Dans un sous-repertoire nommé **my_secmalloc** à la racine du dépôt du projet.

Notions évaluées	Xp à gagner
Rendre un projet	1
Mise en oeuvre de logiciel sécurisé	19

Fichier à rendre:

```
.

0 directories, 0 files
```

Fichier fournit dans **provided_files.zip**:

```
.
├── include
│   ├── my_secmalloc.h
│   └── my_secmalloc_private.h
├── lib
├── libmy_secmalloc.a
├── Makefile
├── sample
│   ├── heap_overflow_from_file.cc
│   ├── Makefile
│   ├── README.md
│   ├── save1.db
│   ├── save2.db
│   └── xploit_heap_overflow_from_file.py
├── src
│   └── my_secmalloc.c
└── test
    └── test.c

5 directories, 12 files
```

Sujet:

Il s’agit de ré-écrire les fonctions suivantes:

```c
void    *malloc(size_t size);
void    free(void *ptr);
void    *calloc(size_t nmemb, size_t size);
void    *realloc(void *ptr, size_t size);
```

Le comportement sera similaire aux fonctions équivalentes décrites dans *man 3 malloc*.
L’utilisation du *malloc* de la bibliothèque standard C, ainsi que les autres fonctions décrites dans le *man malloc* sont interdites. Vous aurez à utiliser *mmap*, *munmap* et *mremap*.

1 - Contexte

Préliminaires

- Votre code devra respecter au maximum les régles de programmation pour le développement sécurisé de logiciels en langage C de l'ANSSI et notamment l'annexe E concernant les conventions de codage et de nommage.
- Votre code devra respecter le principe de Test Driven Developpement, et ainsi vous devez compléter le fichier de test fournis.
- Vos tests doivent servir de preuve de l'implémentation des différentes fonctionnalités demandées dans le présent document.

Tests

- Le Makefile fourni compile et exécute les tests écrits grâce à *Criterion* dans *test/test.c*.

Bibliothèque statique

- Votre Makefile permet de compiler une bibliothèque statique nommé *libmy_secmalloc.a* par la commande *make clean static*.
- Cette bibliothèque est utilisé par vos test et donne accès à l'intégralité des fonctions/variables internes testés.

Bibliothèque dynamique

- Votre Makefile permet de compiler une bibliothèque dynamique nommé *lib/libmy_secmalloc.so* par la commande *make clean dynamic*.
- Cette bibliothèque n'exporte comme symbole publique que ceux listé ici:

```
$ nm libmy_secmalloc.so | grep " T " | cut -f3 -d' ' | sort
calloc
free
malloc
realloc
```

- En utilisant un mécanisme de l'éditeur de lien dynamique via la variable d'environnement nommé *LD_PRELOAD*, il vous sera possible de forcer l'utilisation de vos fonctions d'allocations mémoires pour n'importe quel programme.

tel que:

```
$ ls
test
$ ls ~/my_secmalloc/lib/libmy_secmalloc.so
libmy_secmalloc.so
$ LD_PRELOAD=~/my_secmalloc/lib/libmy_secmalloc.so ls
test
$
```

Grâce à ce mécanisme vous pouvez réaliser un test End2End (de bout en bout) de toutes vos fonctions.

2 - Objectif

L'objectif de ce projet est de parfaire votre compréhension des domaines suivants:

- Système d'exploitation Linux,
- Mécanisme de gestion mémoire,
- Compréhension des "heap overflows".

Toutefois, il ne s'agit pas ici d'avoir un projet axés sur la performance mais sur la sécurité, ainsi l'emphase ne sera ni sur l'optimisation de la consommation de l'espace mémoire, ni sur le temps d'exécution des fonctions d'allocation/déallocation mais sur les informations obtenues pendant l'exécution d'un programme.

Écriture d'un allocateur ?

L'écriture d'un allocateur de mémoire n'est pas une tâche simple :

- Une partie des fonctions usuelles (printf, …) utiles pour la mise au point des programmes utilises l'allocateur de base *malloc*
- Certain mécanisme du système utilise aussi l'allocateur (ld.so)

Pour vous aidez à comprendre, vous pouvez commencer par réaliser un détournement de malloc en utilisant *dlsym*.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

void    *malloc(size_t size)
{
    (void) size;
    printf("Avant le vrai malloc %ld\n", size);
    void *(*m)(size_t) = dlsym(RTLD_NEXT, "malloc");
    printf("Après le vrai malloc %p\n", m);
    return m(size);
}
```

voir *man dlsym*

En crochetant l'intégralité des fonctions demandés vous pourrez constater des différents appels à *malloc*. N'oubliez pas que le projet sera validé lors d'une soutenance avec relecture de code, tous détournement du vrai *malloc* sera lourdement sanctionné.

Pour vous aider dans la mise au point de votre projet vous aurez besoin de réécrire une fonction de log à l'aide des fonctions suivantes:

- vsnprintf
- alloca

Algorithmes

Le principe de l'algorithme vu en cours est à privilégiée sur tout autre mais vous êtes libre sur son implémentation.

- Vous avez un **pool de data** obtenu par un premier *mmap* contenant vos data et vos canary.
- Le **pool de data** est accessible par une variable globale (mais dont le symbole est privé).
- Vous avez un **pool de meta-information** obtenu par un second *mmap* contenant une liste de descripteur de bloc du pool de data
- pour chaque bloc alloué par *malloc* votre descripteur devrait contenir :
    - pointeur vers le bloc dans le pool de data
    - état du bloc allouée (occupée ou libre)
    - taille du bloc
- Le **pool de meta-information** est lui aussi accessible par une variable globale (mais dont le symbole est privé).
- un appel à malloc:
    - cherchera un descripteur vers un bloc libre de data d'une taille suffisante dans le pool de meta-information
    - au pire cas, il crée se descripteur et le rajoute dans le pool de meta-information
    - dans le cas, où la zone mémoire dans le pool de data n'a pas assez de place, il faudra l'agrandir avec un *mremap*
    - dans le cas, où la zone mémoire dans le pool de meta-information n'a pas assez de place, il faudra l'agrandir avec un *mremap*
- un appel à free:
    - vérifie que le pointeur donné en paramètre correspond bien à un descripteur dans le pool de meta-information
    - que le descripteur pointe vers un bloc occupée
    - que le canary à la fin du bloc n'a pas été altéré, sinon cela provoquera au choix un log ou un arrêt du processus

- marque ce bloc comme libre

La gestion de la fragmentation de bloc ou de la défragmentation des blocs est laissé à la discrétion du binôme.

2.1 - Fonctionnalités de votre projet

Rapport d'exécution du programme

La présence de la variable d'environnement *MSM_OUPUT* provoquera la génération d'un rapport d'exécution dans le fichier décrit par la valeur de cette variable. Le format du rapport d'exécution est libre.

Il devra toutefois permettre de tracer à minima:

- Type de fonction appelé (malloc, free, …)
- Taille des blocs demandés
- Adresse obtenu (a des fins d'identification)

man getenv

D'autres variable d'environnement peuvent être proposé pour contrôler le comportement de *malloc*.

Détection de malveillance

L'emphase du projet est mis sur votre capacité à détecter les erreurs de manipulation de la mémoire et à les écrire dans le rapport d'exécution:

- Heap overflow
- Double free

Et si vous êtes capables de détecter la fin de l'exécution d'un programme:

- Memory leak

Autres fonctionnalités

Vous êtes libre de proposer des fonctionnalités supplémentaires en rapport avec la sécurité des programmes tant que le cadre présenté ici est respecté.

Par exemple:

- Canary randomisé.
- Détection dynamique de l'overflow via un thread de parcours du tas.
- Algorithme novateur via gestion des pages faults (**man userfaultfd**)!

2.2 - Soutenance

- Le projet sera évalués lors d’une soutenance.
- La soutenance se passera sur votre machine (passez en *QWERTY* svp et installer *gvim*).
- Une revue de code sera réalisé et vous serez évalué sur la qualité de votre code aussi bien esthétique que fonctionnel. La sécurité d’un code est corollaire de sa maintenabilité.
- Le niveau de rendu et la richesse des fonctionnalités proposés sont les critères principaux de votre évaluation.
- Vous pourrez être soumis à une session de *recode*. Une sous-fonction de votre rendu est effacée et vous devez la recoder pendant le créneau de soutenance sous le contrôle d’un assistant.