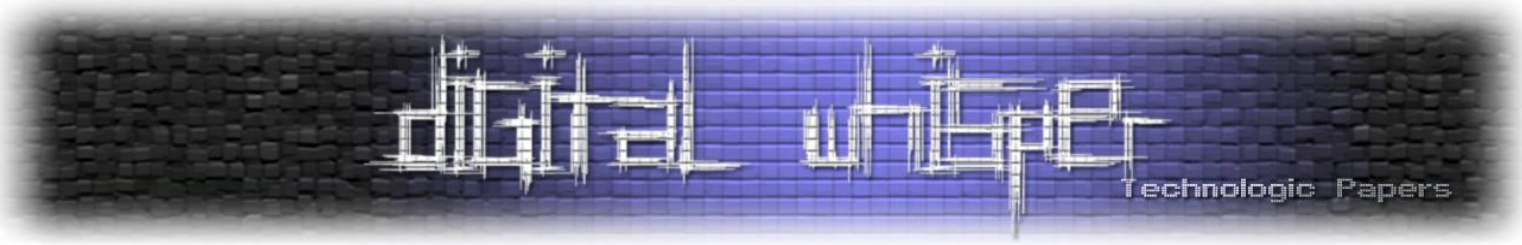


חלק א' - BabySteps into SLUB

מאת נועם אפרגן

תודות

תחילה ארצה להודות וגם לתת קרדיט ל-KonovalovAndrey שנתן לי את הסכמתו להשתמש בתרשימים שהוא יצר עבור ההרצאה שלו SLUB Internals for Exploit Developers כדי שאוכל להשתמש בהם לאורך המאמר ולייצר הסברים נוחים להבנה - Thank you andrey!

הקדמה - זיכרון דינאמי

ישנם כמה סוגי זיכרון שהמערכת יודעת לספק לתהליכים שרצים עליה, כשהבולטים ביניהם הם המחסנית (stack) והערימה (heap). כל אחד מהם נועד למלא תפקיד שונה ולכל אחד יתרונות וחסרונות ביחס לשני, לדוגמה המחסנית מחייבת אותנו להגדיר מראש בקוד את גודל הזיכרון שנרצה להקצות כך שהוא יהיה ידוע כבר בזמן הקומפילציה ולעומתה הערימה מאפשרת להקצות זיכרון גם כאשר הגודל שלו אינו ידוע מראש אלא נקבע רק בזמן הריצה - מה שנקרא שימוש בזיכרון דינאמי.

באופן מקביל לזיכרון הדינאמי שהמערכת מספקת לתהליכים הרצים עליה יש לה מימוש לזיכרון דינאמי עבור הקרנל כך שגם הקרנל יוכל להקצות זיכרון עבור אובייקטים באותה צורה, המימוש של ההיפ הוא ייחודי עבור הפרוססים בלבד ואינו דומה למימוש שהקרנל נעזר בו עבור ההקצאות הדינאמיות שלו, הסיבה לכך היא התוצאה של השיקול בין יעילות ההקצאות והמהירות שלהן לבין ה"בטיחות" שלא יהיו שגיאות שיוכלו לגרום לקריסה של התהליך או המערכת כולה בגלל שגיאות מפתח, קריסה של הקרנל בגלל שגיאת זיכרון היא נזק הרבה יותר דרסטי מאשר פרוסס יחיד שקורס ואיטיות של ההקצאות בקרנל גורמות לאיטיות המערכת כולה לעומת איטיות של הקצאות עבור פרוסס כלשהו שתגרום אך ורק לו להיות איטי ללא כל השפעה על שאר הפרוססים. הרכיב האחראי לספק את ההקצאות הדינמיות האלו בקרנל נקרא ה-Slab Allocator ולו יש כמה מימושים שונים ואנו נעסוק באחד מהם. בחלק הזה של המאמר בחלק הזה של המאמר נתמקד בבסיס החשוב להבנה כיצד ה-Slab Allocator עובד במימוש SLUB והוא ישמש אותנו בין היתר כרפרנס בחלקים הבאים שבמהלכם נראה אף ניצולים מעניינים שיכולים להיווצר משימוש שגוי בו, קריאה מהנה!

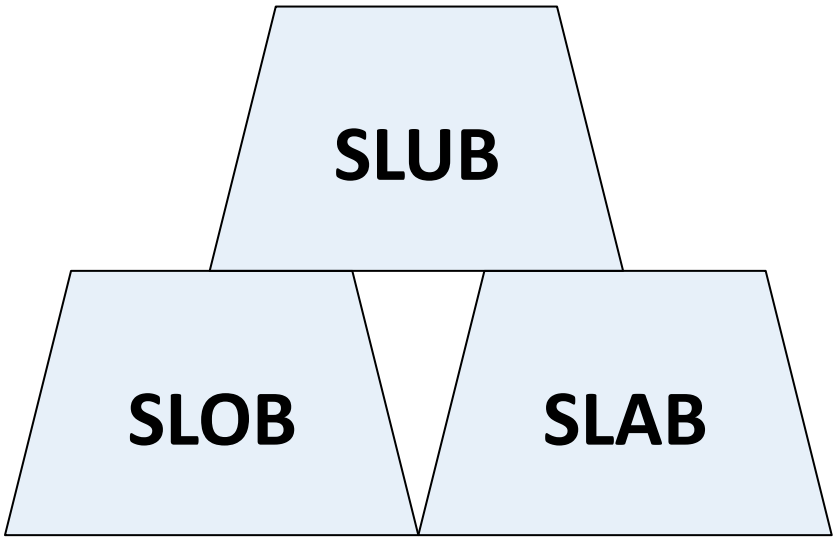
הקדמה - Slab Allocator

ישנם שלושה מימושים של Slab Allocators:

1. SLAB - הוסר בקרנל גרסה 6.8.
2. SLOB - מימוש פשוט (מוקטן) של SLUB, הוסר בקרנל גרסה 6.4 אך עדיין קיים במערכות מסוימות וניתן להשתמש בו באמצעות קינפוג וקימפול מחדש של הקרנל.
3. SLUB - שעליו נדבר.

כל ה-Slab Allocators הינם בעלי אותו API כלומר הם מספקים את אותן פונקציות בעבור המפתחים כך שהשוני ביניהם מתבטא במימוש שלהם בלבד.

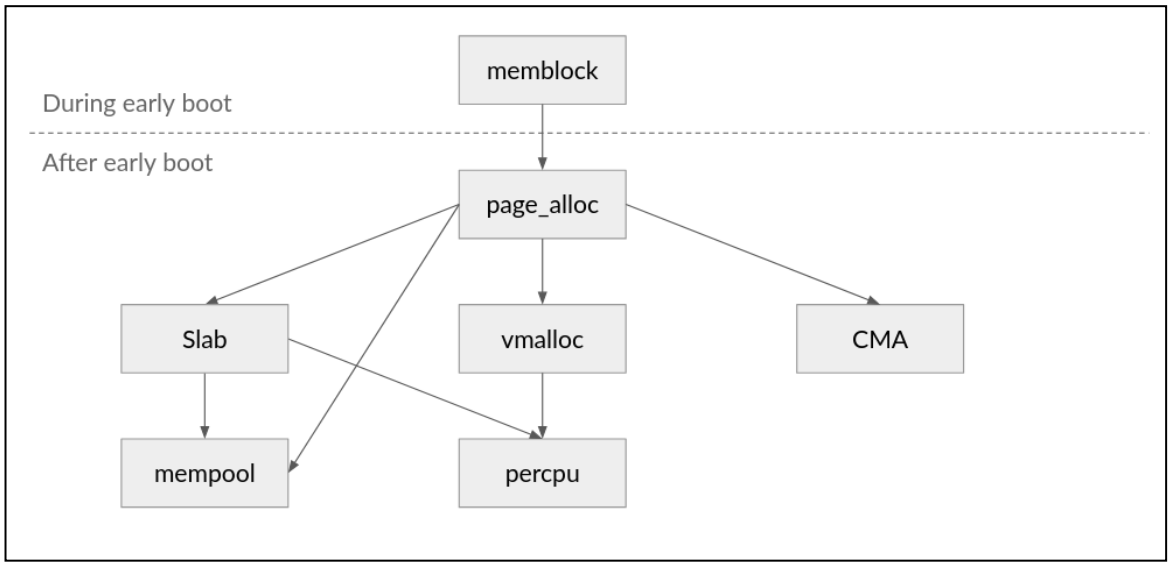
הבהרה: כאשר אני מזכיר\s Slab Allocator אני מתכוון לכלל המימושים SLUB, SLAB ו-SLOB אך כאשר ארצה להזכיר מימוש ספציפי הוא יוזכר באותיות גדולות וככה ימנע בלבול מיותר. הסיבה שנדבר דווקא על SLUB היא משום שברוב המקרים בהם יהיה עלינו להתעסק עם זיכרון קרנלי של מערכות לינוקס המימוש של מנהל הזיכרון איתו נעבוד יהיה מימוש זה.



הקדמה - system allocators hierarchy

לפני שניגש למימוש של ה-Slab Allocator נבין באילו מנהלי זיכרון - "אלוקאטורים" המערכת משתמשת כדי לנהל את הזיכרון עוד לפני שהוא מונגש לידיים שלו.

התבוננו בתרשים הבא:



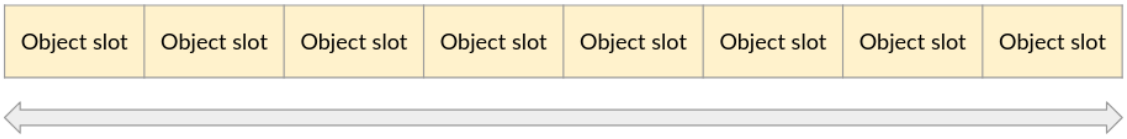
[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

כפי שניתן לראות ישנם שני אלוקאטורים תחת ה-Slab Allocator בשני שלבים שונים של המערכת:

- **memblock** - לפני העליה של המערכת עצמה, בתהליך ה-boot ומשמש אותה בשלביה הראשונים לצורך שמירה של טבלאות ומבני זיכרון לפי צורך.
- **page_alloc** - מספק הקצאה ושחרור דפים מהזיכרון הפיזי לאחר עליית המערכת.
- **Buddy Allocator** - ה-Buddy Allocator הוא מעין wrapper מעל page_alloc ומשמש לניהול של בלוקים (בלוק = רצף דפים בזיכרון) ויוצר ניהול יעיל ומחושב יותר של הקצאות דפים גדולות, ה-Buddy Allocator מחזיק רשימות של בלוקים בגדלים של 2^n*4096, לדוגמה הרשימה החמישית שלו תהיה עבור בלוקים בגודל 32 דפים כי $2^5 = 32$ והשישית 64 דפים כי $2^6 = 64$ וכן הלאה. למרות היתרונות שלו יש לו חיסרון אחד בולט מאוד, ננסה להבין מה יקרה אם למשל נרצה בלוק של 33 דפים? נהיה חייבים להקצות אותו מהרשימה ה-6 שהיא של 64 דפים, אך מתוכם נשתמש רק ב 33 ולכן זה יהיה בזבוז של 31 דפי זיכרון... ומה יקרה עבור 1025, 2049 או 4097 דפים? אותו דבר והבזבוז רק יגדל, זו היא עוד אחת מהבעיות אותן ה-Slab Allocator פותר ומאפשר ניהול יעיל יותר של הזיכרון.

מושגים - slab regions

slabs הינם אזורי זיכרון שה-Slab Allocator משתמש בהם כדי לנהל גדלים שונים של אובייקטים, כדי ליצור slab חדש ה-Slab Allocator מבקש מה-buddy allocator כמות דפים בהתאמה לכמות וגודל האובייקטים שה-slab החדש יצטרך לנהל.



One slab with 8 object slots of the same size

[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

מושגים - caches

למעשה כאשר אנו משתמשים בהקצאה או שחרור של קטעי זיכרון דינאמי בקרנל אנו ניגשים מאחורי הקלעים ל-caches והם מנהלים עבורנו את התהליכים הללו.

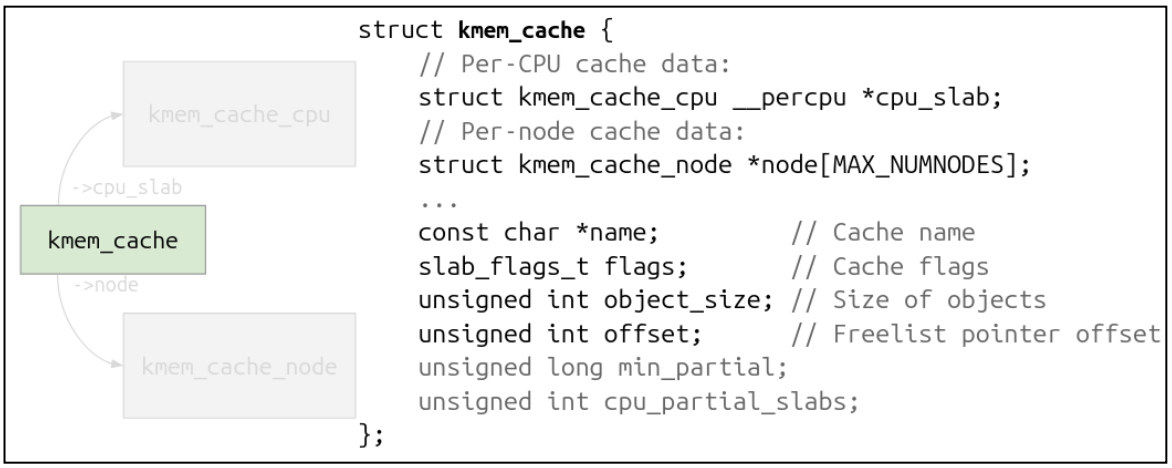
ישנם שני סוגי cache איתם ניתן לעבוד:

- **dedicated cache** - זהו cache שנוצר על פי בקשה של מודול קרנלי כדי לנהל slabs עבור הקצאות של אובייקטים מגודל \ סוג מסוים, לרוב מודולים משתמשים בסוג הזה כאשר יש להם אובייקט שההקצאה שלו חוזרת על עצמה והם רוצים לנהל אותו בצורה יעילה יותר או שיש להם סיבה לשמור על אובייקטים מסוימים באופן מבודד משאר הזיכרון מטעמי אבטחה, ניתן ליצור cache כזה באמצעות הפונקציה kmem_cache_create() שמחזירה מצביע ל-cache ולאחר מכן יהיה ניתן להקצות ממנו אובייקטים עם הפונקציה kmem_cache_alloc() ולשחרר אובייקטים עם kmem_cache_free().

- **generic caches** - אלו הם caches שנוצרים באופן דיפולטיבי על ידי הקרנל עבור גדלים שונים של אובייקטים כדי לנהל הקצאות דינאמיות של קטעי זיכרון עבור הקרנל ללא צורך ב-cache ייחודי, כלומר הם משותפים לכלל המודולים וכל אחד מהם יכול להקצות אובייקטים דרכם, ניתן להשתמש בהם באמצעות הפונקציות kmalloc() לצורך הקצאת זיכרון ו-kfree() לצורך שחרור.

מבנים ב-SLUB - מבנה ה-cache

כעת נביט במבנה ה-cache במימוש של SLUB ובשדות שלו:

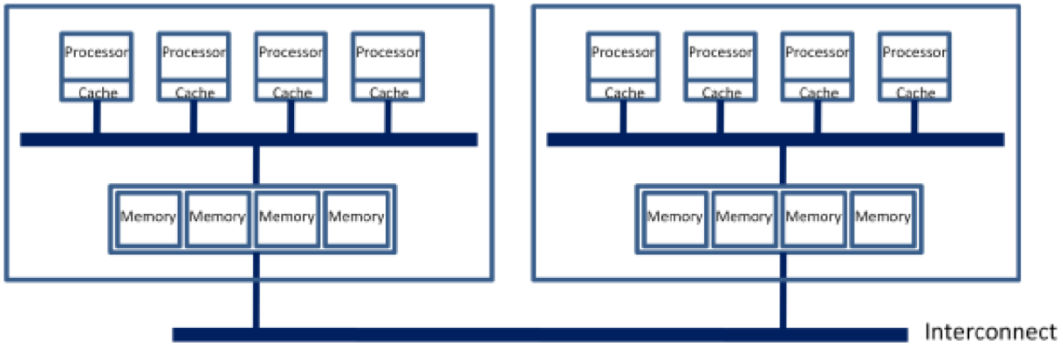



```
struct kmem_cache {
    // Per-CPU cache data:
    struct kmem_cache_cpu __percpu *cpu_slab;
    // Per-node cache data:
    struct kmem_cache_node *node[MAX_NUMNODES];
    ...
    const char *name;          // Cache name
    slab_flags_t flags;        // Cache flags
    unsigned int object_size; // Size of objects
    unsigned int offset;       // Freelist pointer offset
    unsigned long min_partial;
    unsigned int cpu_partial_slabs;
};
```

[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

ה-cache מחזיק פרטים גנריים כמו שם ה-cache, דגלים, גודל האובייקטים אותם הוא מנהל וכו'... לפי שני המבנים הראשונים שהוא מחזיק ניתן לראות שהוא יודע לשרת בקשות זיכרון עבור כל מעבד בנפרד "per-CPU" ויודע לנהל slabs עבור כל קבוצת מעבדים "per-Node" במידה וקיים זיכרון משותף - "NUMA" עבור כל קבוצה כזו.

<u>הרחבה:</u> NUMA \ Non-Uniform Memory Access - בהנחה שיש לנו ארכיטקטורה עם מספר רב של מעבדים וכולם צריכים לעבוד עם אותו הזיכרון, יכול להיווצר לנו צוואר בקבוק של בקשות זיכרון מכל המעבדים וכתוצאה מכך נקבל עבודה איטית יותר, את הבעיה הזו ניתן לפתור באמצעות חלוקה של המעבדים לקבוצות - Nodes וכל קבוצה כזו מקבלת איזור זיכרון משותף אליו רק[1] המעבדים השייכים לאותה קבוצה יכולים לגשת, מה שמקצר באופן משמעותי את הזמן שלוקח למעבד לגשת לזיכרון הפיזי עבור כל צורך שהוא. (להסבר מעמיק יותר קראו את המסמך הבא).



[מקור: https://www.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf]

[1] ישנה דרך לבקש זיכרון גם מ-nodes אחרים אך בקשה שכזו עלולה לעלות בעוד זמן בפער מבקשת זיכרון מה-node הנוכחי בו המעבד נמצא ולכן המעבד יעדיף להימנע מכך ברוב המקרים.

הצגת נתונים - caches

כדי להציג אילו caches יש לנו ולראות פרטים נוספים נציג את תוכן הקובץ "/proc/slabinfo":

```
[~]
✘ noam sudo cat /proc/slabinfo
[sudo] password for noam:
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit>
 <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
```

קיצרתי חלק מהפלט כך שיהיה ניתן לראות את שני סוגי ה-cache עליהם דיברנו:

```
files_cache          368    368    704   23    4 : tunables    0    0    0 : slabdata     16     16      0
signal_cache         718    812   1152   28    8 : tunables    0    0    0 : slabdata     29     29      0
sighand_cache        551    690   2112   15    8 : tunables    0    0    0 : slabdata     46     46      0
task_struct         1781   1788  12096    2    8 : tunables    0    0    0 : slabdata    894    894      0
cred_jar           87799  88704    192   21    1 : tunables    0    0    0 : slabdata   4224   4224      0
anon_vma_chain     45765  46208     64   64    1 : tunables    0    0    0 : slabdata    722    722      0
anon_vma           25818  26052    104   39    1 : tunables    0    0    0 : slabdata    668    668      0
pid                 3165   3360    128   32    1 : tunables    0    0    0 : slabdata    105    105      0
Acpi-ParseExt        312    312    104   39    1 : tunables    0    0    0 : slabdata      8      8      0
Acpi-State          3825   3825     80   51    1 : tunables    0    0    0 : slabdata     75     75      0
shared_policy_node 1748258 1775310     48   85    1 : tunables    0    0    0 : slabdata  20886  20886      0
numa_policy           30     30    272   30    2 : tunables    0    0    0 : slabdata      1      1      0
perf_event           200    200   1280   25    8 : tunables    0    0    0 : slabdata      8      8      0
trace_event_file    3150   3150     96   42    1 : tunables    0    0    0 : slabdata     75     75      0
ftrace_event_field  8906   8906     56   73    1 : tunables    0    0    0 : slabdata    122    122      0
pool_workqueue      1312   1312    512   32    4 : tunables    0    0    0 : slabdata     41     41      0
maple_node         10367  11328    256   32    2 : tunables    0    0    0 : slabdata    354    354      0
radix_tree_node    58077  58128    584   28    4 : tunables    0    0    0 : slabdata   2076   2076      0
task_group          1526   1725    640   25    4 : tunables    0    0    0 : slabdata     69     69      0
mm_struct            391    391   1408   23    8 : tunables    0    0    0 : slabdata     17     17      0
vmap_area          33152  33152     72   56    1 : tunables    0    0    0 : slabdata    592    592      0
kmalloc-cg-8k         68     68   8192    4    8 : tunables    0    0    0 : slabdata     17     17      0
kmalloc-cg-4k        386    400   4096    8    8 : tunables    0    0    0 : slabdata     50     50      0
kmalloc-cg-2k        985    992   2048   16    8 : tunables    0    0    0 : slabdata     62     62      0
kmalloc-cg-1k        770    896   1024   32    8 : tunables    0    0    0 : slabdata     28     28      0
kmalloc-cg-512       886    960    512   32    4 : tunables    0    0    0 : slabdata     30     30      0
kmalloc-cg-256       384    384    256   32    2 : tunables    0    0    0 : slabdata     12     12      0
kmalloc-cg-192       777    777    192   21    1 : tunables    0    0    0 : slabdata     37     37      0
kmalloc-cg-128       480    480    128   32    1 : tunables    0    0    0 : slabdata     15     15      0
kmalloc-cg-96       1129   1260     96   42    1 : tunables    0    0    0 : slabdata     30     30      0
kmalloc-cg-64       1029   1280     64   64    1 : tunables    0    0    0 : slabdata     20     20      0
kmalloc-cg-32       2019   2176     32  128    1 : tunables    0    0    0 : slabdata     17     17      0
kmalloc-cg-16       9472   9472     16  256    1 : tunables    0    0    0 : slabdata     37     37      0
kmalloc-cg-8        4096   4096      8  512    1 : tunables    0    0    0 : slabdata      8      8      0
dma-kmalloc-8k         0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-4k         0      0   4096    8    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-2k         0      0   2048   16    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-1k         0      0   1024   32    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-512        0      0    512   32    4 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-256        0      0    256   32    2 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-192        0      0    192   21    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-128        0      0    128   32    1 : tunables    0    0    0 : slabdata      0      0      0
```

מהפלט של הקובץ, ניתן לראות את אלו שנוצרו עבור סוגי אובייקטים ספציפיים, כגון files_cache, task_struct, mm_struct... ואלו שנוצרו בצורה גנרית בשביל kmalloc עבור הקצאות בגדלים שונים.

ישנם פרטים נוספים שאנו יכולים לזהות כגון num_objs \ active_objs המשמשים כדי לדעת כמה אובייקטים זמינים ומוקצים יש בו סה"כ וכן גם objsize שמורה מה הוא גודל האובייקטים ב-cache.

נוסף על כך ה-cache מחזיק נתונים גם על ה-slabs שהוא מנהל, כמה אובייקטים יהיו בכל slab, מה יהיה גודל הבלוק (רצף הדפים) של כל slab וכן גם כמה slabs אותו cache מכיל.

מבנים ב-SLUB - מבנה ה-slab

לפני שניגע כיצד ה-cache מנהל את הגישה ל-slabs עבור כל מעבד[2] וכיצד הוא מקצה אובייקטים מתוכו נבין כיצד נראה המבנה של slab:

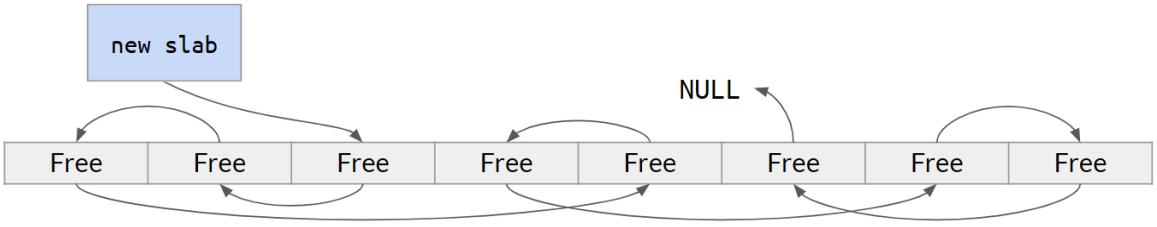
```
struct slab {
    struct kmem_cache *slab_cache; // Cache this slab belongs to
    struct slab *next;             // Next slab in per-cpu list
    int slabs;                     // Slabs left in per-cpu list
    struct list_head slab_list;    // List links in per-node list
    void *freelist;                // Per-slab freelist
    ...
};
```

[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

השדה הראשון במבנה - slab_cache הינו מצביע ל-cache אליו אותו ה-slab משויך, שני השדות הבאים הינם - next מצביע למבנה slab נוסף כחלק מרשימה מקושרת ואחריו גם מונה - slabs של כמות ה-slabs באותה הרשימה, השימוש בהם נעשה כאשר ה-slab מופיע כחלק מרשימה של slabs הכלולה במבנה הנקרא kmem_cache_cpu (נדבר עליו בהמשך) הנוצר ב-cache עבור כל מעבד לצורך ניהול ה-slabs השייכים לו.

בנוסף קיים השדה - slab_list שהוא רשימה מקושרת דו-כיוונית ורשימה זו משמשת כאשר ה-slab נמצא תחת ניהול ה-node, במילים אחרות היא נכנסת לשימוש רק כאשר ה-slab נכנס כחלק ממאגר ה-slabs המשותף לכל קבוצת המעבדים באותו NUMA node המוחזק בתוך המבנה kmem_cache_node (יוזכר בהמשך) ומנוהל בתור רשימה מקושרת דו כיוונית, שימוש ברשימה דו-כיוונית מאפשר גישה יעילה של הוספה והסרה מהירה של slabs בהתאם לצרכים של כל מעבד ב-node.

השדה האחרון הינו רשימה מקושרת פנימית של ה-slab ה-"free list" שעוקבת אחרי כל האובייקטים שאינם מוקצים או הוקצו ושוחררו מתוך אותו ה-slab כך שכאשר נרצה להקצות אובייקט מתוך ה-slab נוכל לגשת אליה ולקחת את האובייקט הראשון. כאשר slab חדש נוצר כלל האובייקטים מעורבבים ברשימה כך שהיא אינה מאורגנת אף פעם מטעמי אבטחה[3] וכאשר אובייקט משוחרר הוא נכנס ישירות לתחילת הרשימה.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

[2] גם עבור כל "מעבד לוגי" או בשם אחר "ליבה".
[3] מקשה על תוקפים להקצות \ לשחרר אובייקטים פגיעים בצורה מסודרת שתקל על מימוש המתקפה שלהם.

slab - הרשימה free list

מטעמי אבטחה ה-free list של ה-slab בנויה באופן מורכב יותר מרשימה מקושרת קלאסית, המצביע לאובייקט הבא מהאובייקט הנוכחי ברשימה נמצא בקירוב למרכז שלו לפי החישוב הבא:

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

[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

כאשר דיברנו על מבנה ה-cache[4] השמטנו את הפרט שאחד השדות שלו הוא אופסט השייך לרשימות הקשורות ל-slabs אותם הוא מחזיק, אז הרשימות האלו הן ה-free lists של ה-slabs השייכים לאותו ה-cache וההיסט בעצם אומר באיזה מרחק מתחילת כל אובייקט נוכל למצוא את המצביע לאובייקט הבא ברשימה, הסיבה שיש את ההגנה הזו והמצביע לאובייקט הבא לא נמצא פשוט בתחילת כל אובייקט היא כדי למנוע מבאגים פשוטים[5] לדרוס לנו את המצביע לאובייקט הבא ברשימה וככה אולי אפילו להצליח להקריס לנו את הקרנל ואת המערכת כולה. החישוב של האופסט הוא פשוט - חלוקה של גודל האובייקט לחצי ולאחר מכן יישור שלו (בקירוב) כלפי מטה בגודל של מצביע יחיד באותה מערכת (8 ב-64 ביט ו-4 ב-32 ביט), לאחר מכן ניתן להוסיף את הערך שנוצר למצביע של אובייקט כלשהו מתוך הרשימה ולקבל מתוכו את המצביע לאובייקט הבא.

ישנה הגנה נוספת על המצביעים של ה-free list, כל מצביע לאובייקט הבא ברשימה מוצפן מה שמקשה עוד יותר על הדלפת \ דריסת המצביעים ברשימה הזו:

```
cache->random = get_random_long();
freelist_ptr = (void *)(
    (unsigned long)ptr ^ cache->random ^ swab(ptr_addr));
```

[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

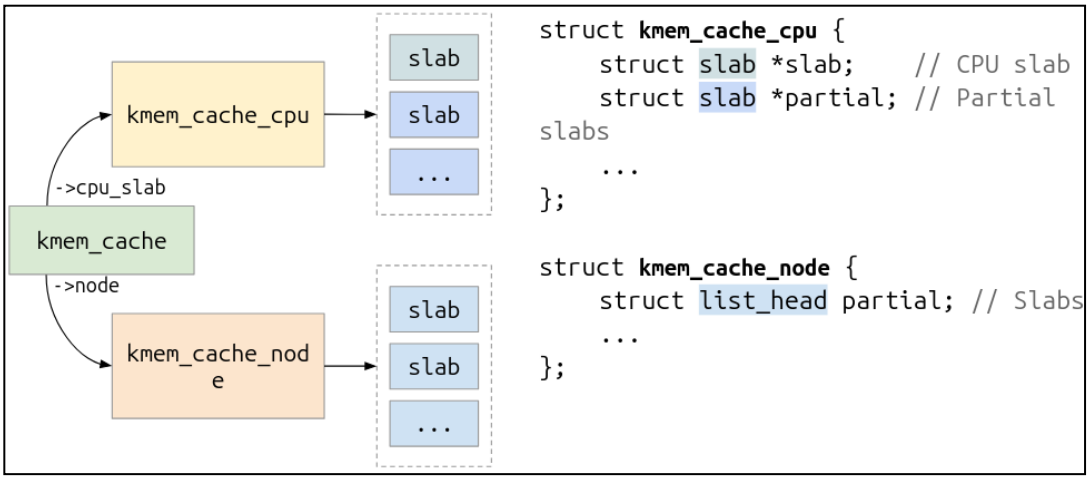
לאחר שקיבלנו את האופסט מהמבנה של ה-cache והשגנו את הערך המופיע באובייקט כחלק מה-free list עלינו לבצע עליו שתי פעולות xor, האחת עם ערך רנדומלי שלא הזכרנו גם אותו לפני כן כאשר דיברנו על ה-cache, הערך הזה נוצר בשלבי האתחול הראשונים של SLUB בקרנל ומשמש להצפנה של כלל המצביעים ב-free list בכל ה-caches.

הערך השני מיוחד לכל רשימה בפני עצמה משום שהוא הכתובת של ה-slab עצמו אליו היא שייכת, לאחר ביצוע xor עם הערכים הללו נקבל את המצביע לאובייקט הבא ברשימה.

[4] בותרת SLUB 2.1 - מבנה ה-cache.
[5] לדוג' Off by one או פרימיטיבים כמו BOF אך חלשים מכדי להגיע למרכז האובייקט.

מבנים ב-SLUB - kmem_cache_cpu & kmem_cache_node

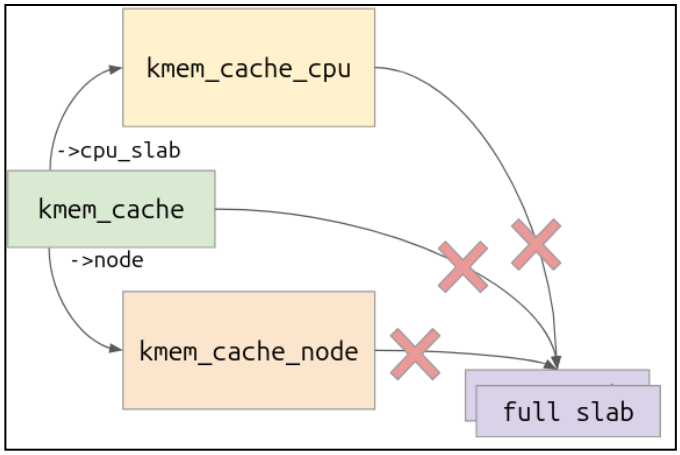
קודם לכן הזכרנו שה-cache יודע לשרת בקשות זיכרון עבור כל מעבד בנפרד, הוא עושה זאת באמצעות רשימה מקושרת של מבני kmem_cache_cpu, כדי להשיג את אותם slabs עבור המעבד הוא יודע לבקש אותם מה-Node בו המעבד חבר בעזרת המבנה kmem_cache_node. כעת נביט על שני המבנים הללו וכיצד הם מחזיקים את אותם slabs ומנהלים אותם וכן גם כיצד הם מעבירים אותם ביניהם:



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

kmem_cache_cpu - המבנה מחזיק שני שדות עבור ה-slabs בהם הוא משתמש כדי לספק את אלוקציות הזיכרון עבור הבקשה של המעבד:

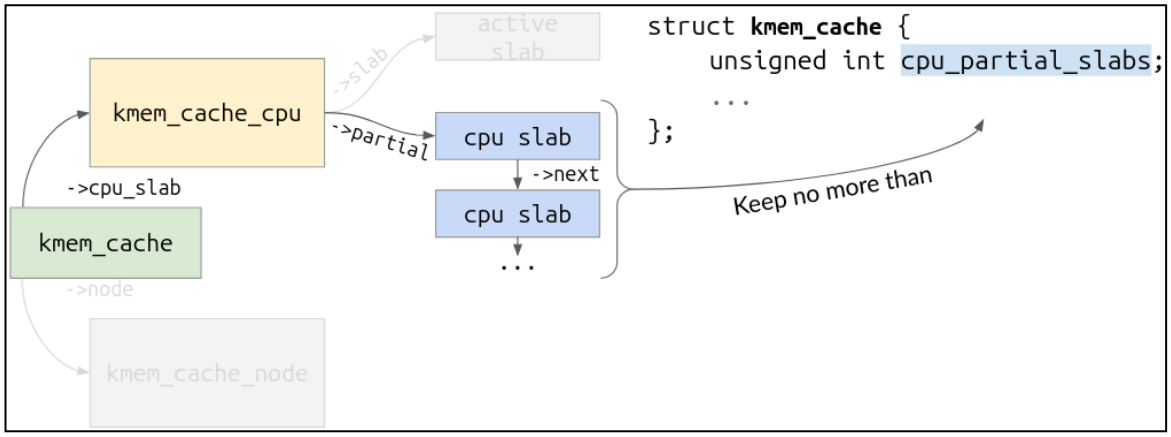
1. Slab - מצביע ל-slab בודד ולא לרשימה, ידוע גם בתור ה-CPU slab שבו ה-cache משתמש כעת ומספק אובייקטים מתוך ה-free list שלו לפני שהוא משתמש ב-slabs אחרים.
2. Partial - מצביע לרשימה מקושרת של slabs שה-cache יכול להשתמש בהם כאשר ה-free list של ה-CPU slab כבר ריק, בפשטות הוא יקח slab מהרשימה הזו של ה-partial slabs ויכניס אותו במקום ה-CPU slab, הרשימה הזו נקראת partial משום שחלק מה-slabs שהיא מכילה כבר שומשו בעבר ובוצעו מהם אלוקציות זיכרון ולכן ה-free list שלהם לא בהכרח מצביע לכל האובייקטים אותם הם מכילים ולכן הם חלקיים.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

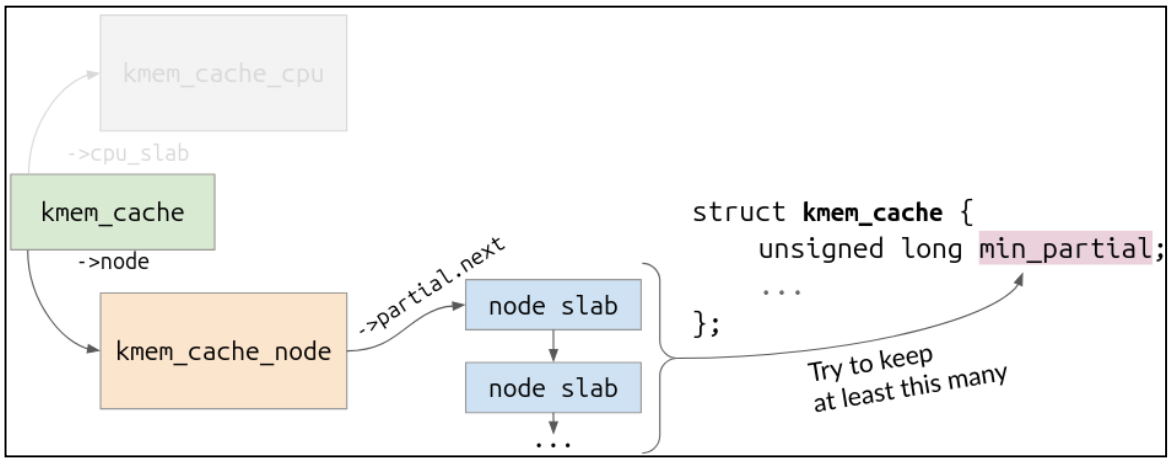
<u>הערה:</u> כאשר slab מתמלא באובייקטים מאולקצים וה-free list מתרוקן ונעשה ריק אף אחד מהמבנים של ה-cache לא מחזיק תיעוד על קיומו של ה-slab עד לרגע שבו אחד האובייקטים שבו ישוחרר באמצעות kfree ורק אז הוא יחזור לידיים של ה-Slab Allocator.

kmem_cache_node - המבנה מנהל רשימה דו כיוונית של slabs עבור כלל המעבדים החברים באותו ה-Node, הרשימה הזו מחזיקה slabs שלא נעשה בהם שימוש בכלל וגם כאלו שנעשה בהם שימוש חלקי בדומה לרשימה partial של המבנה kmem_cache_cpu ותפקידה הוא להעביר אליה slabs במידה והיא התרוקנה או לקבל ממנה slabs במידה והיא מכילה יותר מדי, המקסימום slabs שה-partial list יכולה להכיל מוגדר במבנה הראשי של ה-cache כלומר ב-kmem_cache ולפיו ה-cache מחליט אם יש יותר או פחות מדי slabs ב-partial list.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

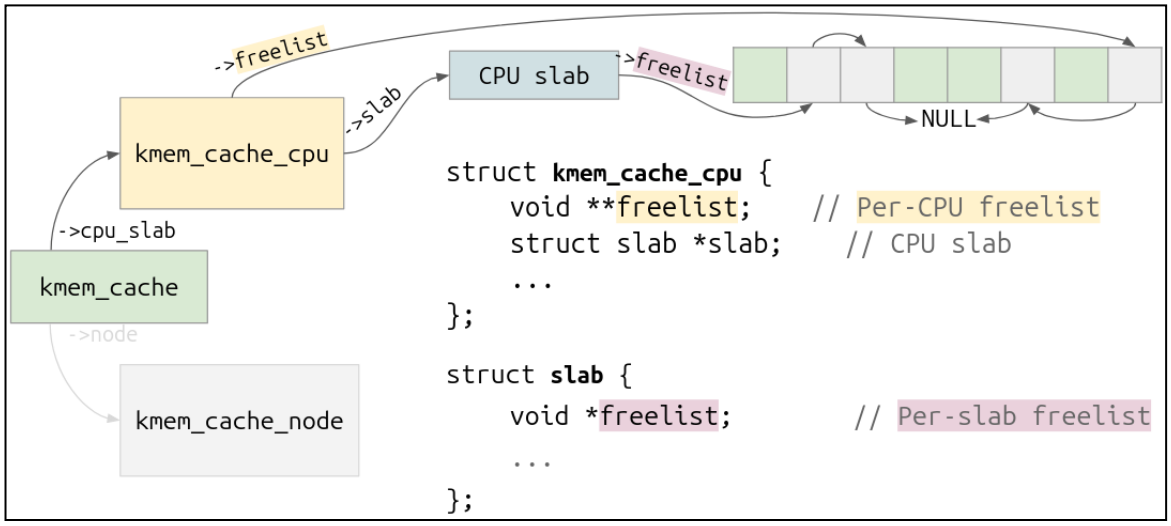
ה-kmem_cache גם מחזיק את הכמות slabs המינימלית שה-kmem_cache_node צריך לשמור ברשימה של ה-slabs שהוא מחזיק.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

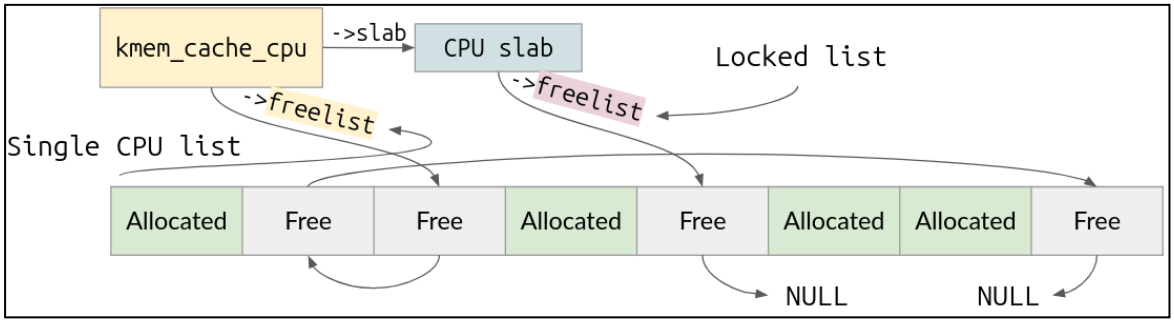
kmem_cache_cpu - השדה CPU slab

כאשר דיברנו על המבנה kmem_cache_cpu הזכרנו את ה-CPU slab ואמרנו שבו ה-cache משתמש כדי לבצע את הקצאת הזיכרון הבאה כאשר תתבקש, כדי לייעל את התהליך של הקצאת האובייקט מתוך ה-CPU slab נשמר מצביע נוסף המשמש בתור free list עבורו בתוך המבנה kmem_cache_cpu שהופך להיות ה-free list "הראשי" שלו, ה-cache לוקח את המצביע של ה-free list המקורי מה-slab ומכניס אותו בשדה המתאים באובייקט kmem_cache_cpu ושם NULL ב-free list המקורי של ה-slab, כעת ה-free list של ה-slab משמשת רק לצורך שחרור של אובייקטים אשר נעשו דרך מעבדים אחרים ב-Node והיא מצריכה ביצוע פעולת lock לפני הוספת האובייקט ל-free list כדי שלא יהיו בעיות במידה ומעבד נוסף מנסה לבצע פעולה דומה באותו הזמן.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

הרשימת free list הנמצאת ב-kmem_cache_cpu משמשת אך ורק מעבד אחד ולכן אינה מצריכה פעולת lock בכלל והשחרור אליה וההקצאה ממנה יבוצעו בצורה מהירה יותר, כאשר היא מתרוקנת אז ה-free list המקורי של ה-CPU slab שכנראה התמלא באובייקטים ששוחררו על ידי מעבדים אחרים מועבר למבנה kmem_cache_cpu וכך יהיה ניתן להשתמש באותו ה-slab שוב.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

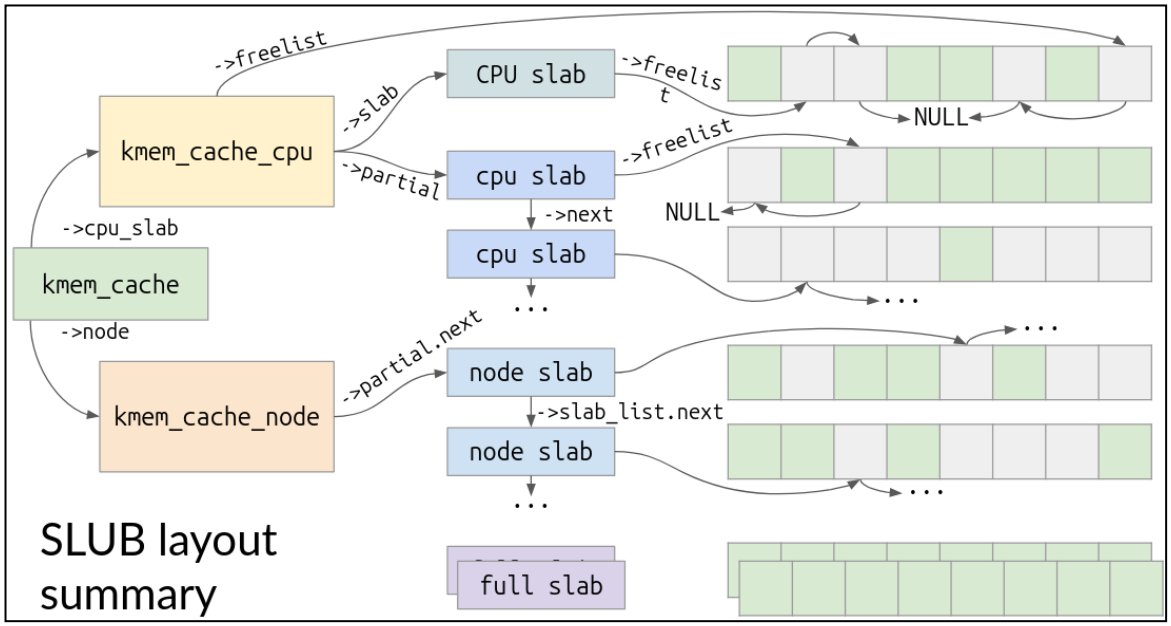
מבנים ב-SLUB - סיכום

הבנו שכדי להקצות אובייקטים דרך ה-Slab Allocator עלינו לגשת ל-cache, בתוך ה-cache יש את המבנה kmem_cache_cpu שנוצר עבור כל מעבד בנפרד ואת המבנה kmem_cache_node שנוצר עבור כל Node (קבוצת מעבדים).

במבנה kmem_cache_cpu ישנו מצביע ל-CPU slab ודרכו ה-cache משרת את המעבד שיוזם את בקשת הזיכרון דרך המצביע הנוסף ל-free list של אותו ה-slab ב-kmem_cache_cpu, כמו כן גם אמרנו שה-kmem_cache_cpu מחזיק מצביע לרשימה נוספת של partial slabs המכילה אובייקטים שלא שומשו כלל או שומשו חלקית.

סוג המבנה הנוסף שה-cache מחזיק הוא ה-kmem_cache_node המכיל רשימה מקושרת דו-כיוונית של slabs משומשים חלקית המיועדים לשימוש על ידי כלל המעבדים החברים באותו ה-Node.

לגבי slabs שה-free list שלהם התרוקן ולא ניתן להקצות מהם יותר אמרנו שה-Slab Allocator לא מחזיק תיעוד כלשהו גם לא ב-cache עבורם והם ישארו ככה עד שאחד האובייקטים מתוכם ישוחרר ואז הם יחזרו לרשימה בתוך אחד ה-caches.

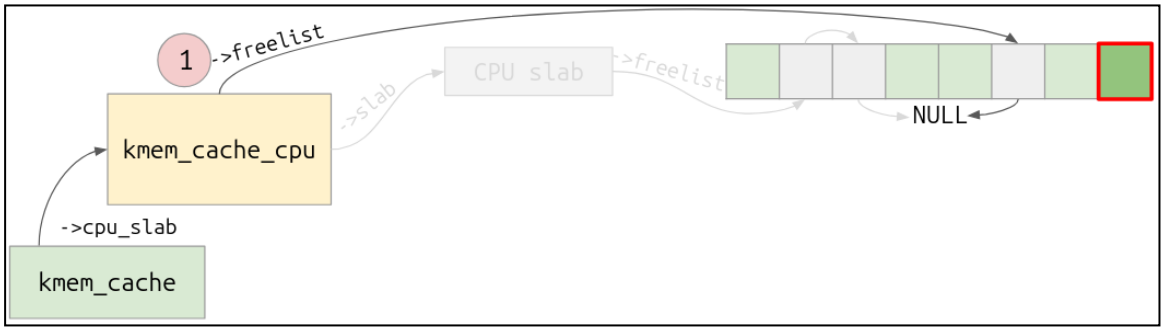


[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

הקצאת אובייקטים ב-SLUB

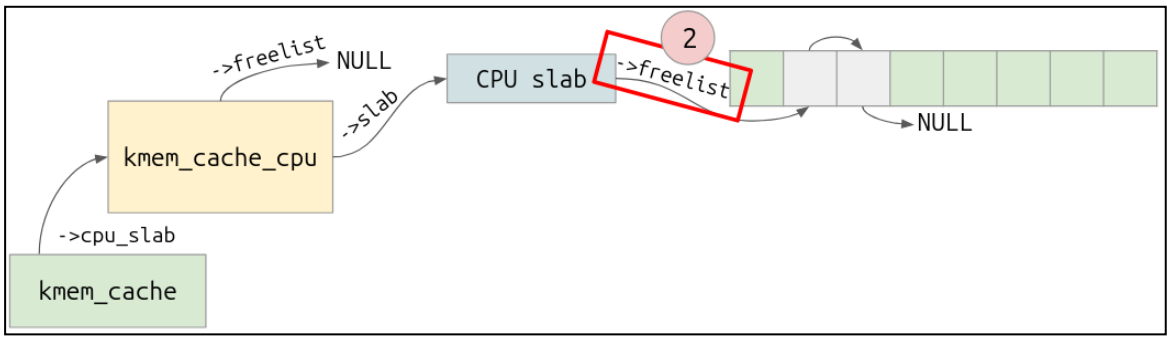
התהליך של הקצאת אובייקט מתוך cache הוא דינאמי ויכולים להיות כמה מצבים שונים בהם ה-cache נמצא, לכן כדי להבין מהיכן האובייקט בסוף יגיע ומה הן הפעולות אותן ה-cache יצטרך לנקוט כדי לבצע זאת עלינו להבין שלב אחר שלב כיצד ה-cache מנהל את ההקצאות האלו.

CPU slab: לכתחילה ה-cache ינסה את הדרך המהירה ביותר שלו שהיא לגשת אל ה-free list של ה-kmem_cache_cpu ולהקצות משם אובייקט.



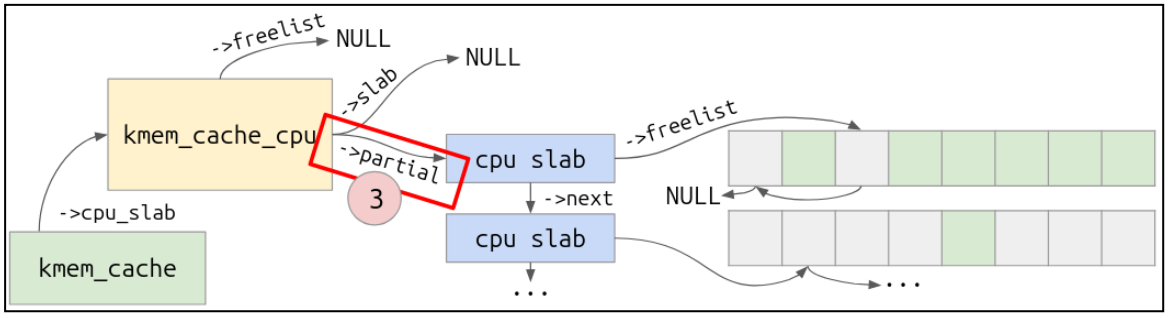
[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

במידה והרשימה הזו ריקה ה-cache יגש לlocked free list של ה-CPU slab ויעביר אותה ל-kmem_cache_cpu וישתמש בה לצורך ביצוע הקצאת האובייקט.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

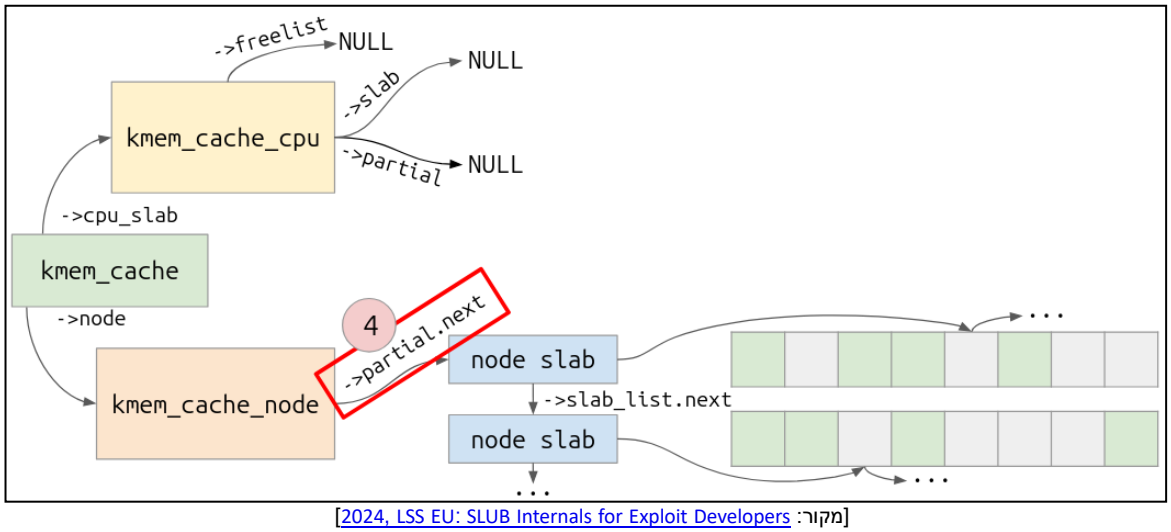
Partial list: אם גם ה-locked free list של ה-CPU slab ריקה זה אומר שאין ל-slab עוד אובייקטים שניתן להקצות אותם וצריך להשיג slab חדש ומה שה-cache יעשה לצורך כך הוא לגשת ל-partial list ולחלץ משם slab ולהפוך אותו להיות ה-CPU slab החדש ולהשתמש בו כמו שהסברנו קודם לכן.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

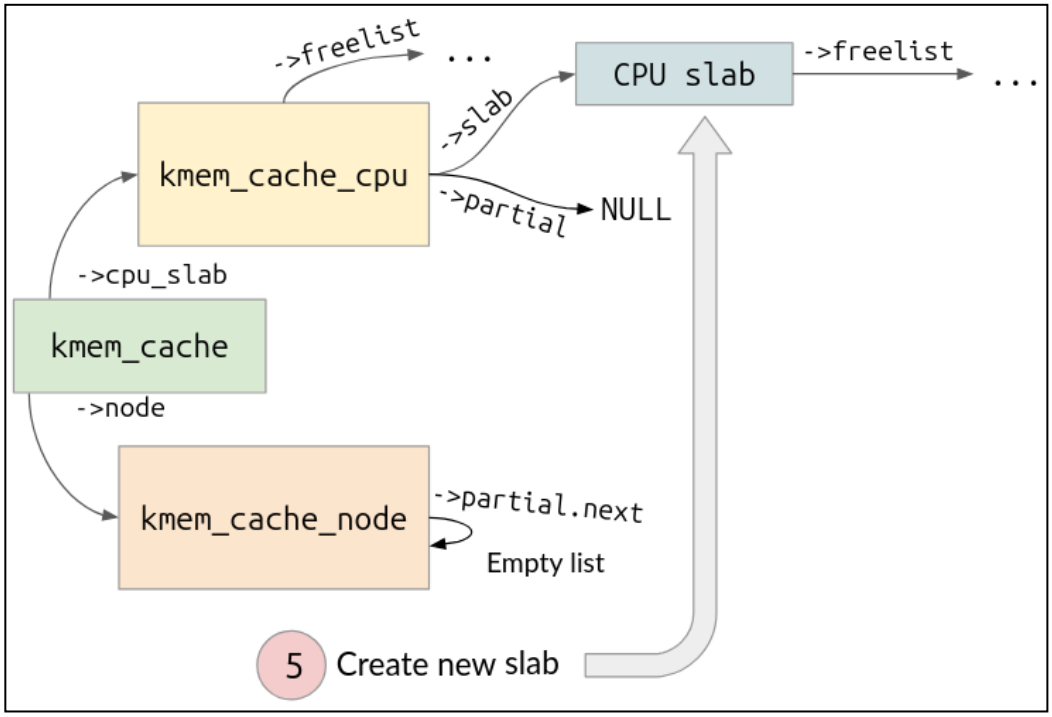
Node list: במידה וגם ה-partial list התרוקן כלומר ה-kmem_cache_cpu כבר לא מחזיק slabs עם אובייקטים שניתן להקצות אז הוא ינסה להשיג slabs נוספים מה-Node שבו המעבד שביצע את בקשת הזיכרון חבר כלומר מתוך ה-slabs שהמבנה kmem_cache_node מחזיק.

ה-slab הראשון שהוא יקח יהיה ה-CPU slab החדש ולאחר מכן הוא יקח slabs נוספים בהתאם לשדה cpu_partial_slabs המתואר במבנה kmem_cache_node וימלא איתם את ה-partial list של ה-kmem_cache_cpu מחדש ולבסוף יבצע את הקצאת הזיכרון דרך ה-CPU slab.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

New slab: בהנחה וגם ה-partial list של ה-kmem_cache_node ריק ואין slabs משותפים כלל ב-Node בו המעבד חבר במצב כזה ה-cache ייצר מבנה slab חדש שבו הוא ישתמש בתור ה-CPU slab החדש שלו ויבצע את הקצאת הזיכרון דרכו באופן הרגיל:

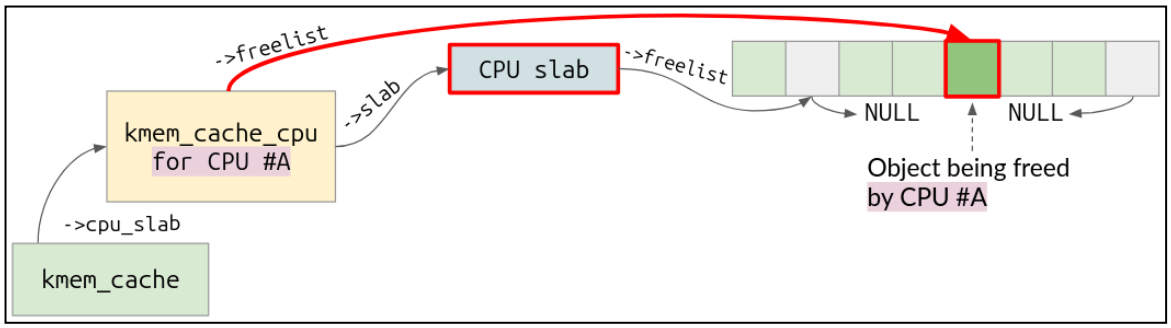


[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

שחרור אובייקטים ב-SLUB

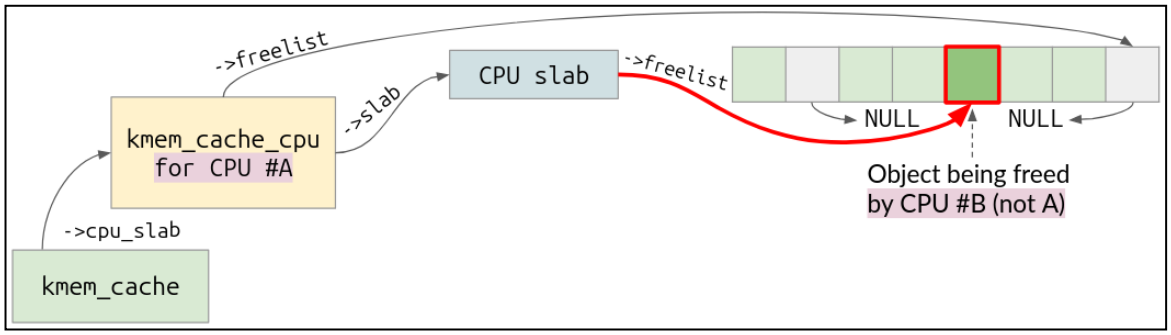
החזרה של אובייקטים אל ה-free lists לאחר השחרור שלהם היא כדי למחזר אותם להקצאות חדשות, כמו תהליך ההקצאה גם בתהליך השחרור יכולים להיות מצבים שונים וחלקם מורכבים הדורשים הסבר כל אחד בפני עצמו.

CPU Slab: במידה והאובייקט המשוחרר שייך ל-CPU slab של המעבד שמבצע את פעולת השחרור אזי הוא יכניס אותו ישירות ל-free list שהוא מחזיק בתוך המבנה kmem_cache_cpu עבור אותו המעבד.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

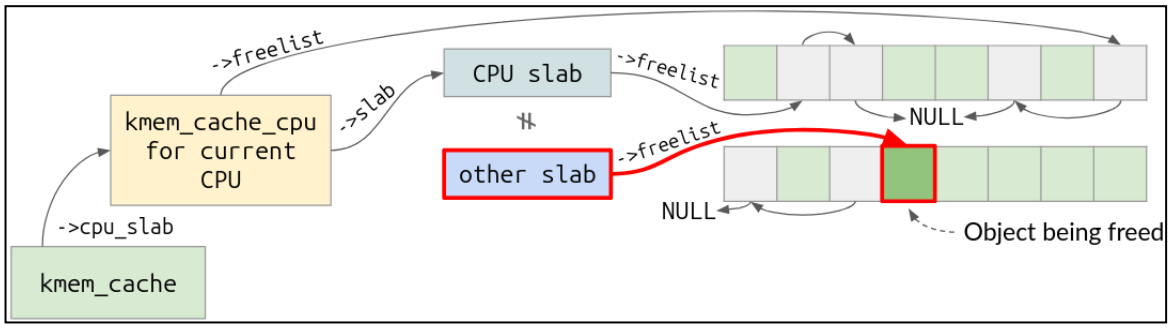
Other CPU: כאשר אנו משחררים אובייקט ששייך ל-CPU slab אבל לא למעבד שאנו עובדים איתו כרגע אזי הוא מתווסף אל ה-free list של אותו ה-CPU slab אך דרך ה-locked list הרגילה שלו ולא דרך המבנה kmem_cache_cpu השמור רק למעבד שעובד עם אותו ה-CPU slab בלבד.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

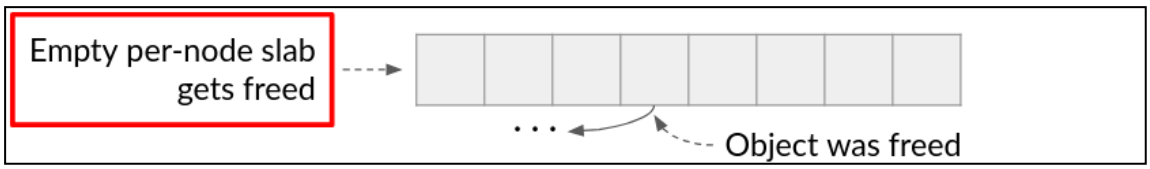
Other slab: אם האובייקט ששחררנו שייך ל-slab שהוא לא ה-CPU slab ישנם שני מקרים בהם ה-Slab Allocator בוחר כיצד להתייחס אל ה-slab אליו הוא שוחרר:

- **Partial list:** במידה והוא שייך לאחד ה-partial slabs של מבנה kmem_cache_cpu כלשהו אזי האובייקט פשוט יכנס אל תוך ה-free list שלו.



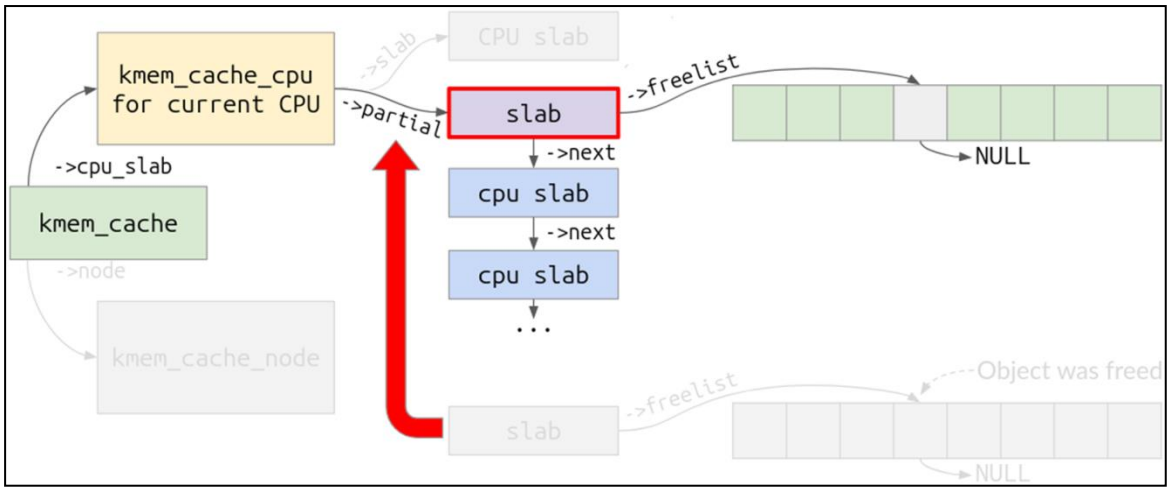
[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

- **Node slab:** במקרה שאותו ה-slab לא בשימוש עבור אף אחד מהמעבדים ונמצא ברשימת ה-partial slabs במבנה ה-kmem_cache_node אזי הוא ישתחרר אל תוך ה-free list של ה-slab אליו הוא שייך, במידה ולאחר השחרור של האובייקט הזה אותו ה-slab כעת יהיה כולו לא בשימוש כלומר ללא אובייקטים מאולקצים עם free list מלא אז ה-Slab Allocator ידאג לשחרר את הזיכרון שאותו ה-slab תופס אל ה-buddy allocator לשימוש חוזר.



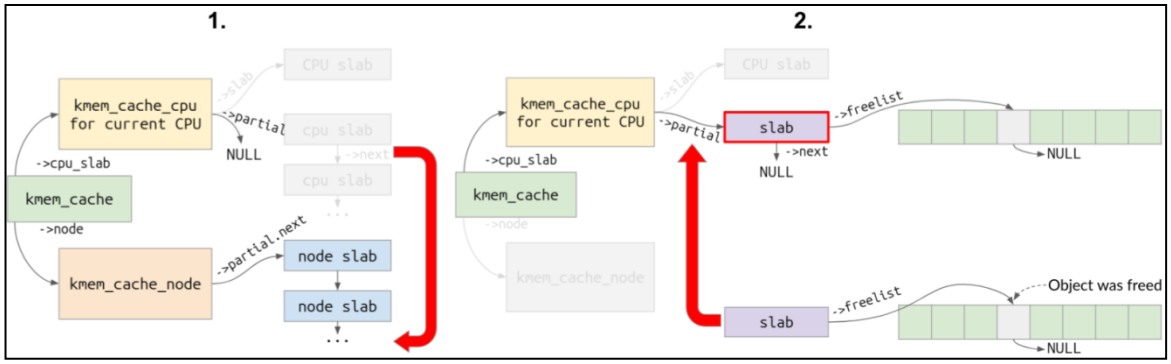
[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

Full slab: הזכרנו לפני כן שה-cache לא מצביע יותר אל slab במידה והוא מלא וה-free list שלו ריק, לכן כאשר נשחרר את אחד האובייקטים שלו ה-Slab Allocator ימקם אותו על ה-cache מחדש ברשימה של ה-partial slabs במידה ועדיין היא לא הגיעה למקסימום slabs שהיא יכולה להכיל.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

במידה והיא הגיעה אל המקסימום, תחילה ה-Slab Allocator ירוקן את כולה לתוך ה-partial list של המבנה kmem_cache_node המייצג את ה-Node של אותו המעבד, זאת כדי לנסות לשחרר זיכרון מיותר של slabs מלאים הנמצאים ב-partial list של המעבד שלא נעשה שימוש באף אחד מהאובייקטים שלהם לצורך שחרור הזיכרון שהם תופסים אל ה-buddy allocator ודבר זה מתאפשר אך ורק כאשר הם חלק מה-partial list של המבנה kmem_cache_node כמו שהזכרנו לפני כן, במקומם הוא יכניס את ה-slab שאליו שוחרר האובייקט.



[מקור: 2024, LSS EU: SLUB Internals for Exploit Developers]

סיכום

דיברנו על הקצאות זיכרון דינמיות בקרנל של לינוקס והבנו שהן מתבצעות דרך ה-Slab Allocator, הבנו שכדי להקצות זיכרון באמצעותו עלינו לפנות ל-caches שהם נוצרים ומנהלים את הזיכרון עבורנו על פי גודל וכמות של אובייקטים לבחירתינו או שהם נוצרים בצורה גנרית כדוגמת kmalloc כדי לאפשר הקצאת זיכרון גם ללא צורך להגדיר מראש מאיזה cache היא תגיע.

בנוסף גם ראינו כמה מנגנוני אבטחה של ה-free list כדי למנוע שגיאות לא רצויות שיגרמו לקריסה של הקרנל ואיתו תבוא קריסה של המערכת כולה וכן גם כיצד נעשה שימוש חוזר באובייקטים באמצעות ה-free lists כדי לאפשר יעילות טובה יותר בניהול הזיכרון.

לבסוף ראינו כיצד מועברים slabs מתוך ה-Node אל מעבד יחיד כדי לבצע ממנו את הקצאת הזיכרון וגם להפך מה-partial list אל ה-Node וכן גם את כיצד מתבצע התהליך המלא של הקצאה ושחרור של אובייקטים עבור ה-Slab Allocator במימוש SLUB.

על המחבר

מלש"ב, לומד בישיבה הגבוהה מעלה אליהו בתל אביב. נהנה לקרוא ולחקור בעולמות ה-low level.
פרויקטים או מאמרים נוספים שכתבתי ניתן למצוא כאן:
https://github.com/4f3rg4n | https://4f3rg4n.github.io | linkedin.com/in/noam-afergan-074176284

ליצירת קשר אפשר לפנות אליי במייל noam18.af@gmail.com או בטלגרם @cyb3rat

מקורות מידע

- Andrey Konovalov - SLUB Internals for Exploit Developers:
  https://www.youtube.com/watch?v=XulsBDV4n3w&feature=youtu.be
  Slides:
  https://static.sched.com/hosted_files/lsseu2024/37/2024%2C%20LSS%20EU_%20SLUB%20Internals%20for%20Exploit%20Developers.pdf
- Non-Uniform Memory Access by Intel:
  https://www.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf
- Elixir bootlin - linux:
  https://elixir.bootlin.com/linux/v6.6/source/include/linux/slub_def.h
  https://elixir.bootlin.com/linux/v6.6/source/mm/slab.h