

linux kernel slub allocator writeup

4 SI 3 2025 - 2026

Groupe	Date
Groupe 22	18 Janvier 2026

Nom	Prenom	@ github
Agrane	Sabrina	sabrina1ag
Fergui	Sabrina	sabrinafergui2

Table des matières

Table des matières	2
I. Introduction : contraintes de la mémoire dans le kernel Linux	3
A. Pourquoi le kernel Linux a besoin d'une heap différente ?	3
B. Limites des allocateurs userland	3
C. La fragmentation mémoire	3
II. Slab Allocators - Historique	4
A. Terminologie	4
B. Concrètement, un slab allocator c'est quoi ?	4
III. SLAB Allocator :	6
A. Vue d'ensemble : cache chain, caches et slabs	6
B. Notion de cache et structure kmem_cache_s	7
1. Rôle d'un cache	7
2. Structure centrale struct kmem_cache_s	7
3. Cache static flags	9
4. Alignement des objets dans le SLAB allocator	9
Cache Coloring :	10
IV. Pourquoi le SLUB Allocator remplace le SLAB	12
IV-A. Gestion complexe des files d'objets	12
IV-B. Surcharge mémoire des files d'objets	12
IV-C. Surcharge de métadonnées dans SLAB	13
IV-D. Simplification du mécanisme de reaping	13
IV-E. Simplification de la politique NUMA	13
IV-F. Accumulation des slabs partiels	14
IV-G. Fusion des caches (Slab Merging)	14
IV-H. Diagnostics et débogage	14
IV-I. Résilience et détection d'erreurs	14
IV-J. Tracing et analyse post-mortem	15
IV-K. Amélioration des performances	15
V- Slub Internals	16
1- Concepts généraux : Cache, Slabs & Objets	16
2- Slab Memory - Slots & Freelist : Comment les objets sont alloués dans le slab, dans les slots et comment ces slots sont reliés ?	16
3- kmem_cache slub	18
4- Structures internes SLUB	19
5- Processus d'allocation SLUB	23
6- Processus de libération (Freeing)	24
7- Exploitation du SLUB Allocator	26
8- UAF Exploit	30
VI- Conclusion	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 - 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` (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

III. SLAB Allocator :

Nous allons maintenant détailler la structure et les composants d'un SLAB allocator.

Le SLAB allocator constitue une implémentation du concept générique de *slab allocator* et est utilisé par le noyau Linux pour gérer efficacement l'allocation dynamique de structures kernel de taille fixe.

Il repose sur une organisation hiérarchique précise : cache → slabs → objets.

Cette organisation permet d'optimiser les allocations mémoire en réduisant la fragmentation interne, en favorisant la réutilisation des objets et en améliorant la localité mémoire ainsi que l'efficacité du cache processeur.

A. Vue d'ensemble : cache chain, caches et slabs

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*. Cette liste globale permet au noyau de parcourir l'ensemble des caches existants, notamment lors d'opérations de maintenance telles que le reaping ou le shrinking.

Dans le contexte du SLAB allocator, un **cache** est un gestionnaire d'objets d'un **type unique et bien défini**, par exemple `mm_struct`, `inode`, `file` ou `dentry`. Chaque cache est responsable de l'allocation, de la libération et de la réutilisation des objets appartenant à ce type.

Chaque cache maintient plusieurs blocs de mémoire contigus appelés **slabs**.

Un **slab** est constitué d'une ou plusieurs **pages physiques contiguës**, obtenues depuis l'allocateur de pages (buddy allocator), puis découpées en **objets de taille fixe**. Ces objets sont les unités réellement retournées lors des allocations.

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

figure x.2

Lecture du schéma :

- la cache chain regroupe tous les caches du système ;
- chaque cache est indépendant et spécialisé ;
- les slabs servent de réserve d'objets prêts à l'emploi.

Cette organisation permet au noyau d'éviter des allocations de pages fréquentes et coûteuses pour chaque petit objet kernel.

B. Notion de cache et structure `kmem_cache_s`

1. Rôle d'un cache

Un cache a pour rôle principal de **gérer des objets de taille fixe** appartenant à un même type.

Cette contrainte est fondamentale : un cache ne doit jamais contenir des objets de tailles différentes, car cela permet :

- une gestion simple et rapide des allocations ;
- une réduction significative de la fragmentation interne ;
- une meilleure prévisibilité des accès mémoire.

Il est possible d'obtenir une vue d'ensemble des caches présents sur un système Linux via :

```
cat /proc/slabinfo
```

Ce fichier fournit, pour chaque cache :

- `cache-name` : nom lisible du cache ;
- `num-active-objs` : nombre d'objets actuellement utilisés ;
- `total-objs` : nombre total d'objets (libres + utilisés) ;
- `obj-size` : taille d'un objet ;
- `num-active-slabs` : slabs contenant au moins un objet actif ;
- `total-slabs` : nombre total de slabs ;
- `num-pages-per-slab` : nombre de pages physiques par slab.

Ces informations donnent une vision globale de l'utilisation mémoire du SLAB allocator.

2. Structure centrale struct `kmem_cache_s`

Chaque cache est représenté dans le noyau par une structure centrale : `struct kmem_cache_s`.

Cette structure contient **toutes les métadonnées nécessaires** à la gestion des

slabs et des objets associés.

```
struct kmem_cache_s {
    struct list_head slabs_full;
    struct list_head slabs_partial;
    struct list_head slabs_free;
    unsigned int objsize;
    unsigned int flags;
    unsigned int num;
    spinlock_t spinlock;
#ifdef CONFIG_SMP
    unsigned int batchcount;
#endif
};
```

a) Listes de slabs

- **slabs_full** : slabs dont **tous les objets sont alloués** ;
- **slabs_partial** : slabs contenant **au moins un objet libre** (prioritaires pour l'allocation) ;
- **slabs_free** : slabs **entièrement libres**.

Cette séparation permet au noyau de choisir rapidement le slab le plus adapté lors d'une allocation.

b) Informations sur les objets

- **objsize** : taille réelle d'un objet géré par le cache (après alignement) ;
- **num** : nombre total d'objets contenus dans un slab.

Ces champs déterminent directement la capacité d'un slab et l'efficacité mémoire du cache.

c) Synchronisation

- `spinlock` : protège les structures globales du cache contre les accès concurrents, notamment sur les systèmes multi-cœurs.

d) Support SMP

- `batchcount` (si `CONFIG_SMP`) : nombre d'objets transférés en une seule opération entre les structures globales du cache et les caches locaux par CPU.

Note : la notion de caches *per-CPU* sera détaillée plus loin. À ce stade, il est important de comprendre que ce champ vise à réduire la contention liée aux accès concurrents.

3. Cache static flags

Les caches disposent de **flags statiques**, définis lors de leur création et constants durant toute leur durée de vie.

On distingue notamment :

- des flags définis par le SLAB allocator lui-même ;
- des flags définis par le créateur du cache.

D'autres catégories de flags (dynamiques, liés à l'allocation) existent également et sont décrites plus en détail dans la documentation du SLAB allocator.

4. Alignement des objets dans le SLAB allocator

En complément des flags statiques, le SLAB allocator applique des **contraintes d'alignement** lors de la création des caches.

L'alignement détermine la manière dont les objets sont positionnés en mémoire afin de respecter les contraintes matérielles de l'architecture cible et d'optimiser les performances.

Lors de la création d'un cache via `kmem_cache_create()`, la taille logique d'un objet (taille de la structure C) est **ajustée** afin de :

- respecter l'alignement naturel des mots machine (par exemple 4 ou 8 octets selon l'architecture) ;

- éviter les accès mémoire non alignés, qui peuvent entraîner des pénalités de performance ou être interdits sur certaines architectures ;
- améliorer la localité mémoire et réduire les accès mémoire inutiles.

L'alignement est **calculé une seule fois**, au moment de la création du cache, et reste constant pendant toute la durée de vie de celui-ci.

Il conditionne directement la disposition des objets à l'intérieur des slabs.

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 défini par le SLAB Allocator
- Flags défini 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 débiter 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	Offset de départ
Slab 1	0
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.

IV. Pourquoi le SLUB Allocator remplace le SLAB

Au fil des versions du noyau Linux, de nombreux correctifs ont été appliqués à l'allocateur SLAB afin de corriger des bugs et d'améliorer ses performances. Cependant, ces correctifs successifs ont accru la complexité du code, notamment dans le fichier `mm/slab.c`, rendant l'allocateur difficile à maintenir, à déboguer et à faire évoluer.

Le SLUB allocator a été introduit avec pour objectif principal de simplifier l'implémentation existante, de réduire la surcharge mémoire et d'améliorer la scalabilité sur les systèmes modernes multi-cœurs et NUMA, tout en conservant les performances offertes par les allocateurs de type slab.

IV-A. Gestion complexe des files d'objets

L'allocateur SLAB repose sur de nombreuses files d'objets (object queues), réparties par CPU et par nœud NUMA. Cette organisation implique des mécanismes de synchronisation complexes et un coût important lors des allocations et des libérations.

SLUB supprime entièrement ces files d'objets intermédiaires. Chaque CPU travaille directement sur son slab actif, sans passer par des files séparées. Cette simplification réduit considérablement la complexité du chemin critique d'allocation et de libération.

IV-B. Surcharge mémoire des files d'objets

Dans SLAB, les files d'objets existent :

- par CPU,
- par nœud NUMA,
- ainsi que sous forme de files dites *alien* pour les accès distants.

Sur des systèmes de grande taille comportant des centaines ou des milliers de CPU et de nœuds NUMA, cette multiplication des structures entraîne une consommation mémoire importante. Dans certains cas, plusieurs gigaoctets peuvent être utilisés uniquement pour stocker ces métadonnées, indépendamment des objets alloués.

SLUB élimine ces files, réduisant drastiquement la consommation mémoire et supprimant ce risque de dérive.

IV-C. Surcharge de métadonnées dans SLAB

Dans SLAB, les métadonnées sont stockées au début de chaque slab. Cette organisation empêche un alignement naturel des objets en mémoire et entraîne un gaspillage d'espace.

SLUB déplace l'ensemble des métadonnées dans la structure `struct page`. Les objets contenus dans un slab peuvent ainsi être naturellement alignés (par exemple, un objet de 128 octets aligné sur 128 octets) et remplir efficacement une page mémoire, ce que SLAB ne permet pas.

IV-D. Simplification du mécanisme de reaping

Le mécanisme de *cache reaping* est particulièrement complexe dans SLAB. Il nécessite des parcours coûteux des structures internes afin d'identifier les slabs pouvant être libérés.

SLUB simplifie ce mécanisme :

- sur un système mono-processeur, le reaping est inutile ;
- sur un système SMP, un slab per-CPU est simplement remplacé dans une liste de slabs partiels, sans parcours des objets.

Cette approche réduit la complexité et améliore les performances.

IV-E. Simplification de la politique NUMA

SLAB applique les politiques NUMA au niveau des objets individuels. Cela implique des accès fréquents aux politiques mémoire et peut entraîner des bascules coûteuses entre nœuds NUMA.

SLUB délègue entièrement la gestion NUMA à l'allocateur de pages. Cette approche simplifie la logique de l'allocateur slab et améliore la localité mémoire à un niveau plus approprié.

IV-F. Accumulation des slabs partiels

SLAB maintient des listes de slabs partiels par nœud NUMA. Avec le temps, ces listes peuvent croître de manière significative, augmentant la fragmentation mémoire, car les slabs partiels ne peuvent être réutilisés que par des allocations provenant du même nœud.

SLUB utilise un pool global de slabs partiels, permettant une réutilisation plus efficace des slabs et une réduction de la fragmentation.

IV-G. Fusion des caches (Slab Merging)

SLAB maintient de nombreux caches similaires sans mécanisme automatique de regroupement.

SLUB détecte les caches équivalents au démarrage du système et les fusionne automatiquement. En pratique, jusqu'à 50 % des caches peuvent être éliminés, ce qui :

- améliore l'utilisation de la mémoire,
- réduit la fragmentation,
- permet de remplir à nouveau des slabs partiellement alloués.

IV-H. Diagnostics et débogage

Les outils de diagnostic de SLAB sont difficiles à utiliser et nécessitent souvent une recompilation du noyau.

SLUB intègre nativement des mécanismes de débogage activables dynamiquement via `slab_debug`, sans pénaliser les chemins critiques lorsque ces options sont désactivées.

IV-I. Résilience et détection d'erreurs

Lorsque les vérifications de cohérence sont activées, SLUB est capable de détecter des erreurs courantes telles que :

- corruption mémoire,
- double libération,
- incohérences dans les freelists.

Dans certains cas, l'allocateur peut tenter de maintenir le système opérationnel malgré ces erreurs.

IV-J. Tracing et analyse post-mortem

SLUB permet le traçage précis des opérations effectuées sur un cache donné (`slab_debug=T`). Il est ainsi possible d'observer l'état des objets lors de leur libération, ce qui est particulièrement utile pour l'analyse post-mortem et le débogage de bugs complexes.

IV-K. Amélioration des performances

Les benchmarks montrent des gains de performance de l'ordre de 5 à 10 % sur des charges telles que *kernbench*. Grâce à des slabs de plus grande taille, une meilleure gestion de la fragmentation et une réduction de la complexité interne, SLUB offre un potentiel de scalabilité supérieur à SLAB.

Référence

LWN.net — *Why SLUB replaces SLAB*
<https://lwn.net/Articles/229096/>

V- Slub Internals

1- Concepts généraux : Cache, Slabs & Objets

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.

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.

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

Dans le cas d'un kcalloc 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).

2- Slab Memory - Slots & Freelist : 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, il est ajouté en tête de la liste, mettant à jour les pointeurs internes. la structure `kmem_cache_cpu` conserve un pointeur vers la tête de cette liste. Chaque slot libre utilise une partie de sa mémoire pour stocker l'adresse du slot suivant, sans métadonnées externes.

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 objet :

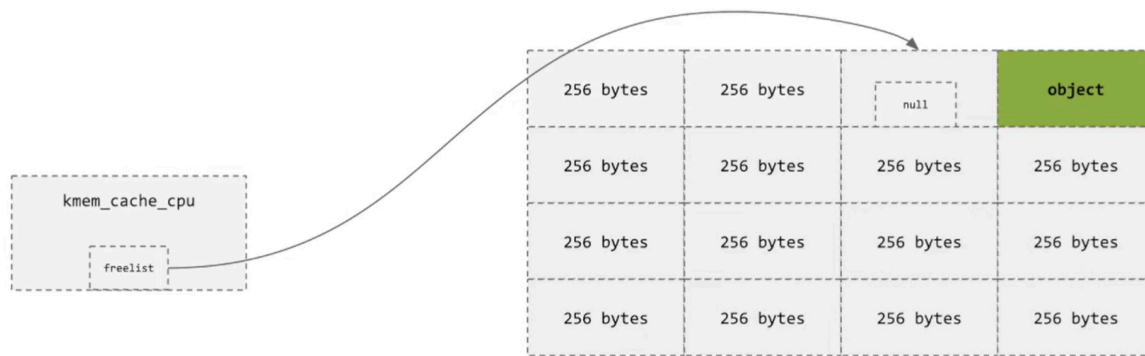


Figure 5 - Free Object - Source pwn.college Kernel Exploitation - Slab Allocators

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 :

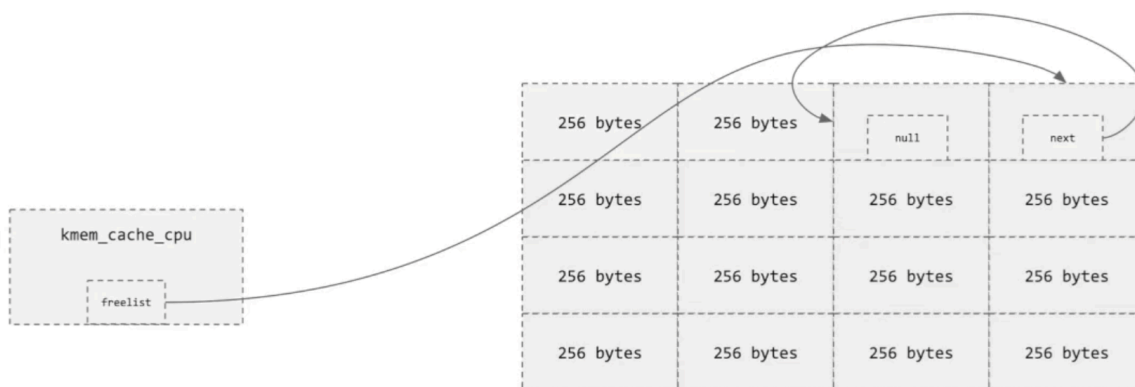


Figure 6 - Free Object Chained List - Source pwn.college Kernel Exploitation - Slab Allocators

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.

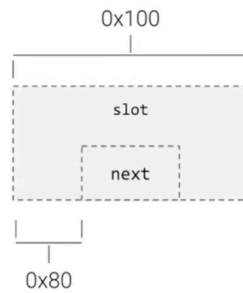
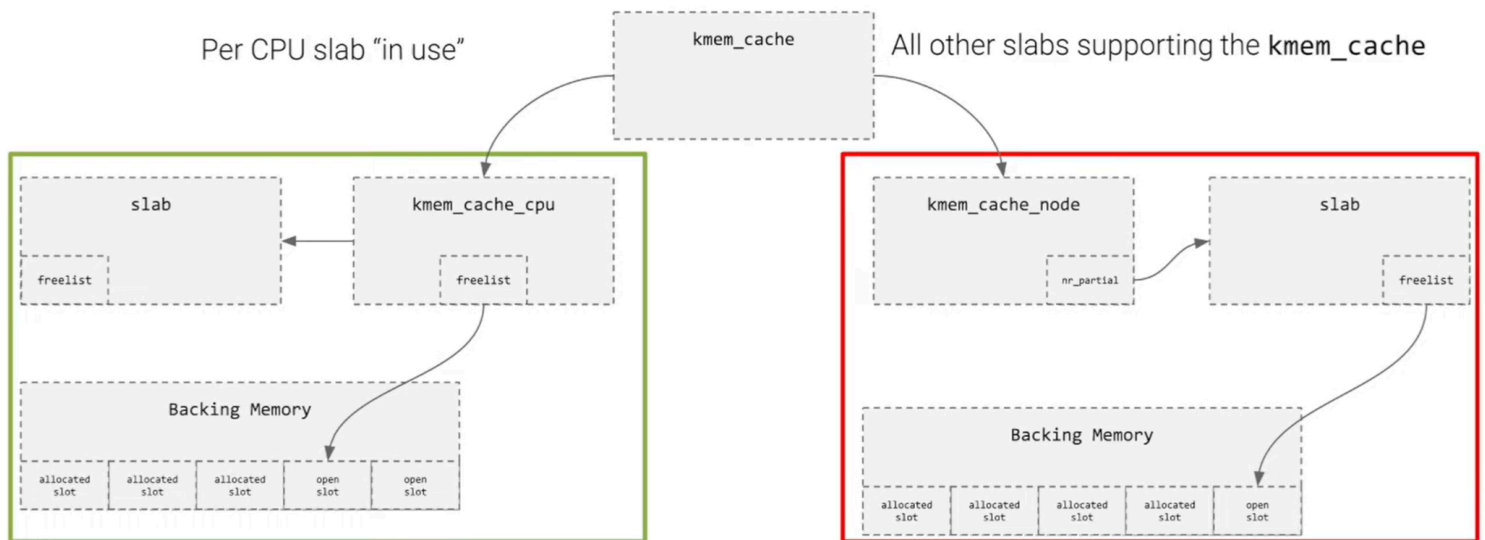


Figure 7 - Object Exemple - Source pwn.college Kernel Exploitation - Slab Allocators

—> la notion de libération d'un objet est exploré plus en détails dans la partie “freeing process”

3- kmem_cache slub



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.

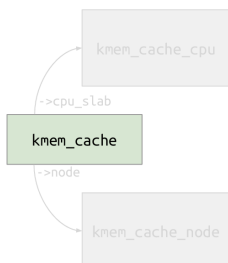
Ce slab peut déjà contenir quelques slots alloués, mais tant qu'il dispose encore de slots libres, il peut être utilisé comme slab actif.

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**.

4- Structures internes SLUB

cache :

struct kmem_cache



```

kmem_cache {
    /* CPU cache data:
    kmem_cache_cpu __percpu *cpu_slab;
    /* Node cache data:
    kmem_cache_node *node[MAX_NUMNODES];

    char *name; // Cache name
    slab_flags_t slab_flags; // Cache flags unsigned
    unsigned int size; // Size of objects unsigned
    unsigned int *freelist; // Freelist pointer offset
    unsigned long min_partial;
    unsigned int cpu_partial_slabs;

```

- **struct kmem_cache_cpu __percpu *cpu_slab;** il ya une instance de ce pointeur pour chaque cpu dans le système, il est lié à des données concernant un cpu, et la raison derrière ça c'est une question de performance (évite la contention), on ne voudrait pas avoir plusieurs variables communes pour plusieurs CPU.

- **struct kmem_cache_node *node[MAX_NUMNODES];** il ya une instance de ce pointeur pour chaque NUMA (non uniform memory access), c'est aussi pour de la performance, pour les pcs avec différents banque de mémoire et propriété d'accès, (on ne peut pas aborder la notion de un seul numa node qui discus avec plusieurs nodes).

Nous allons rester sur un seul CPU et un seul NUMA node pour ce qui suit, les cpu et les nodes ont des pointeurs vers des slabs, mais les nodes utilise une liste doublement chaînée

Struct Slab :

Pour Chaque Slab dans le linux kernel nous avons une structure slab correspondante, et pour chacun il ya une page structure qui elle correspond à une page physique, dépendant que si la page physique est normale est alloué seulement à ce slab, ou si elle est partagé avec un autre, le layout sera différent.

```

struct slab { // Aliased with struct page
    void **mem_cache; // Cache this slab belongs to
    struct slab *next; // Next slab in per-cpu list
    int free; // Slabs left in per-cpu list
    struct list_head slab_list; // List of free objects in per-node list
    struct list_head freelist; // Per-slab freelist
};

```

- chaque slab a une backing memory alloué via `page_alloc` et qui contient les objets slots
- `freelist` pointe vers le premier slot libre dans le slab, chaque slot libre pointe vers le suivant.

Le pointeur `freelist` est stocké près du milieu de l'objet pour prévenir la corruption en cas de petits débordements.

Le pointeur `freelist` est également `hashé` pour compliquer la falsification dans les exploits SLUB, c'est toujours possible mais au lieu de leak l'adresse du `freelist` on doit aussi leak l'adresse du `cache->random`, et du `swab(ptr_address)`, ça reste possible mais ça complique la tâche ! mais ce n'est pas la première cible/but d'un exploit kernel !

Quand la liste est totalement allouée, le `freelist` pointeur est juste NULL.

```

cache->offset = ALIGN_DOWN(cache->object_size / 2, sizeof(void *));
freeptr_addr = (unsigned long)object + cache->offset;

```



Organisation des slabs et rôle du cache

Plusieurs Slab peuvent être liés entre eux, Il existe deux façons principales de les organiser :

- Liste simplement chaînée pour per cpu cache (un pointeur vers le slab suivant, un entier indiquant le nombre de slabs dans la liste)
- Liste Doublement chaînée pour les `pcache_per_node`

Revenons au cache :

Les slabs **per-CPU** sont liés à un CPU particulier.

Lorsqu'un CPU a besoin d'un objet, il regarde d'abord dans **sa propre liste per-CPU**, c'est seulement une question de performance (locking, CPU caches; etc).

Slab Actif :

`kmem_cache_cpu` possède **un slab actif**.

(Le nom réel est *CPU slab* en mode SLUB, mais on l'appelle ici *slab actif* pour simplifier.)

Un slab active a deux free lists (c'est une notion très importante pour comprendre pourquoi Slub est la norme aujourd'hui) :

Free list dans `kmem_cache_cpu` :

Quand le CPU veut un slot, il n'a **pas besoin de parcourir une liste**, Il utilise directement l'objet pointé par `Kmem_cache_cpu`

Free list dans le *slab actif* (`struct page`) :

stocké dans `struct page` du slab et représente **l'état réel et officiel** des slots libres du slab, Il est utilisé quand un slab est partagé, quand il quitte un cpu, ou quand il passe en partiel.

Synchronisation des freelists

Pendant les allocations —> seul le pointeur **CPU** est modifié (rapide, lockless)

Quand le slab est **abandonné ou synchronisé** —> la freelist CPU est **renversée** dans la freelist du slab (`struct page`)

Les partiels Slabs :

Ne possèdent **qu'une seule freelist**, Celle stockée dans la `struct page`, Sont utilisés pour les allocations **lorsque le slab actif est plein**.

Rôle de `kmem_cache_node` :

`kmem_cache_node` maintient une liste de **per-node partial slabs**.

Son rôle principal est de gérer une **liste de slabs partiellement utilisés** (*partial slabs*), ces ne sont **plus attachés à un CPU spécifique**, Chaque slab de cette liste possède sa **mémoire physique** et une **seule freelist**, stockée dans sa `struct page` qui décrit tous les objets libres du slab.

Tant que les **slabs per-CPU** ont des objets libres les allocations se font **sans lock**, localement, les allocations se font **sans lock**, localement un slab est **pris depuis la liste *partial* du nœud**, ce slab est alors **migré vers ce CPU** et devient son slab actif.

Limite de Large de ces listes :

Le nombre maximal de slabs partiellement utilisés conservés par CPU est **limité**, cette limite se voit ici dans la structure du `kmem_cache`, le `cpu_partial_slabs` ne peut pas être directement connu comme valeur, cependant nous avons `cpu_partial` il est lié à [ce calcul](#) dans `/sys/kernel/slab/$CACHE/cpu_partial`, et donc on peut déduire `cpu_partial_slabs`.

connaître ce nombre est intéressant pour certains slab attacks, comme la cross-cache attaque, qui utilise le overflowing de per_cpu_per_partial liste.

En **SLUB**, le nombre de slabs partiellement utilisés conservés **par CPU** est **volontairement limité** afin d'éviter une rétention excessive de mémoire. À l'inverse, SLUB impose un **minimum de slabs partiels conservés par nœud NUMA**.

voici un schéma qui englobe tout ce qu'on vient de voir :

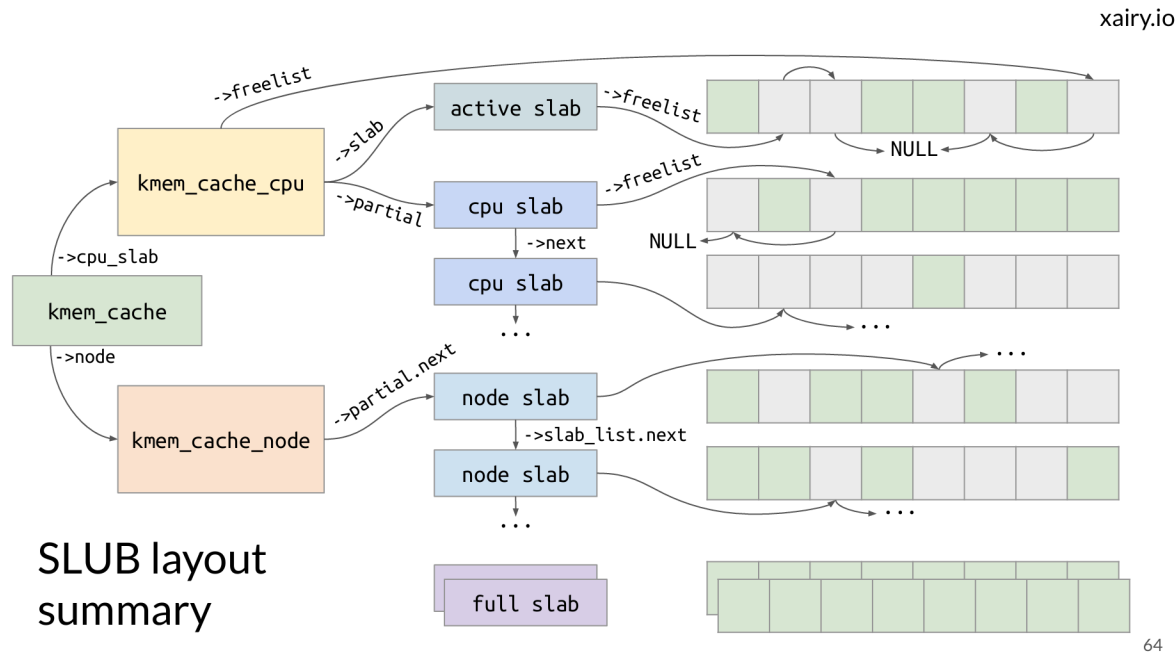


Figure x - Structure du SLUB - Ressource : xaiory

5- Processus d'allocation SLUB

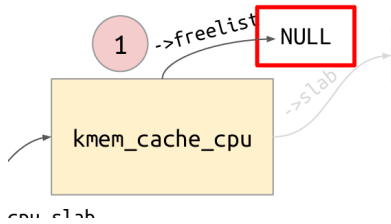
il ya 5 tiers de déroulement d'allocation :

pour l'allocation nous avons plusieurs cas, dependant de l'etat du slab au moment de l'allocation, voici ces cas :

1- CPU tente d'allouer un objet depuis un cache spécifique :

Premier choix : **per-CPU partial list (lockless)**, Si un slot est disponible → retourner le premier objet.

2- Si per-CPU freelist est vide :



Allocation depuis le **slab freelist** (avec lock)

3. Si slab freelist vide :

Prendre le premier slab de la **per-CPU partial slabs** et le rendre actif, ensuite on se retrouvera dans le cas 1 et 2.

4. Si per-CPU partial slabs vides : on fait de l'allocation depuis le `per_node`, on prend le premier slab dans la slab list et on l'assigne en active, mais en plus de ça nous allons bouger certaines `per_node` slabs a la `per_cpu` liste.

l'idée derrière c'est vu qu'on a run out de toute les méthodes d'allocations (inférieurs et donc moins coûteuse en timing et plus performante) il faut toute les restocker (le slab active avec une free list per cpu et une freelist (chaîne) et une partial list). on se retrouvera dans l'étape 1 et 2 !

5. Si aucun slab disponible : Allouer un **nouveau slab actif**, Quand un slab actif se remplit, un nouveau slab est assigné comme actif.

6- Processus de libération (Freeing)

Dans le processus de freeing on a plusieurs cas qui dépendent de quels objets on va libérer.

cas 1 : l'objet libéré appartient à l'active slab du cpu actuelle

Dans ce cas l'objet est mis dans le lockless per-cpu freelist.

<schéma a faire> minute 53.

Fréquent pour allocations/désallocations rapides (vu plus haut) utile pour attaques free-box.

case 2 : l'objet libre n'est pas dans l'active slab du current cpu et ce slab n'est pas full.

Objet ajouté à la freelist du slab, Si slab per-CPU ou actif d'un autre CPU : **ne jamais libérer le slab**, juste mettre à jour la freelist.

Ça arrive quand l'objet est alloué par le cpu A mais libéré par le cpu B.

La suite dépend du type du slab :

- **slab per-CPU ou active slab d'un autre CPU :**

on ne libère jamais la slab, on update uniquement la freelist.

- **slab per-node (global) :**

Si le slab n'est pas complètement vide, il reste dans la liste per-node.

Si il est complètement vide alors le slub se pose une seconde question : Est-ce que le node a déjà assez de slabs partiellement libres ?

si c'est non alors on garde le slab, si c'est oui alors on libère la slab entier, en la retirant de la liste per-node et en rendant les pages rendues aux pages allocator.

case 3 : l'objet libéré appartient à un full slab :

Objet ajouté à la freelist du slab → slab devient partiellement plein

promotion du slab : slub tente maintenant d'optimiser :

si Per-CPU partial list a de la place :

alors c'est mis en tête.

si Per-CPU partial list est pleine :

vide partiellement la per-CPU partial list

déplace certains slabs vers la per-node list

libère les slabs vides

ajouter le slab courant

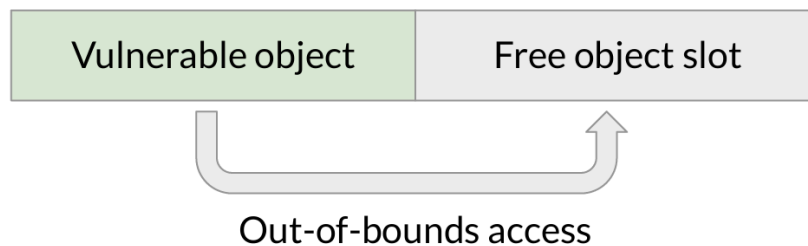
7- Exploitation du SLUB Allocator

Actuellement, dans le kernel Linux moderne, il n'existe plus que **SLUB** comme allocateur de slabs.

Cependant, si l'on cible un **ancien kernel**, il est toujours utile de connaître les anciens allocateurs et leur fonctionnement. Un **objet vulnérable** est un objet affecté par un certain type de vulnérabilité.

D'un point de vue exploitation, que peut-on faire ?

Dans le cas d'un **out-of-bounds access**, si le slot situé juste après l'objet vulnérable est **libre**, on peut accéder aux **métadonnées** de cet objet libre et donc les compromettre. Si cet objet n'est pas libre, l'exploitation devient beaucoup plus compliquée à mettre en œuvre. Ce cas sera expliqué plus loin.



Une seconde attaque possible consiste à **corrompre un autre objet**. On peut imaginer qu'un autre objet contient, par exemple, un **function pointer**. Si l'on est capable d'écraser ce function pointer via un out-of-bounds write, puis de faire en sorte que le kernel déclenche l'appel de ce function pointer, il devient alors possible de détourner le flot d'exécution du kernel. Cela peut mener, par exemple, à une **élévation de privilèges**. Cette approche est généralement beaucoup plus simple à exploiter.

Un autre objet peut également être utilisé pour d'autres objectifs, comme par exemple **leak l'adresse de vmlinux** afin de bypass **KASLR**, ou obtenir une primitive plus puissante comme une **Arbitrary Address Read** ou une **Arbitrary Address Write**, ce qui permet ensuite une élévation de privilèges. Spoiler : c'est une approche relativement simple et donc très souvent utilisée.

Pour réaliser une Slub Memory Corruption, il est nécessaire de réunir trois conditions.

1. Il faut disposer d'un **bug kernel** qui provoque un out-of-bounds access sur l'objet vulnérable. Nous n'aborderons pas dans ce document la manière de trouver un bug kernel ; nous partons du principe qu'un tel bug existe déjà.
2. il faut trouver un **objet cible**. Il existe de nombreux write-ups décrivant des objets cibles intéressants dans le kernel, sur lesquels il est possible de construire soit une exploitation, soit un leak d'informations.
3. Il est nécessaire de **façonner la mémoire SLUB** (heap grooming ou heap spraying).

Dans le cas d'un out-of-bounds, cela consiste à placer l'objet vulnérable et l'objet cible **l'un à côté de l'autre**.

Dans le cas d'un use-after-free, cela consiste à placer l'objet cible **dans le slot de l'objet vulnérable libéré**.

Il faut donc comprendre comment placer l'objet vulnérable et l'objet cible côte à côte, et dans le cas d'un use-after-free, comment faire en sorte que l'objet cible soit placé dans un slot référencé par une référence use-after-free.

Out-of-bounds exploitation :

Case 1 :

On suppose que l'on dispose de deux syscalls distincts. Le premier sert à déclencher l'allocation de l'objet vulnérable, et le second sert à déclencher la lecture ou l'écriture out-of-bounds sur cet objet vulnérable.

Par exemple :

IOCTL_ALLOC : alloue l'objet vulnérable

IOCTL_OOB : écrit ou lit des données en dehors des limites de l'objet vulnérable

L'algorithme utilisé pour l'exploitation est le suivant :

On commence par allouer **beaucoup, beaucoup d'objets** afin de déclencher la création d'un **nouvel active slab**. L'objectif est d'obtenir un slab contenant uniquement des objets que l'on contrôle, et avec lequel les autres CPU ne peuvent pas interagir (par exemple en le modifiant ou en le récupérant).

On a au préalable choisi notre **objet vulnérable** et notre **objet cible**, ce qui est nécessaire pour bien comprendre la suite.

La question centrale est alors la suivante : **combien d'objets doit-on allouer ?**

En utilisateur non privilégié, on ne peut pas le savoir directement. Les fichiers comme

`/proc/slabinfo` et les autres fichiers associés ne sont pas accessibles. De plus, il n'existe pas de limite supérieure fixe au nombre de slots libres dans un slab. Par conséquent, pour un exploit, il existe deux approches principales pour obtenir cette information.

La première approche consiste à faire une **estimation**. On reproduit localement l'environnement cible, ce qui permet d'accéder à `/proc/slabinfo` et donc d'obtenir une estimation du nombre de slots disponibles. On utilise ensuite ce nombre comme base (en général, on commence par le double) pour allouer suffisamment d'objets afin de déclencher un nouveau slab. Bien sûr, ce n'est pas une science exacte, et le système cible peut avoir un nombre de slots libres complètement différent.

La seconde approche repose sur une **analyse en timing**. De manière générale, le syscall d'allocation d'un objet prend à peu près le même temps à chaque appel. Dès que ce syscall commence à prendre significativement plus de temps, on peut en déduire qu'un **nouveau slab vient d'être alloué**. [source](#)

Dans `/proc/slabinfo`, le champ `active_objs` correspond au nombre d'objets actuellement alloués, tandis que `num_objs` correspond au nombre total de slots, incluant les slots libres et occupés.

Ces données ne sont pas mises à jour en temps réel. Il est donc nécessaire de forcer le SLUB allocator à les mettre à jour. Pour cela, on peut shrink le cache avec la commande suivante :

```
echo 1 | sudo tee /sys/kernel/slab/kmalloc-32/shrink
```

Le problème avec cette commande est que les slabs dont tous les objets sont libres quittent la liste des partial slabs. Cependant, comme les données n'étaient déjà pas à jour, cette mise à jour reste acceptable dans le cadre de l'analyse.

Une fois que l'on a une estimation du nombre de slots, on alloue suffisamment d'objets pour déclencher une **nouvelle active slab**.

Le but est alors d'allouer **un seul objet**, que l'on appelle l'objet *vulnérable*, dans ce nouvel active slab. Tous les autres objets alloués seront des objets *target*. De cette manière, on augmente fortement les chances que l'objet vulnérable soit suivi immédiatement par un objet cible, ce qui permet à l'exploit de fonctionner.

vulnerable object is now removed by target object.

Target	Target	Target	Vuln	Target	Target	Target	Target
--------	--------	--------	------	--------	--------	--------	--------

On peut ensuite déclencher l'accès out-of-bounds via l'appel système `IOCTL_00B`.

Cet exploit peut cependant échouer dans certains cas. Par exemple, si l'objet vulnérable est le **dernier objet du slab**, il ne sera pas possible de déclencher l'out-of-bounds.

Un autre cas d'échec est une **migration du processus** : si le processus de l'exploit est migré vers un autre CPU en pleine exécution, le comportement attendu peut être cassé. Pour éviter cela, on peut pinner le processus de l'exploit à un CPU spécifique en utilisant `sched_setaffinity`.

```
cpu_set_t my_set;
CPU_ZERO(&my_set);
CPU_SET(0, &my_set);
sched_setaffinity(0, sizeof(my_set), &my_set)
```

Case 2 :

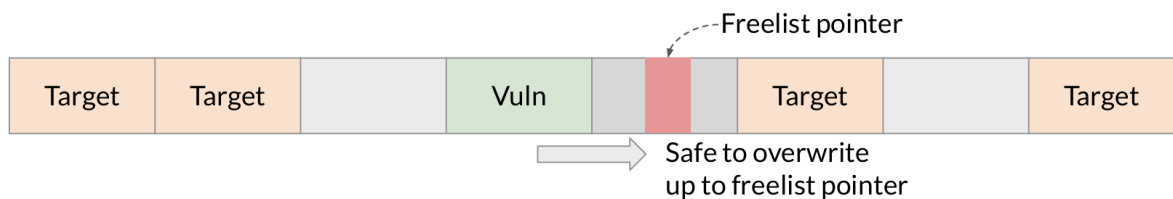
On dispose d'un seul syscall qui réalise les deux opérations en même temps, via l'appel système `IOCTL_ALLOC_AND_OOB`. Dans ce cas, l'allocation de l'objet vulnérable et le déclenchement de l'accès out-of-bounds ne peuvent pas être séparés.

On commence donc par allouer suffisamment d'objets *target*, puis on passe à l'allocation de l'objet vulnérable, ce qui déclenche en même temps l'out-of-bounds.

Dans ce cas précis, on peut tomber sur une situation particulière liée à la structure de SLUB.

Il est possible que l'accès out-of-bounds tombe sur le pointeur de la freelist, qui est stocké au milieu du slab. Dans ce cas, tant que le pointeur n'est pas corrompu (out-of-bounds de petite taille), on est en sécurité. En revanche, si le pointeur est modifié, cela provoque **un crash du kernel**.

Dans le cas plus général, où l'objet vulnérable est suivi d'un objet *target*, l'exploitation reste possible, mais elle peut nécessiter plusieurs tentatives.

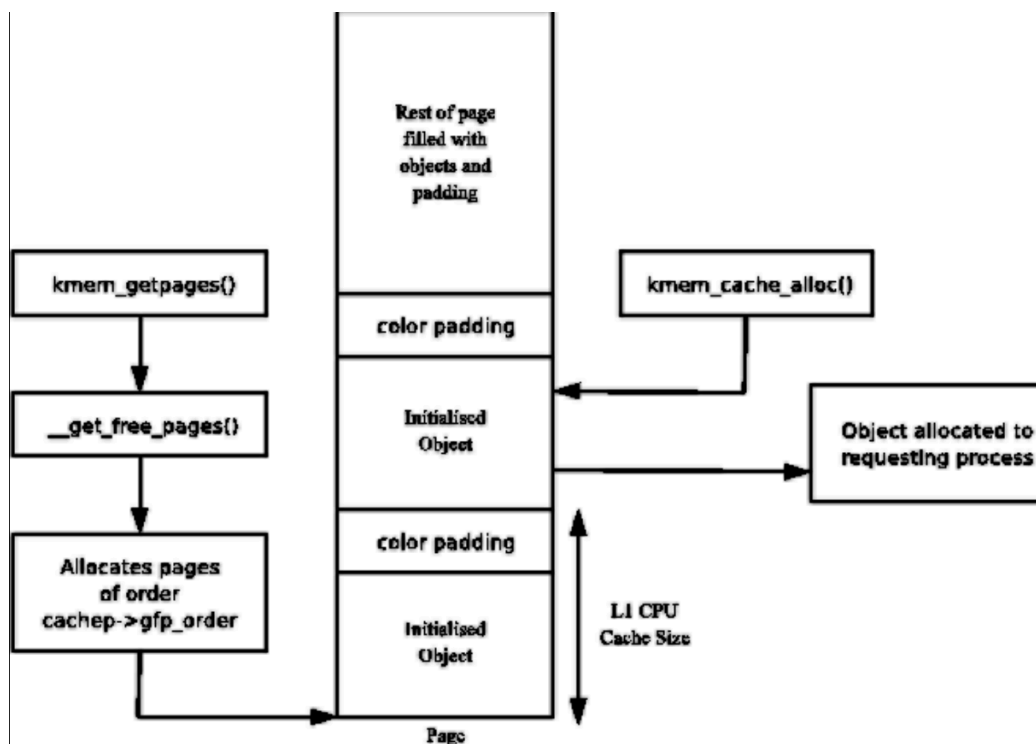


8- UAF Exploit

Un use-after-free (UAF) survient lorsqu'un objet est libéré alors qu'une référence vers celui-ci est toujours utilisée. Dans le contexte du SLUB allocator, l'exploitation repose sur la réutilisation du slot mémoire d'un objet libéré.

Le principe est le suivant : après la libération de l'objet vulnérable, son slot est ajouté à la liste libre du slab. Lorsqu'un nouvel objet de même taille est ensuite alloué, il est très probable qu'il soit placé dans ce slot libéré, en raison du fonctionnement en pile (LIFO) des freelists. La référence use-after-free pointe alors vers ce nouvel objet.

L'attaquant peut exploiter cette situation pour lire ou modifier les données de l'objet cible, ce qui peut permettre des fuites d'informations ou, dans certains cas, un détournement du flot d'exécution du noyau.



VI- Conclusion

Le SLAB allocator joue un rôle fondamental dans la gestion de la mémoire dynamique du noyau Linux. Conçu pour répondre aux contraintes spécifiques du kernel, il repose sur une organisation en caches, slabs et objets de taille fixe, permettant des allocations rapides tout en limitant la fragmentation mémoire.

À travers ce document, nous avons détaillé les principes de fonctionnement du SLAB allocator, depuis ses concepts de base jusqu'à son organisation interne. L'étude des structures telles que les caches, les slabs et les freelists met en évidence les choix de conception visant à optimiser les performances, la localité mémoire et la réutilisation efficace des objets.

Nous avons également montré comment ces mécanismes influencent directement le comportement du noyau, tant du point de vue de l'allocation que de la libération de la mémoire. La compréhension fine du SLAB allocator permet ainsi de mieux appréhender les compromis réalisés entre performance, consommation mémoire et complexité de gestion.

Enfin, l'analyse détaillée du SLAB allocator constitue une base essentielle pour comprendre l'évolution vers des implémentations plus modernes comme SLUB. Ce travail met en évidence l'importance des allocateurs de type slab dans le noyau Linux et souligne la nécessité d'en maîtriser le fonctionnement interne pour toute étude approfondie des systèmes d'exploitation.

