linux kernel slub allocator writeup

4 SI 3 2025 - 2026

Groupe	Date
Groupe	18 Janvier 2026

Version Finale -

Nom	Prenom	@ github
Agrane	Sabrina	sabrina1ag
Fergui	Sabrina	

Table des matières

Table des matières

I. Introduction : contraintes de la mémoire dans le kernel Linux

A. Pourquoi le kernel Linux a besoin d'une heap différente ?

Le kernel Linux ne peut pas utiliser la même heap que les programmes userland pour plusieurs raisons fondamentales :

- **Memory Efficient** : Le kernel tourne en continu (contrairement à un programme userland) et ne peut pas être redémarré facilement. Un redémarrage du kernel correspond en pratique à un reboot de la machine entière. Il doit donc gérer la mémoire de façon très efficace, sans fuites ni pertes inutiles.

- **Performance** : Les opérations mémoire du kernel doivent être rapides et déterministes, car chaque cycle CPU compte pour le fonctionnement global du système.

B. Limites des allocateurs userland :

Les allocateurs de userland comme glibc, ne sont pas optimisés pour le kernel, ils stockent beaucoup de métadonnées (reading, writing, size, previous size, pointer, nextpointer, large bins), aussi, on perds beaucoup de temps CPU à bouger les free chunks entre différents bins.

C. La fragmentation mémoire :

Si l'allocateur fournit un chunk plus grand que nécessaire, il doit le **diviser** en deux, générant une fragmentation interne.

La fragmentation se produit lorsque les allocations et deallocations répétées de tailles variables créent des "trous" dans la mémoire.

Exemple :

```
char *my_buf = malloc(0x80);
free(my_buf);
my_buf = malloc(0x60);
```

Dans cet exemple, il est probable que le chunk précédemment libéré soit réutilisé pour le second **malloc**. Le chunk plus grand est alors **découpé en deux** : un chunk de 0x60 utilisé et un reste de 0x20 inutilisé.

Si cette opération est répétée (une boucle), de nombreux petits chunks inutilisés apparaissent, générant de la fragmentation de mémoire. Cette situation est particulièrement problématique pour le kernel, qui tourne indéfiniment et ne peut se permettre de gaspiller des ressources.

⇒ Side Note : ce n'est pas 100% véridique, il est probable qu'un autre chunk soit utilisé au second malloc, c'est un exemple illustratif.

Pour répondre à ces contraintes, le kernel utilisé les **slab allocators**, le document suivant va détailler leur fonctionnement, utilité et architecture.

II. Slab Allocators - Historique :

A. Terminologie

Pour ce document, nous utiliserons les normes de nomination standard :

- slab (en minuscule) désigne un espace mémoire contigu pouvant inclure les trois types d'allocateurs.
- SLAB (en majuscule) désigne une implémentation spécifique du slab générique.

B. Concrètement, un slab allocator c'est quoi ?

Un **slab allocator** est un mécanisme qui **pré-alloue des blocs mémoire** (appelés *slabs*), chacun contenant plusieurs **objets de taille identique**.
Ces objets sont organisés en **listes**, ce qui permet une **allocation et désallocation rapide**, tout en minimisant la fragmentation et le coût en temps CPU.

⇒ les slabs, et objet, ainsi que la listes seront détaillé plus bas (voir section XX)

Le kernel Linux a évolué à travers **trois générations de slab allocators** :

1. **SLOB (Simple List of Blocks) – "As compact as possible" :**

- Utilisé de **1991 à 1999** et encore présent dans certains **systèmes embarqués**.
- Pas de Structure de cache complexe (notion exploré un peu plus bas dans ce document)
- fragmentation encore élevée.
- Pas de Structure de cache complexe (notion exploré un peu plus bas dans ce document)
- Origine : inspiré du livre *K&R*.

2. **SLAB (Solaris-type allocator) – "Cache friendly"**

- Utilisable de 1999 à 2008, mais pas le mode par défaut dans Linux.
- Conçu pour être **benchmark friendly**, avec une structure adaptée à la **localité des caches CPU**.
- Structure typique (`kmem_cache (detaillé un peu plus bas)`) :

```
kmem_cache
  ├── slabs_full      # slabs complètement alloués
  ├── slabs_partial   # slabs partiellement alloués
  └── slabs_free      # slabs vides
```

3. **SLUB (Uncured slab allocators) - mode par défaut actuel :**
- Debugging amélioré
- Meilleure défragmentation
- Exécution plus rapide

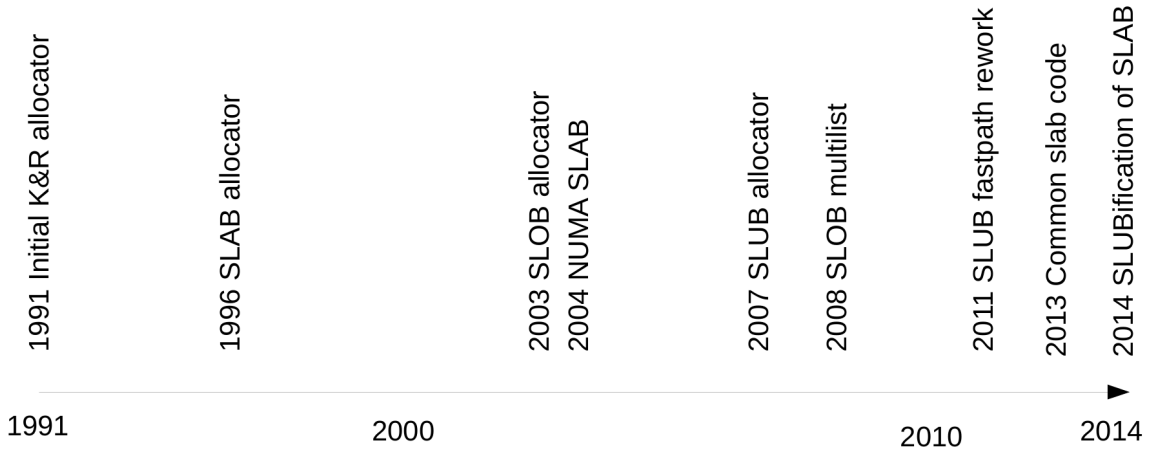


Figure 1 : Timeline - Slab Subsystem Développement - Crédit : Christoph Lameter, LinuxCon/Düsseldorf 2014

III. SLAB Allocator :

Comm : A AJOUTER : PERCPU, Alignement,

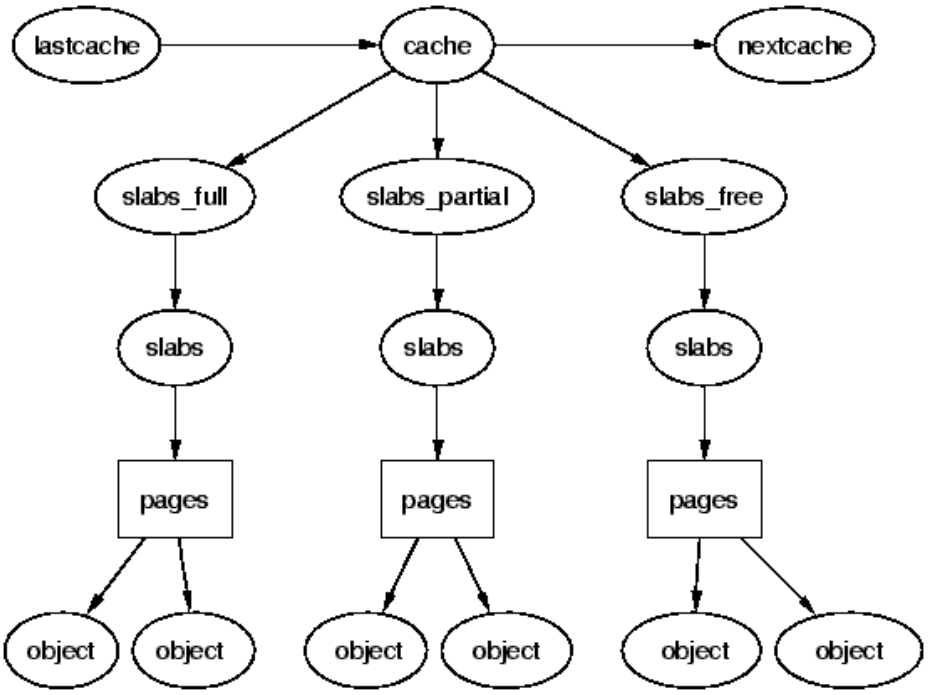
Nous allons maintenant détailler la structure et les composants d'un **slab allocator**.
L'objectif principal de ce document reste la description du **SLUB allocator** ; toutefois, une compréhension préalable du **SLAB allocator** est indispensable pour saisir pleinement les évolutions et améliorations introduites par SLUB.

Le SLAB allocator est composé d'un nombre variable de **caches**, reliés entre eux au sein d'une **liste doublement chaînée circulaire**, appelée *cache chain*.

Dans le contexte du slab allocator, un **cache** est un gestionnaire d'objets d'un type donné (par exemple `mm_struct` ou `fs_cache`). Chaque cache est représenté et administré par une structure `struct kmem_cache_s`, qui sera détaillée ultérieurement. Les différents caches sont chaînés entre eux via le champ `next` de cette structure.

Chaque cache maintient plusieurs blocs de mémoire contigus, appelés **slabs**, constitués d'une ou plusieurs pages physiques. Ces slabs sont ensuite découpés en unités de taille fixe afin d'héberger

les structures de données et les objets kernel pris en charge par le cache, comme illustré dans le schéma suivant.



Les structures d'un SLAB Allocateur et leur positionnement - Source : https://www.kernel.org/doc

A. Cache

Il est possible d'obtenir une vue d'ensemble des caches existants sur un système Linux via la commande suivante :

```
cat /proc/slabinfo
```

Ce fichier expose des informations synthétiques sur chaque cache, notamment :

- **cache-name** : nom lisible du cache (ex. tcp_bind_bucket)
- **num-active-objs** : nombre d'objets actuellement utilisés
- **total-objs** : nombre total d'objets disponibles, y compris les objets libres
- **obj-size** : taille de chaque objet, généralement faible
- **num-active-slabs** : nombre de slabs contenant au moins un objet actif
- **total-slabs** : nombre total de slabs existants
- **num-pages-per-slab** : nombre de pages nécessaires pour former un slab (souvent 1)

La définition et la logique de gestion d'un cache se trouvent principalement dans le fichier mm/slab.c.

La structure centrale représentant un cache est struct kmem_cache_s :

```
struct kmem_cache_s {
    struct list_head        slabs_full;
    struct list_head        slabs_partial;
    struct list_head        slabs_free;
    unsigned int            objsize; # taille de chaque objet contenu dans le slab
    unsigned int            flags; # indicateurs contrôlant le comportement de l'allocateur pour ce cache
    unsigned int            num; # nombre d'objets contenus dans chaque slab
    spinlock_t              spinlock; # protège la structure contre les accès concurrents
#ifdef CONFIG_SMP
    unsigned int            batchcount; # nombre d'objets alloués en une seule fois pour les caches per-CPU
#endif
};
```

slabs_* : listes contenant respectivement les slabs entièrement alloués, partiellement alloués et totalement libres

Note : la notion de per-cpu sera exploré plus loin dans ce document.

Cache Static Flags :

Les caches disposent de flags définis lors de leur création, qui restent constants pendant toute leur durée de vie. On distingue deux catégories :

- Flags defini par le SLAB Allocateur
- Flags defini par le créateur du Cache

On a aussi deux autres catégorisation de flags (dynamic, allocation), se référer au slab allocator document pour les détails exacte sur ces flags.

Cache Coloring :

Le **cache coloring** est une optimisation orientée **CPU** et non mémoire.
Au lieu de faire débuter tous les slabs au même offset mémoire, le slab allocator décale le point de départ des objets dans chaque slab afin de réduire les collisions dans le cache matériel du processeur.

Ce calcul est effectué lors de la création du cache, via kmem_cache_create().

Exemple de détermination du cache coloring :

Supposons que l'adresse de base du SLAB est (s_mem = 0), avec 100 octets inutilisés dans le slab et un alignement L1 = 32 octets.

Les offsets possibles serait ; 0, 32, 64, 96, la color serait 3, et le color offset serait 32.

Allocations des SLAB se fera comme suit :

Slab	**Offset de départ**
Slab 1	0
Slab 2	32
Slab 3	64
Slab 4	96
Slab 5	0 (wrap-around)

C'est relativement complexe et constitue l'une des raisons ayant motivé l'introduction du **SLUB allocator**, qui simplifie fortement ce modèle.

Cache Creation, Reaping & Shrinking, Destruction :

La fonction **kmem_cache_create()** est responsable de la création des caches et de leur insertion dans la *cache chain*. Parmi ses principales tâches :

- calcul du cache coloring ;
- initialisation des champs restants du descripteur de cache ;
- ajout du cache à la liste globale des caches ;
- alignement de la taille des objets sur la taille des mots machine.

Cache Reaping et Cache Shrinking (SLAB)

Cache Reaping :

Le reaping consiste à sélectionner un cache candidat et à lui demander de réduire sa consommation mémoire.
La sélection ne prend pas en compte les nœuds NUMA ni les zones mémoire, ce qui peut entraîner la libération de mémoire dans des régions non soumises à pression. Ce comportement est acceptable sur des architectures simples comme x86.

Lors du reaping :

- seul un nombre limité de caches est examiné à chaque itération ;
- les caches récemment agrandis ou en cours de croissance sont évités
- les caches capables de libérer le plus de pages sont favorisés
- une partie des slabs présents dans `slabs_free` est libérée.

Cache Shrinking :

Une fois un cache sélectionné, le shrinking est volontairement simple et agressif :

- les caches per-CPU sont vidés ;
- Les slabs présents dans `slabs_free` sont libérés.

Deux variantes existent :

- une fonction destinée aux utilisateurs du slab allocator, qui libère uniquement les slabs libres ;
- une fonction interne, utilisée lors de la destruction complète d'un cache, garantissant que celui-ci est entièrement vidé.

PARTIE SLUB

C. Cache, Slabs & Objets :

1. **Cache :** gère les objets de taille fixe, chaque cache contient plusieurs slabs et permet d'allouer rapidement, des objets de la même taille. exemple :
    - Cache pour objets de 256 octets
    - Cache pour objets de 512 octets

   Il est également possible de créer des **caches personnalisés** pour un type d'objet kernel particulier.

2. **Slabs :** Un **slab** est un **bloc contigu de mémoire**, composé d'une ou plusieurs pages physiques, chaque slab appartient toujours à un **seul cache** et contient plusieurs **slots**, chacun capable de stocker un objet de taille fixe.

3. Un **slot** est une **région de mémoire de taille fixe** dans un slab, déterminée par le cache auquel il appartient, il est soit :
    - libre
    - occupé

   exemple : dans le cas d'un kmalloc le pointeur retourné est sur l'un de ces slots disponible, quand il sera utilisé on dira que ce slots contient un objet (un kernel objet)
   <ajouter schéma conceptuelle a refaire avec excel alone inspiré de pwn>

D. Slab Allocators - Schéma Technique :

Le cache (kmem_cache) utilise deux types de structures

- **kmem_cache_cpu**
- **kmeme_cache_node**

Figure 3 - Schema Technique Slab Allocators - Source : pwn.college Kernel Exploitation - Slab Allocators

- Dans cette figure, ce qui est à gauche (**kmem_cache_cpu**) peut être considéré comme le "working" slabs in use.
- Ce qui est à droite (**kmeme_cache_node**) c'est ce qui n'est pas actuellement utilisé par l'actuel cpu

Pourquoi as-on une division ?

- c'est exactement comme dans le userland avec les threads, on ne veut pas avoir des threads qui s'attendent entre eux donc, et ça nous permet de savoir ce que le CPU actuel utilise sans avoir de la contention avec d'autres cpu.

Pour chaque **cache** et pour chaque **CPU**, il existe un **slab actif**, utilisé pour les allocations courantes.
Lorsque ce slab actif ne contient plus de slots libres et qu'une nouvelle allocation est requise, le kmem_cache va rechercher un slab disponible au niveau du **node** et en **revendiquer la propriété** pour le CPU courant.

Ce slab peut déjà contenir quelques slots alloués, mais tant qu'il dispose encore de slots libres, il peut être utilisé comme slab actif.

Les slabs qui contiennent encore au moins un slot libre sont suivis dans une liste nr_partial, maintenue au sein de la structure kmem_cache_node.

E. **Slab Memory - Slots : Comment les objets sont alloués dans le slab, dans les slots et comment ces slots sont reliés ?**

Lorsqu'un objet est libéré, son slot est inséré dans une liste simplement chaînée des slots libres, si un second objet est libéré, celui-ci est ajouté en tête de la liste, ce qui met à jour les pointeurs internes, la structure kmem_cache_cpu conserve un pointeur vers la tête de

cette liste de slots libres, le slot correspondant au dernier objet libéré pointe vers le slot libéré précédemment, qui pointe à son tour vers NULL lorsqu'il n'y a que deux slots libres.

Schématisation de l'exemple :

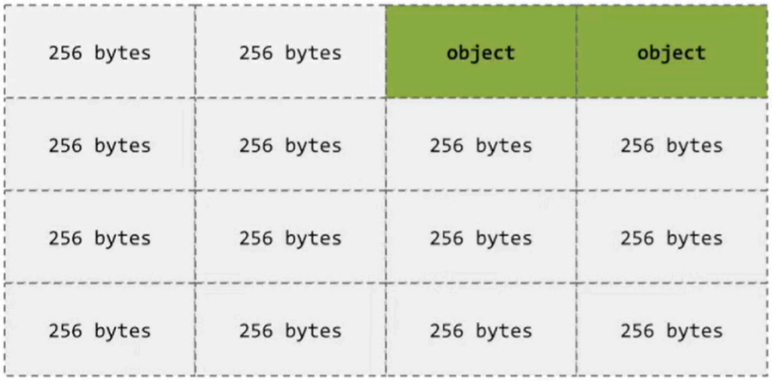


Figure 4 - Slots Exemple - Source pwn.college Kernel Exploitation - Slab Allocators

Libération d'un object :

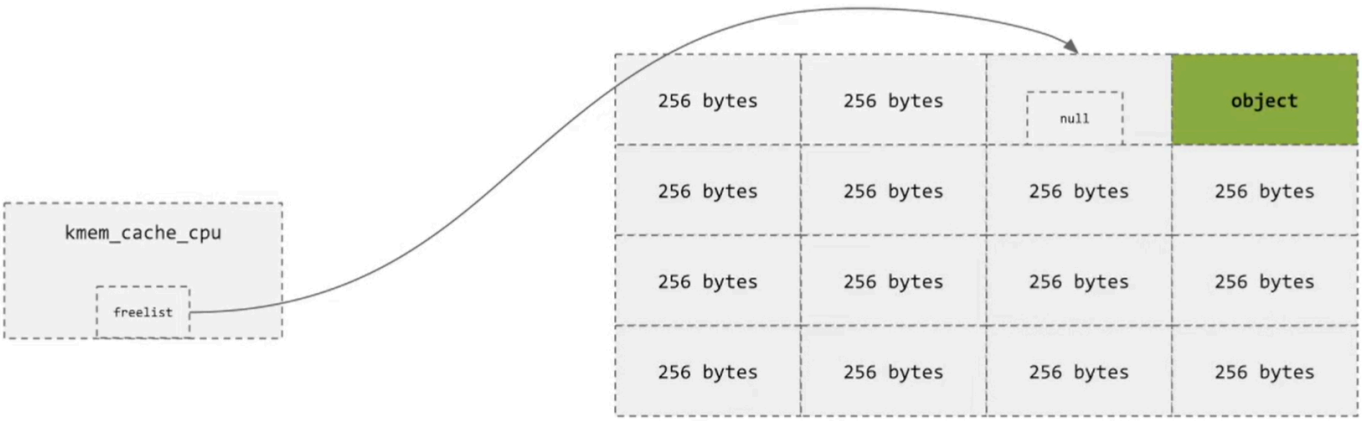


Figure 5 - Free Object - Source pwn.college Kernel Exploitation - Slab Allocators

La tête de la **liste LIFO des slots libres** est référencée par le cache.
Lorsqu'un second objet est libéré, il est inséré en tête de la liste, formant ainsi une **liste chaînée** de slots libres. Dans chaque slot libre, une partie des octets est utilisée pour stocker **l'adresse du slot libre suivant**, permettant de chaîner les slots entre eux sans structure de métadonnées externe, comme dans la figure ci dessous :

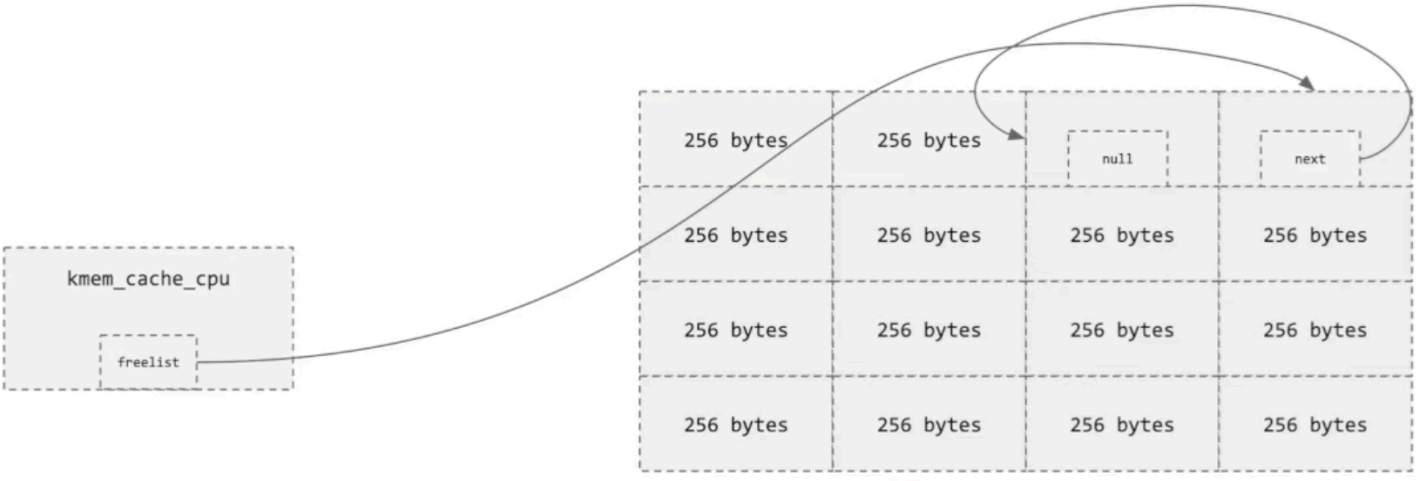


Figure 6 - Free Object Chained List - Source pwn.college Kernel Exploitation - Slab Allocators

Pour un objet de taille `0x100` octets, une partie de la mémoire de l'objet — par exemple à l'offset `0x80` — est utilisée pour stocker le **pointeur vers le prochain slot libre** lorsque l'objet est libéré.

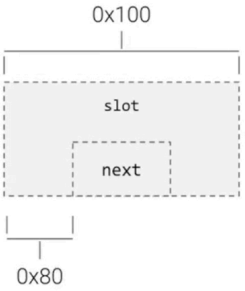


Figure 7 - Object Exemple - Source pwn.college Kernel Exploitation - Slab Allocators

expliquer c'est quoi un kernel object ;kernel object (fd, semaphores, file objects)
Ce modèle a également des implications importantes en matière de sécurité, notamment dans le contexte des vulnérabilités de type use-after-free, heap overflow ou exploitation du kernel heap.