

# 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

<b>Table des matières</b>	<b>2</b>
<b>Compte rendu TP1</b>	<b>2</b>
1.	3
2.	3
3.	4
1.	4
2.	5
3.	6
3.1	7
4.	8
4.	10
5.	16
C	17
	17
1.	17
2.	20
3.	25
4.	28
5.	30
2.	31

## 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 perd 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 utilise les **slab allocators**, le document suivant va détailler leur fonctionnement, utilité et architecture.

## II. Slab Allocators :

### 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. 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** (détaillé 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

## 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**
- **kmem\_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 (**kmem\_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 :

256 bytes	256 bytes	object	object
256 bytes	256 bytes	256 bytes	256 bytes
256 bytes	256 bytes	256 bytes	256 bytes
256 bytes	256 bytes	256 bytes	256 bytes

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

Libération d'un object :

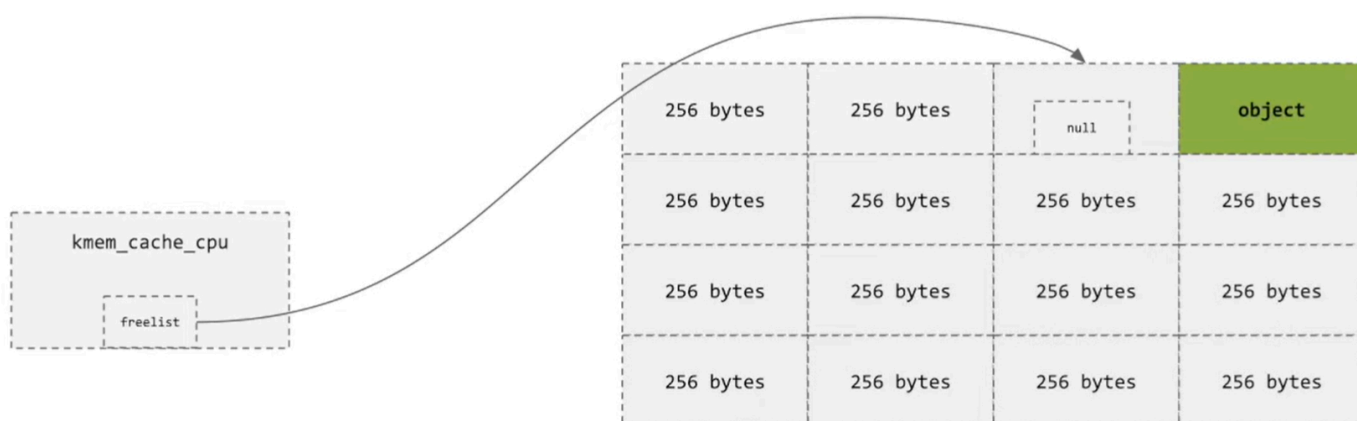


Figure 5 - Free Object - Source pwn.college Kernel Exploitation - Slab Allocators

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 :

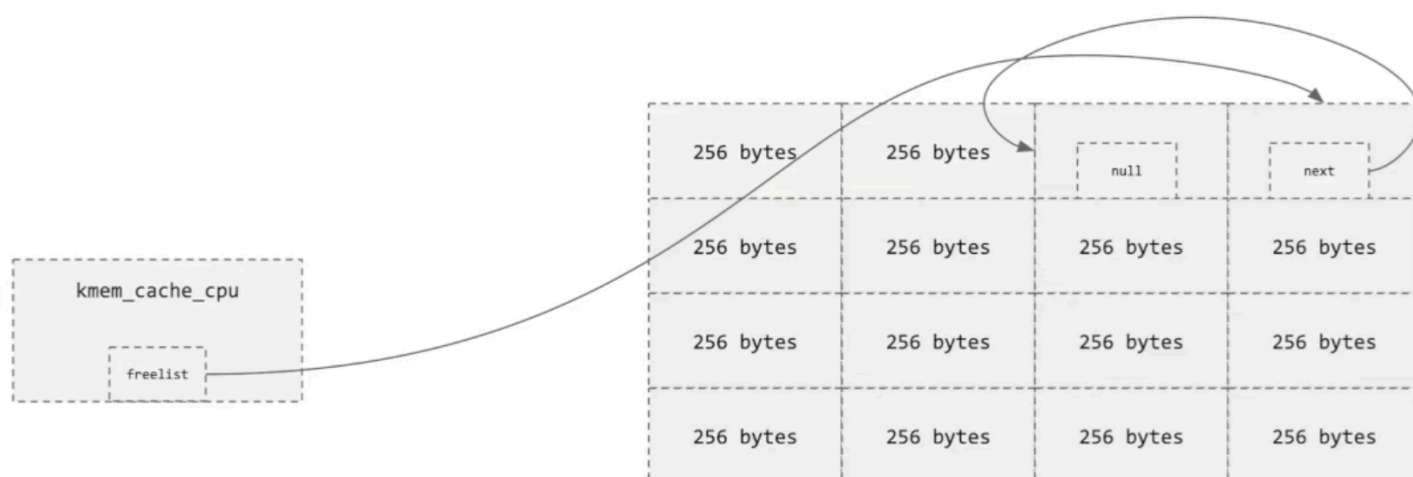


Figure 6 - Free Object Chained List - Source pwn.college Kernel Exploitation - Slab Allocators

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.

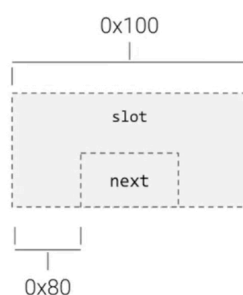


Figure 7 - Object Exemple - Source pwn.college Kernel Exploitation - Slab Allocators