* s1, const string* s2);`
- `string* str_slice(const string* s1, size_t start_idx, size_t end_idx)`
- Bonus: `bool str_regex(const string* s1, const string *pattern);`

## Exercice : mmap / munmap

Le principe va être de faire communiquer deux processus au travers d'un fichier projeté en mémoire. Ceci vous familiarisera avec la fonction `mmap()`. Pour cela, réalisez dans l'ordre :

1. Créer un fichier de 64 caractères (minimum)
2. écrire un programme `writer.c` qui projette ce fichier en mémoire et inscrit son PID au début de l'espace mémoire projeté. Ensuite, il est mis en pause et attend d'intercepter un signal (`SIGUSR1` par exemple) pour se terminer.
3. écrire un second programme `reader.c` qui projette ce même fichier en mémoire et lit le PID. Il appelle ensuite `kill(SIGUSR1)` au premier processus.
4. Est-il possible d'écrire au-delà des 64 caractères du fichier ? Pourquoi ?

## Exercice : Copie de fonction

L'objectif est de partager le code d'une fonction entre deux processus alors qu'un seul l'un deux possède le code de la fonction. Pour cela, dérouler la procédure comme suit:

1. Écrire un programme `owner.c` qui possède une fonction `size_t square(int n)`, qui retourne le carré d'un nombre `n`.
2. Dans la fonction `main()` de ce programme, créez un fichier `libsquare.o` projeté avec `mmap()`. On mettra la permission d'écriture ainsi que sa visibilité (`MAP_SHARED`).
3. On copiera ensuite le code de la fonction `square()` dans cette espace mémoire (attention aux permissions !)
4. Écrire un second programme `user.c` chargeant le fichier `libsquare.o` avec `mmap()` avec la permission d'exécution (`PROT_EXEC`).
5. Invoquer alors cette fonction `square()` depuis le `main()` de `user.c`. Que se passe-t-il si vous lancez plusieurs fois le programme `user.c` ?

## Exercice: Lecture ELF

Écrire un programme capable de lire le nom d'un programme passé en paramètre, dire s'il s'agit d'un programme ELF. Si oui, afficher le maximum d'informations à son sujet. A minima, on pourra lister les données de chaque segment chargé.

## Exercice: Allocateur mémoire

Reproduire le code fourni par l'allocateur mémoire de la libc. L'objectif n'est évidemment pas d'être aussi complet. On implémentera les fonctions malloc/realloc/calloc/free en s'appuyant sur une gestion manuelle des pages. On pourra gérer les différents blocs avec un mécanisme de liste chaînée.

## Exercice : Dépassement de pile

Dans un premier temps, compilez le code ci-dessous (en mode 32 bits si possible, -m32, plus simple). En vous basant sur le fonctionnement de la pile, exploitez le binaire pour lui faire lancer le shell:

```
#include <stdlib.h>
#include <stdio.h>
int main() {
    int var, check = 0x04030201;
    char buf[40];

    fgets(buf,45,stdin);

    if (check == 0xdeadbeef) {
        printf(« You Win !\n »);
        system("/bin/sh");
    }
    return 0;
}
```

## Exercice : Shellcode

Ici l'objectif est d'invoquer un shell en injectant son code de lancement directement dans le programme: le shellcode. Pour plus de fun, compilez, le code sous un autre utilisateur (root à vos propres risques) et donnez lui la permission d'exécution d'un tiers (setuid): `chmod 6744 ./a.out`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/*
gcc file.c -fno-stack-protector -no-pie -Wl,-z,relro,-z,now,-z,noexecstack
*/
int main(int argc, char **argv){
    char buffer[32];
    int len, i;
    scanf("%s", buffer);
    len = strlen(buffer);
    printf("Hello %s\n", buffer);
    return 0;
}
```

Voici un shellcode possible (source: <http://shell-storm.org/shellcode/files/shellcode-827.php>) de longueur 23 octets :

```
char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
```

## Exercice : mon propre readelf

Pour les plus avancés, écrire un programme C qui reproduit les capacités de "readelf". Le but ici n'est pas d'invoquer **readelf** mais de parcourir le programme (courant ou mappé depuis le système de fichiers) pour en analyser le contenu. Voici une rapide liste des fonctionnalités proposés:

- Déterminer si le fichier passé en paramètre est un fichier ELF
- Lister les sections présentes dans le fichier
- Lister les en-têtes de programmes du fichier
- Lister les symboles du programme (juste leur nom)
- ...

Il est fortement conseillé de s'appuyer sur `<elf.h>`, pour obtenir les types. Il est aussi possible d'utiliser `libelf` pour parcourir le binaire grâce à l'API, mais pour les plus débrouillards, le challenge sera de s'en passer. (En cas de besoin, s'adresser à [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf))