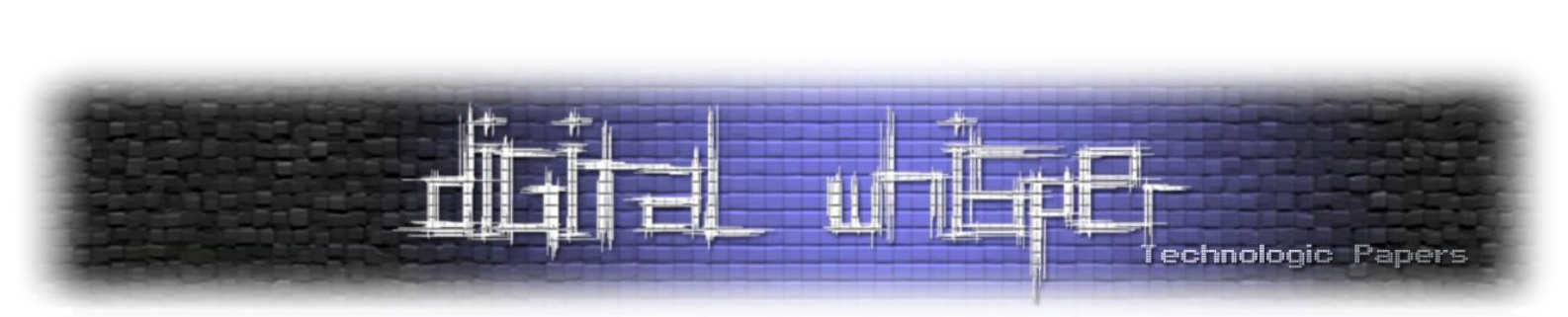


ניצולי Low Fragmentation Heap ב- Windows Userspace

מאת סער אמר

הקדמה

ניצול חולשות דריכת זיכרון דורש מיומנות והבנה בלא מעט רכיבים במערכת, והוא לובש צורות שונות בין פלטפורמות שונות. בשנים האחרונות, ניצול חולשות דריכה ב-Windows userspace נהיה מאתגר ומעניין. כשמנצלים חולשות memory corruptions שמערבות את ה-heap, כמו קריאה/כתיבה Out of bound, UAF וכו', חשוב מאד להבין איך ה-allocator הרלוונטי עובד: איך הוא מנהל את chunk-ים בזיכרון וכמובן איך הם נראים.

מאמר זה יספר על ה-allocator המרכזי שמעורב ב-Windows userspace, שהוא ה- Low Fragmentation Heap (LFH).

בפיתה או בלאפה?

כנראה שלא יפתיע את רוב קוראי המאמר שתוכנות מנהלות זיכרון דינאמית במהלך הריצה שלהן. הממשקים של הספרייה הסטנדרטית של C לניהול זיכרון, malloc ו-free, הם פשוטים מאוד בתכליתם: הם מאפשרים לקוד לבקש אזור זיכרון פנוי בגודל מסוים, או לשחרר אותו. הממשק הזה גמיש מאוד, ומשאיר את העבודה הקשה של ניהול המשאבים בידי ה-allocator. בפועל, הוא דורש ממנו להתמודד עם מצבים קשים יחסית, כי אין שום הבטחה בנוגע לגודל הקצאות הזיכרון ("אלקוצים") שיהיו, הכמות שלהם, סדר האלקוצים והשחרורים, והשימוש שלהם ב-thread-ים שונים. המורכבות הנדרשת מהאלוקטור, בשילוב השימוש הנפוץ שלו בקוד, הפכה אותו ליעד חשוב בניצול חולשות זיכרון.

ב-Windows, יש מספר ממשקים עשירים יותר שמגיעים לאותו אלוקטור. פונקציות כמו malloc ו-LocalAlloc הן עטיפות שמשתמשות ב-heap המרכזי ועושות עליו HeapAlloc. הפונקציה הזו מאפשרת

לנו לציין באיזה heap אנחנו מאלקצים, במידה ואנחנו משתמשים בכמה heapים. מעבר לציון ה-heap ו-flagים נוספים, גם היא כמו malloc מקבלת את גודל האלקוץ ומחזירה כתובת.

עד ל-Windows Vista, האלוקטור המרכזי ב-userspace היה ה-NT heap, וקריאות לפונקציות האלה היו מגיעות אליו. NT heap הוא general purpose allocator מורכב יחסית, שמנהל chunk-ים בגדלים משתנים, ויודע לחלק ולחבר אותם בזיכרון במידת הצורך. הוא מזכיר במידה מסוימת את מקבילו הלינוקסי, dlmalloc. המטרה שלהם היא להיות יעילים בזמן ובזיכרון, לכן בכל אלקוץ היו חותכים חתיכה מאזור יחסית גדול (chunk), ובשחרור מחזירים את החתיכה הזאת לטיפול האלוקטור. זה היה יוצר פרגמנטציה ב-heap שאיתה הן היו צריכים לטפל בה, על ידי איחוד ופיצול חתיכות כאלה.

למרות זאת, סגנון התכנות הנפוץ בקרנל של Windows ו-Linux היה שונה: במקום להשתמש באלקוצים בגדלים משתנים, מוגדרים pool-ים, שהם למעשה אזורים בזיכרון שמוכנים מראש וחתוכים ל-chunk-ים בגודל קבוע. למעשה, ניהול הזיכרון ב-pool-ים כאלה הופך לפשוט יותר, כי עוד לפני האלקוץ האלוקטור יודע בדיוק מה יהיה הגודל שיבקשו ממנו (או שיחזירו אליו, בשחרור). זה מאוד יעיל בזמן כי זה חוסך את אותה לוגיקה של איחוד ופיצול chunk-ים, שצריכה לקרות כל הזמן. לקרנל זה חשוב כי זה גם מאפשר לו לתכנן מראש ולדעת שאלקוצים מסוימים לא יכולים להכשל (בהנחה וניהול המשאבים שמטופלים ב-pool הזה הוא נכון, בצד ה-caller).

פרקטית, עם מרחב זיכרון וירטואלי גדול מאוד, ועם זיכרון פיזי שכבר מגיע לגדלים יפים, היה אפשר ליישם design דומה גם ב-userspace, אבל זה היה דורש לשנות המון תוכנות שכיום עובדות ישירות עם פונקציות האלקוץ הסטנדרטיות והמוכרות.

כדי לענות על זה, ב-Windows Vista פיצלו את האלוקטור לשני אלוקטורים - frontend ו-backend. ה-backend הוא ה-NT heap הישן והמוכר. אלקוצים דרך כל ה-API-ים לאלקוץ יגיעו קודם כל ל-frontend, שנעזר ב-backend לניהול המשאבים של עצמו.

ופה נכנס לתמונה ה-Low Fragmentation Heap, או בקיצור, LFH. ה-LFH הוא ה-frontend allocator החל מ-Windows Vista. המטרה שלו הוא להרגיש מה גדלי האלקוצים הנפוצים במהלך ריצת התוכנית, ולאלקץ עבורם pool-ים יעילים מאוד דרך ה-backend. משם, הוא יתחיל לשרת גדלים כאלה מתוך אותו pool (שנקרא בשפת ה-LFH בשם userblocks), במקום ללכת ל-backend עבור כל אלקוץ.

אז למה Microsoft מימשו את כל הדבר הזה? על מנת להבין את הרציונאל מאחורי צעד זה, בואו נבחן את היתרונות והחסרונות ב-NT heap וב-LFH:

	NT heap	LFH
מהירות אלקוץ	מהיר אך מסורבל. במקרה הפשוט, הוא ימצא אזור פנוי דרך linked list, יפצל אותו, ויחזיר אותו. במקרה הקשה, יבקש זיכרון נוסף ממערכת ההפעלה.	מהיר מאוד. במקרה הפשוט, הוא ימצא chunk פנוי בחיפוש ב-bitmap, ויחזיר אותו ישר. במקרה הקשה, הוא יבקש זיכרון נוסף מה-NT heap עבור ה-chunk הזה ועבור אלקוצים עתידיים.
מהירות שחרור	איטי. בכל שחרור הוא חייב לטפל במצב של איחוד עם chunk-ים סמוכים, ושינויים של ה-linked listים שבהם הם מופיעים.	מהיר מאוד. שחרור הוא כיבוי של ביט.
ניהול זיכרון	יעיל בזיכרון. הוא תמיד יעדיף למלא חורים בזיכרון שהם לא בשימוש כדי להשתמש בהם מחדש.	בזבזני. הוא יעדיף לשמור מראש רזרבות לאלקוצים עתידיים, ולא ישתמש בהם אף פעם לגדלים שונים מהגודל המקורי שהתכוונו אליו.
בדיקות על דריכות בין chunk-ים	קיימת הגנה על חלק מה-header באמצעות xor עם ערך רנדומלי.	קיימת הגנה על חלק מה-header באמצעות xor עם ערך רנדומלי.
סדר אלקוצים צפוי	לוגיקה מורכבת אך דיטרמיניסטית.	לא דיטרמיניסטי.

נראה שיש יתרונות וחסרונות לשניהם. פרט לעובדה שה-LFH לפעמים מבזבז זיכרון, כשמסתכלים על שיקולים אלו, הבחירה ב-LFH מתבקשת. הפיצול ל-frontend ו-backend מאפשר ל-LFH להכנס לפעולה רק שהוא היוריסטית מזהה שמשתלם לו לנהל את הזיכרון. במערכות מודרניות הזיכרון הוירטואלי והפיזי כבר גדול מספיק שמשתלם לנו לבזבז קצת זיכרון כדי לחסוך בזמן ריצה. חוץ מזה, כמובן שאנחנו רוצים allocator שהוא כמה שיותר secure.

אם מנצלים כתיבה רלטיבית ב-heap או UAF, אנחנו בדרך כלל צריכים למצוא פרמיטיבים לאלקוץ ולשחרור, לחפש מבנים שמעניין לדרוך עליהם, לזייף אותם בזיכרון, וכו'. המטרה של המאמר הזה היא לדבר ההשפעה של מנגנוני האבטחה החדשים, ועל איך ה-LFH משנה את צורת העבודה שלנו בניצול חולשות מסוגים שונים. חלקם היו מאוד קלים לניצול באלוקטורים אחרים כמו NT heap או dlmalloc, אך לא כל כך פשוטים ב-LFH כמו שהוא היום.

לפני שנצלול לפרטים, כדאי לדעת שה-LFH הוא מנגנון שמשתנה לעיתים. השיפור המשמעותי ביותר עד כה היה בין Windows 7 לבין Windows 8 בו הופיע לראשונה סדר האלקוצים הלא דיטרמיניסטי של ה-

LFH, וגם השתנה ה-metadata של chunk להקשות על דריכות לינאריות. המאמר מתייחס למצב היום, נכון ל-Windows 10 RS5, אבל כדאי להתעדכן מעת לעת.

LFH Internals

אז היום ה-allocator frontend הוא ה-Low Fragmentation Heap. הרעיון מאחריו מסתכם לשני כללים פשוטים:

- Chunk-ים בגדלים קרובים יושבים באותו אזור זיכרון, קרובים אחד לשני.
- אין fragmentation - לא יהיה consolidate ולא coalesce. Chunk-ים אף פעם לא מתפצלים או מתאחדים באלקוץ או בשחרורם.

לפי החוקים האלה אפשר לחשוב על ה-LFH בתור יותר pool מאשר heap. כל pool מוכן on-demand, ברגע שהיו ב-runtime מספיק אלקוצים באותו גודל וככה נותן ממשק דמוי heap.

ה-chunk-ים יושבים בזיכרון במבנה שנקרא userblocks, שהוא פשוט אוסף של pages ששברנו לחתיכות קטנות בגודל שווה, כל חתיכה כזו היא chunk שישרת אחר כך אלקוץ באותו הגודל. ה-userblocks מכיל chunk-ים בגדלים קרובים, ולא מערבב גדלים.

בניגוד ל-heap קלאסי, ה-allocator אף פעם לא יפצל chunk, במקרה בו הוא מאלקץ גודל שקטן מהמקסימום האפשרי ב-userblocks אז ישאר זיכרון "מבוזבז" אחריו. במקרה זה, אם משחררים chunk, וה-chunk-ים הצמודים משחוררים גם הם, ה-allocator לא ימזג את אותו ה-chunk עם שכניו. כלומר, ה-userblocks ישאר תמיד באותו מצב מבחינת חלוקה ל-chunk-ים החל מרגע היווצרו. הפרמטר היחיד שמשתנה הוא הסטטוס של כל chunk - האם הוא מאולקץ או משוחרר.

Chunk-ים שיושבים ב-userblocks מסויים שייכים לאותו bucket. ב-bucket הכוונה לקבוצת גדלים באותה קפיצה של granularity - קפיצה אחת של alignment בגדלי האלקוצים. הם כל אותם אלה שבטווח קפיצה אחת של granularity (מתחיל מ-0x10, ועולה ככל שהגדלים גדלים). כלומר, 0x41, 0x42, 0x43 וכו' - ישבו באותו userblocks, מכיוון שהם באותו טווח. להלן תרשים ה-bucket-ים, גדלי האלקוצים וקפיצות ה-granularity:

Bucket	Allocation Size	Granularity
1 – 64	1 - 1,024 bytes (0x1 - 0x400)	16 bytes
65 – 80	1,025 - 2,048 bytes (0x401 - 0x800)	64 bytes
81 – 96	2,049 - 4,096 bytes (0x801 - 0x1000)	128 bytes
97 – 112	4,097 - 8,192 bytes (0x1001 - 0x2000)	256 bytes
113 – 128	8,193 - 16,368 bytes (0x2001 - 0x3FF0)	512 bytes

יצירת והכנת Userblocks

אז עבור כל granularity יש לנו userblocks, אוסף chunk-ים באותו הגודל בדיוק, שמחכים לשרת אלקוצים במשפחת הגדלים הזו. עתה, נתאר כיצד malloc ו-free עובדים במודל זה.

למרות שה-LFH מזכיר pool allocators, יש הבדל משמעותי בממשק שלהם. Pool-ים בדרך כלל מאותחלים כדי לשרת גודל קבוע מראש, אבל ה-LFH צריך לשרת אלקוצים בגדלים משתנים שאינם ידועים מראש. כדי להקל על העבודה, הוא מחלק קבוצות של גדלים ל-buckets, אבל כיצד הוא יודע איזה גדלים יהיו?

כשאנחנו קוראים לאלקוץ ב-userspace, ה-frontend מקבל את הבקשה ובוחן האם הוא צריך לשרת אותה. כפי שציינו, ה-LFH עובד ב-bucketים ע"י granularity, ולכן הוא פשוט בודק האם ה-bucket של הגודל הרלוונטי הוא "active", או במילים אחרות, שה-LFH כבר התחיל לטפל בו. במידה והוא אכן active, הוא מנסה לענות על הבקשה. במידה ולא, הוא מעביר אותה ל-backend, ומעדכן סטטיסטיקה שאמורה בסופו של דבר להכריע מתי מפעילים את ה-bucket הזה. החתימה ההיוריסטית הזו קובעת כי אחרי 0x11 אלקוצים רציפים מ-bucket מסויים הוא נהיה active, ו-0x12 אלקוצים במידה וזה ה-bucket הראשון שנהיה active. כך ה-LFH לומד איזה גדלים נפוצים משתלמים עבורו לטפל בהם, במהלך ריצת התוכנית.

במידה וה-LFH אכן משרת את הבקשה על ה-bucket הספציפי הזה, הוא צריך לבדוק קודם כל האם יש userblocks פעיל. אם אין אחד כזה, צריך לאלקץ userblock חדש. מצב זה יכול לקרות כאשר:

- ה-bucket נהיה active אך מעולם לא שירתנו אותו.
- כבר אלקצנו את כל ה-chunk-ים מה-userblocks של אותו ה-bucket, וכעת צריך לאלקץ userblocks נוסף.

במצב הפשוט, שבו קיים כבר userblocks ויש בו chunk-ים זמינים, האלוקטור ישתמש בו. ניהול המצב של ה-chunk-ים ב-userblocks הוא פשוט ומהיר, כי אין לנו התייחסות כלל לשכנים שלנו ול-state שלהם, ובוחנים כל chunk בנפרד. עבור כל userblocks קיים subsegment שמחזיק את ה-metadata שלו. ממנו, קיים bitmap שמחזיק את ה-state של כל chunk ב-userblocks. אם צריך לאלקץ chunk, מדליקים את הביט המתאים לאינדקס שלו ב-bitmap, ואם צריך לשחרר, מכבים אותו. בתצורה הזאת, ה-state של ה-chunk בכלל לא מוחזק בתוכו, והתהליך מהיר ופשוט.

במידה וצריך לאלקץ userblocks נוסף, ה-LFH פונה ל-backend ומבקש ממנו לאלקץ לו pageים, שעתידים להיות ה-userblocks החדש. גודל האלקוץ שהוא מבקש מה-backend תלוי בכמות ה-chunk-ים שהוא הולך להחזיק, והחישוב הזה תלוי בסוג ה-bucket.

לא נכנס כאן לכל החישוב של גודל ה-userblocks, אבל כן חשוב לדעת שיש שני חסמים עליו, ששניהם חייבים להתקיים:

0. כמות ה-chunk-ים ב-userblocks לא תעלה על 0x400
1. כמות הבתים הכוללת חייבת להיות מתחת ל-0x78000

שני החסמים האלה קבועים בקוד ב-ntdll-ים. ככל שה-granularity גדולה יותר, ככה יש יותר סיכוי שנפגע קודם בחסם השני מאשר הראשון. לאחר שיש כבר אזור זיכרון חדש עבור ה-userblock מה-backend, צריך לאתחל אותו. התהליך הזה מטופל על ידי ntdll!RtlpSubSegmentInitialize. הוא מתחיל בלאתחל שדות, וקובע אותו להיות ה-userblocks הבא בתור:

```
//figure out the total sizes of each chunk in the UserBlocks
unsigned int TotalSize = ChunkSize + sizeof(_HEAP_ENTRY);
unsigned short BlockSize = TotalSize / 8;

//this will be the number of chunks in the UserBlocks
unsigned int NumOfChunks = (SizeNoHeader - sizeof(_HEAP_USERDATA_HEADER)) / TotalSize;

//Set the _HEAP_SUBSEGMENT and denote the end
UserBlocks->SfreeListEntry.Next = NewSubSeg;

char *UserBlockEnd = UserBlock + SizeNoHeader;

//Get the offset of the first chunk that can be allocated
//Windows 7 just used 0x2 (2 * 8), which was the size
//of the _HEAP_USERDATA_HEADER
unsigned int FirstAllocOffset = ((((NumOfChunks + 0x1F) / 8) & 0x1FFFFFFC) +
       sizeof(_HEAP_USERDATA_HEADER)) & 0xFFFFFFF8;

UserBlocks->FirstAllocationOffset = FirstAllocOffset;
```

עכשיו, עוברים על כל אזור הזיכרון, ובונים את ה-chunk-ים בתוכו, מאתחלים את ה-header של כל אחד מהם.

```
//if permitted, start writing chunk headers every TotalSize bytes
if(UserBlocks + FirstAllocOffset + TotalSize < UserBlockEnd)
{
       _HEAP_ENTRY *CurrHeader = UserBlocks + FirstAllocOffset;

       do
       {
              //set the encoded lfh chunk header, by XORing certain
              //values. This is how a Subsegment can be derived in RtlpLowFragHeapFree
              *(DWORD)CurrHeader = (DWORD)Heap->Entry ^ NewSubSeg ^
                     RtlpLFHKey ^ (CurrHeader >> 3);

              //FreeEntryOffset replacement
              CurrHeader->PreviousSize = Index;

              //denote as a free chunk in the LFH
              CurrHeader->UnusedBytes = 0x80;

              //increment the header and counter
              CurrHeader += TotalSize;
              Index++;
       }
       while((CurrHeader + TotalSize) < UserBlockEnd);
}
```

מכאן, ה-userblock החדש מוכן לשימוש. רואים כאן כמה שדות קריטיים שמאותחלים ב-header של כל chunk:

0. קידוד של פוינטר מכל chunk ל-subsegment שמנהל אותו. ה-subsegment הוא אחד המבנים הבסיסיים ביותר, והוא אחראי על ניהול ה-chunk-ים ב-LFH. לפי קידוד זה, בשחרור של ה-chunk, ה-LFH יודע לגשת ל-subsegment המתאים.

1. LFHFlags, שיושב בשדה UnusedBytes ותמיד שווה ל-0x80. זה מסמן ל-frontend בשחרור כי זהו chunk שנוהל על ידי ה-LFH, ויש להמשיך להתייחס אליו כך. **שחרור של chunk כזה יגיע בהכרח ל-LFH ולא ל-NT heap.**

2. PreviousSize, שלמרות השם המטעה שלו (השם במקור שייך ל-NT heap), משמש אותנו ל-index ב-bitmap. כך ה-LFH יודע איזה ביט לכבות כשמשחררים את ה-chunk.

דבר מעניין שאפשר להסיק מתהליך בניית ה-userblocks הוא הסדר שלהם בזיכרון. מכיוון שה-LFH פונה ל-backend כדי לאלקץ את הזיכרון עבורם, ה-backend יאלקץ את ה-page-ים הללו עם VirtualAllocEx, והם יתאלקצו בצורה רציפה וירטואלית, מהתחתית של ה-heap והלאה. לכן, במידה ואנחנו מסוגלים לרסס ולאלקץ הרבה, ניתן להגיע למצב שהם יתאלקצו בצורה רציפה אחד אחרי השני.

חדי העין יבחינו שה-pointer ל-subsegment ב-header של כל chunk עובר xor. התהליך ההפוך לזה יקרה בשחרור, כדי להגיע מ-chunk ל-subsegment שלו. הסיבה שהפוינטר לא מופיע כמו שהוא היא על מנת להגן מפני overflows בין chunk-ים. הוא עובר xor עם cookie אקראי, שנמצא כסימבול ב-ntdll ונקרא ntdll!RtlpLFHKey. כבר כאן אפשר להבין שמבלי להשיג קריאה ולגלות מה הערך של ה-cookie הזה בצורה כזאת או אחרת, לא נוכל לדרוך על ה-header ולזייף אותו. למרות זאת, לא כל ה-header עובר xor:PreviousSize לא עובר חישוב דומה. נבחן ניצולים פוטנציאליים של עובדה זו בהמשך המאמר.

נחזור לתהליך בניית ה-userblocks. ציינו שעבור כל userblocks פעיל, ה-LFH מאלקץ ומתחזק bitmap שמציין אילו chunk-ים תפוסים ואילו לא. הבנייה והאלקוץ של ה-bitmap קורית מיד לאחר ה-flow המוצג לעיל, והיא נראית כך:

```
//Initialize the bitmap and zero out its memory (Index == Number of Chunks)
RtlInitializeBitMap(&UserBlocks->BusyBitmap; UserBlocks->BitmapData, Index);

char *Bitmap = UserBlocks->BusyBitmap->Buffer;

unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitMap;

memset(Bitmap, 0, (BitmapSize + 7) / 8);

//This will set all the members of this structure
//to the appropriate values derived from this func
//associating UserBlocks and SegmentInfo
UpdateSubsegment(NewSubSeg,SegmentInfo, UserBlocks);
```

אז איפה יושב ה-bitmap הזה? הוא ממוקם בתחילת userblocks, ולכן אנו צריכים להשיג את המבנה userblocks. מגיעים אליו ע"י ה-dereferences הבאים:

```
Heap->LFH->InfoArrays[]->ActiveSubsegment->UserBlocks->BusyBitmap
```

אלקוץ ושחרור

מכאן כבר יחסית פשוט להבין איך עובדים malloc ו-free, בהנחה וכבר יש לנו userblocks. כאשר אנחנו קוראים ל-malloc על גודל מסויים, ה-LFH הולך ל-userblocks הפעיל הרלוונטי לגודל הזה, ניגש ל-bitmap שלו, ומחפש chunk משוחרר על מנת לשרת את הבקשה. בתחילה הוא מוציא את ה-userblocks מה- subsegment של ה-bucket:

```
//at this point we should have acquired a sufficient subsegment and can
//now use it for an actual allocation, we also want to make sure that
//the UserBlocks has chunks left along w/ a matching subsegment info structures
_HEAP_USERDATA_HEADER *UserBlocks = ActiveSubseg->UserBlocks;
```

אחר כך, צריך לבחור chunk פנוי. כאן נכנס מנגון חשוב של ה-LFH: במקום לבחור את ה-chunk הראשון הפנוי, ה-LFH יעבור על ה-bitmap החל מ-offset אקראי. בהמשך נדון על אופן ייצור הרנדום והעבודה איתו. החל מאותו offset אקראי, האלוקטור יחפש את ה-chunk הראשון הפנוי:

```
//we need to know the size of the bitmap we're searching
unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitmap;

//Starting offset into the bitmap to search for a free chunk
unsigned int StartOffset = Rand;

void *Bitmap = UserBlocks->BusyBitmap->Buffer;
```

ברגע שנמצא chunk פנוי, ה-LFH יסמן אותו כתפוס, ויחזיר אותו כ-return value של האלקוץ. במידה וכל ה-bitmap מלא, צריך ללכת ל-userblocks אחר או לאלקץ אחד חדש בדומה למה שהוסבר בחלק הקודם.

אופן החיפוש על גבי ה-bitmap פשוט באמצעות אופקודים יעילים של x86. שימו לב שעלינו למצוא chunk משוחרר, כלומר ביט כבוי. לשמחתנו, ב-x86, יש אופקוד בשם bsf (קיצור של Bit Scan Forward), שמוצא לנו את הביט הראשון שדלוק ב-dword מסויים. כדי למצוא את הביט הכבוי הראשון, נעשה not ל-bitmap, ונמצא את הביט הדולק הראשון אחרי ה-not עם bsf. כך נדע שה-chunk משוחרר.

נעשה זאת עד שנמצא אחד. במידה ואין, נצטרך ליצור userblocks חדש.

```
//Rotate the bitmap (as to not lose items) to start
//at our randomly chosen offset
int RORBitmap = __ROR__(*Bitmap, StartOffset);

//since we're looking for 0's (FREE chunks)
//we'll invert the value due to how the next instruction works
int InverseBitmap = ~RORBitmap;

//these instructions search from low order bit to high order bit looking for a 1
//since we inverted our bitmap, the 1s will be 0s (BUSY) and the 0s will be 1s (FREE)
//    <-- search direction
//H.O                                                    L.O
//-------------------------------------------------------
//| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0
//-------------------------------------------------------
//the following code would look at the bitmap above, starting at L.O
//looking for a bit position that contains the value of one, and storing that index
int FreeIndex;
__asm{bsf FreeIndex, InverseBitmap};
```

תחילת הפונקציה מקדמת את ה-bitmap ל-StartOffset, שהוא ה-offset האקראי ממנו אנו מתחילים לסרוק. לאחר שמצאנו index רלוונטי, עושים not, ומריצים פשוט Bit Scan Forward.

לבסוף, נדליק את הביט ב-bitmap כדי לסמן שה-chunk תפוס:

```
//shows the difference between the start search index and
//the actual index of the first free chunk found
int Delta = ((BYTE)FreeIndex + (BYTE)StartOffset) & 0x1F;

//now that we've found the index of the chunk we want to allocate
//mark it as 'used'; as it previously was 'free'
*Bitmap |= 1 << Delta;
```

וה-caller יקבל את הכתובת של ה-chunk ב-index שנבחר כך ב-userblocks. בזאת הצלחנו לאלקץ!

אופן השחרור של chunk הוא יותר פשוט. עבור heap וכתובת (ה-heap הדיפולטי אם הגענו לשחרור דרך free, או heap ספציפי אם הגענו דרך HeapFree), הולכים אחורה בזכרון כדי להגיע ל-header של ה-chunk. משם אפשר לחשב את ה-subsegment לפי החישוב:

```
SubSegment = *(DWORD*)header ^ (header / 8) ^ heap ^ RtlpLFHKey;
```

שזה ההפך של ה-xor שחושב בבניית ה-chunk ב-userblocks. יש לנו את הכתובת של ה-header (היא קצת לפני הכתובת שאותה משחררים), את הכתובת של ה-heap ואת ה-cookie מהזיכרון, אז כל הפרמטרים כאן ידועים.

עכשיו כשיש את ה-subsegment, מגיעים ממנו ל-userblocks ומשם ל-bitmap. משתמשים ב-PrevSize של ה-chunk כדי לדעת את ה-index של ה-chunk ב-userblocks, באמצעותו מחשבים את ה-index של הביט הרלוונטי ב-bitmap, ומסמנים אותו כמשוחרר. מכיוון שאין לנו מיזוגים ושבירות, זה כל התהליך כדי לשחרר את ה-chunk.

ניצולי LFH

כיום, ה-LFH מתקדם מבחינת שיקולי אבטחה ביחס ל-allocatorים אחרים. ב-allocatorים אחרים, קל להגיע למצב של רציפות בזיכרון, או לניצול של UAF, מכיוון שקל לחזות את הצורה שבה ה-allocator ינהל את הזיכרון שלו. זהו פרט חשוב מאוד, מכיוון שבניצולים אנו צריכים לעצב את ה-heap ולהגיע ל-layout מסויים:

0. אם יש לנו דריכה רציפה, אנו רוצים לאלקץ מבנים בזיכרון כך שהמבנה שדורכים ממנו יהיה לפני המבנה שעליו אנו דורכים.

1. במידה ויש לנו UAF, אנו רוצים לדעת אחרי כמה אלקוצים ושחרורים ה-allocator יחזיר את אותה הכתובת.

ה-LFH, מתוך כוונה ברורה להקשות על עיצובים כאלה, הכניס מנגנון רנדום ל-memory management, החל מ-Windows 8. זה מזכיר במידה מסוימת mitigations כמו ASLR.

לדוגמה, אם נסתכל על userblocks אחד בזכרון, אחרי 8 אלקוצים, הוא יכול להראות ככה:

FREE	FREE	FREE	FREE	BUSY Alloc #3	FREE	FREE	FREE
BUSY Alloc #4	FREE	FREE	BUSY Alloc #7	BUSY Alloc #5	FREE	FREE	BUSY Alloc #6
FREE	FREE	FREE	BUSY Alloc #1	FREE	FREE	FREE	FREE
BUSY Alloc #8	FREE	FREE	FREE	FREE	BUSY Alloc #2	FREE	FREE

בתרשים הזה ניתן לראות userblocks חדש שהתחלנו לאלקץ ממנו chunk-ים, ואנו רואים שהאלקוצים שלנו נופלים בצורה רנדומלית ביחס למיקום שלהם בזיכרון.

קל מאד לבדוק זאת, לדוגמה באמצעות הקוד הבא:

```c
int main(void) {

    HANDLE hHeap = HeapCreate(0, 0, 0);

    printf("[*] activate bucket 0x%x in LFH\n", SIZE);
    spray(hHeap, 0x12, FALSE);

    printf("[*] spray\n");
    spray(hHeap, 0x100, TRUE);

    return 0;
}

void spray(HANDLE hHeap, size_t cnt, BOOL trace) {
    void *p;
    for (size_t i = 0; i < cnt; i++) {
        p = HeapAlloc(hHeap, 0x0, SIZE);
        if (trace) {
            printf("HeapAlloc() == %p\n", p);
        }
    }
}
```

```
[*] activate bucket 0x100 in LFH
[*] spray
HeapAlloc() == 000001540F274B10
HeapAlloc() == 000001540F2747E0
HeapAlloc() == 000001540F2746D0
HeapAlloc() == 000001540F2745C0
HeapAlloc() == 000001540F274C20
HeapAlloc() == 000001540F274A00
HeapAlloc() == 000001540F274D30
HeapAlloc() == 000001540F2748F0
HeapAlloc() == 000001540F273F60
HeapAlloc() == 000001540F274070
HeapAlloc() == 000001540F274180
HeapAlloc() == 000001540F274290
HeapAlloc() == 000001540F2743A0
HeapAlloc() == 000001540F2744B0
HeapAlloc() == 000001540F276CF0
HeapAlloc() == 000001540F275480
HeapAlloc() == 000001540F277130
HeapAlloc() == 000001540F2768B0
HeapAlloc() == 000001540F275BF0
HeapAlloc() == 000001540F275590
HeapAlloc() == 000001540F276470
```

למרות שה-heap חדש לחלוטין, הכתובות שחוזרות מ-HeapAlloc נראות לא צפויות. לעומת זאת, אם נריץ את אותו הקוד בלי ה-LFH (ניצור heap עם הפרמטר HEAP_NO_SERIALIZE), כל האלקוצים חוזרים כצפוי, אחד אחרי השני.

```c
int main(void) {

    HANDLE hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0); //disable LFH

    printf("[*] activate bucket 0x%x in LFH\n", SIZE); // useless in the case of NO_SERIALIZE
    spray(hHeap, 0x12, FALSE);

    printf("[*] spray\n");
    spray(hHeap, 0x100, TRUE);

    return 0;
}

void spray(HANDLE hHeap, size_t cnt, BOOL trace) {
    void *p;
    for (size_t i = 0; i < cnt; i++) {
        p = HeapAlloc(hHeap, 0x0, SIZE);
        if (trace) {
            printf("HeapAlloc() == %p\n", p);
        }
    }
}
```

```
[*] activate bucket 0x100 in LFH
[*] spray
HeapAlloc() == 000002004FFB1A20
HeapAlloc() == 000002004FFB1B30
HeapAlloc() == 000002004FFB1C40
HeapAlloc() == 000002004FFB1D50
HeapAlloc() == 000002004FFB1E60
HeapAlloc() == 000002004FFB1F70
HeapAlloc() == 000002004FFB2080
HeapAlloc() == 000002004FFB2190
HeapAlloc() == 000002004FFB22A0
HeapAlloc() == 000002004FFB23B0
HeapAlloc() == 000002004FFB24C0
HeapAlloc() == 000002004FFB25D0
HeapAlloc() == 000002004FFB26E0
HeapAlloc() == 000002004FFB27F0
HeapAlloc() == 000002004FFB2900
HeapAlloc() == 000002004FFB2A10
HeapAlloc() == 000002004FFB2B20
HeapAlloc() == 000002004FFB2C30
HeapAlloc() == 000002004FFB2D40
HeapAlloc() == 000002004FFB2E50
HeapAlloc() == 000002004FFB2F60
```

דריכה בין userblocks

נתחיל מדוגמא פשוטה יחסית. לצורך ההמחשה, בואו נניח שיש לנו buffer בגודל 0x40 שבלוגיקה המטפלת בו יש חולשה שמאפשרת לנו לדרוך ממנו קדימה בזיכרון באורך כרצוננו. בנוסף, יש לנו מבנה מעניין שאנו רוצים לדרוך עליו, והוא בגודל 0x100. בגלל שמדובר בגדלים שהם ב-granularity שונה, הם יפלו בהכרח ב-userblocks נפרדים, ולצערנו יהיו מאוד רחוקים בזיכרון אחד מהשני. מצב זה מעמיד אותנו בבעיה.

במקום לקרב את ה-chunk-ים בזיכרון, אנחנו יכולים לקרב את ה-userblocks הרלוונטיים אחד לשני. ראינו בחלק הקודם שאלקוץ userblocks מטופל על ידי ה-backend, שזה ה-NT heap, ולכן אחרי ריסוס, ה-userblocks דווקא כן יכולים להופיע בצורה רציפה בזיכרון. אם יש שני userblocks צמודים למרות שהם ב-granularity אחר, ונוכל לדרוך מ-chunk שנמצא בסוף הראשון על chunk בתחילת השני, נוכל לפתור את בעיית ה-ganularity השונה.

כדי לעשות זאת, נתחיל ב לגרום ל-userblocks של גודל מסויים ליפול אחד אחרי השני בצורה רציפה. איך? צריך לעצב את ה-NT heap בצורה עקיפה. נניח שאנחנו מרססים המון עם פרימיטיב אלקוץ כלשהו, בגודל קבוע:

0. בהתחלה נמלא המון חורים ב-userblocks של אותו הגודל שאולי קיימים. הם עלולים להכיל chunk-ים משוחררים שהריסוס יתחיל לתפוס ולמלא.

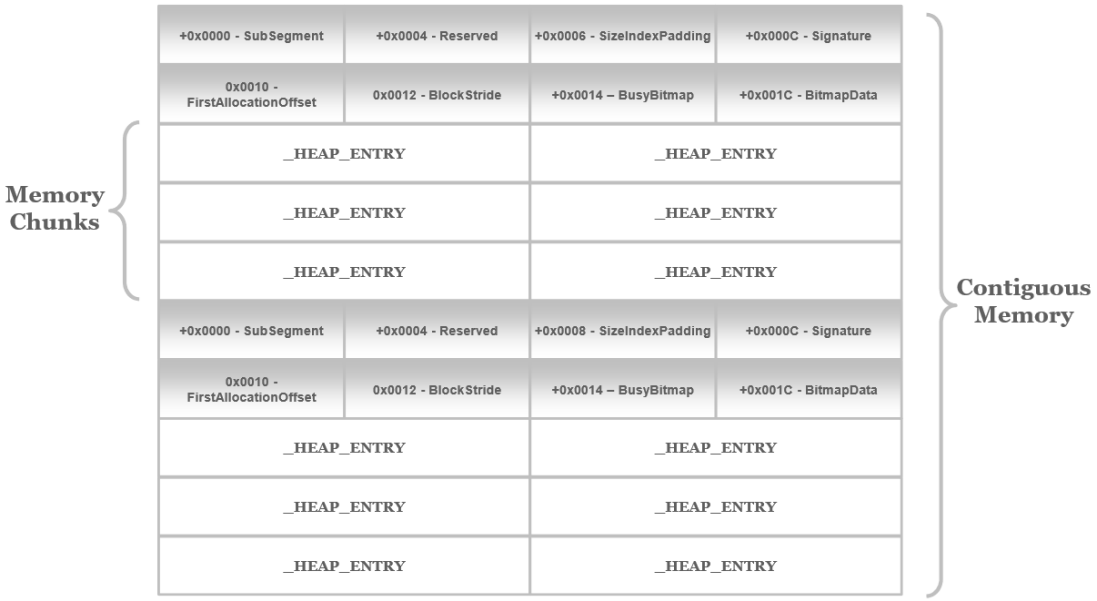
1. אחרי שנרסס מספיק, לא יהיו יותר userblocks עם chunk-ים משוחררים, אז ה-LFH יפנה ל-backend כדי לאלקץ userblocks חדשים. ה-backend יאלקץ לו כמה page-ים, ה-LFH ישבור אותם ל-chunk-ים, וימשיך לשרת את הבקשות שלנו מה-userblocks שהוא קיבל.

2. אם ניצור המון userblocks, נמלא חורים של pageים משוחררים ב-backend. דומה לתחילת התהליך, עכשיו ברמת ה-NT heap ולא ברמת ה-LFH.

3. אחרי שניצור מספיק userblocks, חדשים ול-backend יגמרו ה-page-ים המושחררים, הוא יתחיל להגיד את ה-heap שלו למעלה בכתובות. בגלל שה-backend כן מאלקץ page-ים בצורה רציפה וירטואלית, אז גם ה-userblocks שלנו יפלו בצורה רציפה וירטואלית.

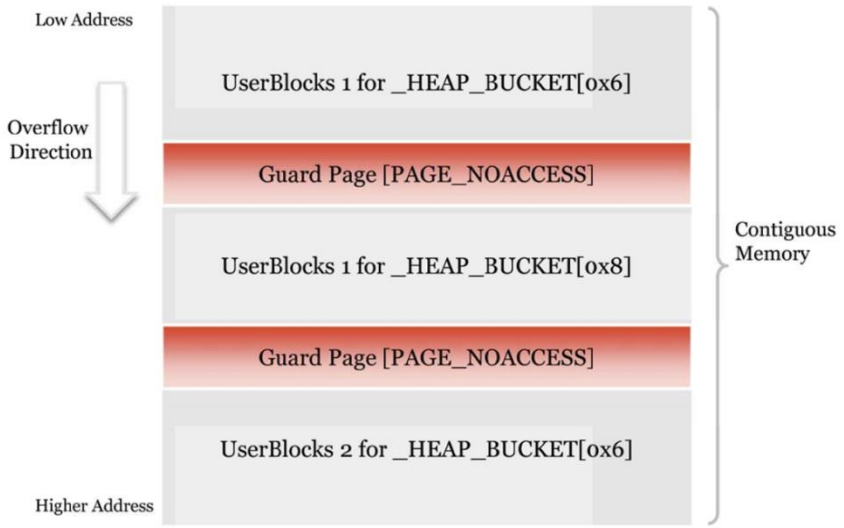
אבל בדוגמא הזאת אנחנו רוצים ש-userblocks מגדלים שונים יפלו אחד אחרי השני. עלינו לעשות את המתואר לעיל, עבור גדלים שונים, במקביל.

נרסס בשני הגדלים הרלוונטיים לסירוגין, ובסוף נקבל מצב ש-userblocks של גדלים שונים יפלו אחד אחרי השני.



בתמונה ניתן לראות שני userblocks שנפלו אחד אחרי השני בזיכרון, כולל ה-header-ים שלהם.

החבר'ה ב-MS חשבו על תקיפות מהסוג הזה, ולכן לפני כמה שנים נכנסה mitigation שקובעת שברגע ש-userblocks מגיע לכמות מקסימלית של chunk-ים בתוכו (0x400 chunk-ים או בגודל כולל של 0x78000 בתים), מאלקצים אחרי ה-userblocks פעם אחת GUARD PAGE, שזה פשוט page עם הרשאות PAGE_NOACCESS, שכל dereference אליו יגרום ל-segfault. כמובן שבהינתן הפרמיטיבים הנכונים, אנו עדיין יכולים לשחק ולגרום למצבים מעניינים, אבל זה מקשה מאוד בלא מעט מהמקרים. לדוגמה, אם יש כתיבה לינארית, בסגנון memcpy עם שליטה בגודל, מחוץ לגבולות הבאפר, והמטרה היא להגיע לאובייקט שנמצא ב-userblock אחר (כי הוא בגודל אחר), אז בדרך אליו בסיכוי גבוה נתקל ב-Guard page ונקרוס.



שיטה זו מתאימה למקרים בהם באופן טבעי אין המון אלקוצים, ואז אפשר להמנע מהשלב שבו יווצר ה-guard page, או למצבים שבהם יש לנו כתיבה ב-offset יחסי כך שנוכל לקפוץ מעל ה-guard page.

דריכה באותו ה-Userblocks

נניח בחלק זה שיש לנו שליטה טובה ב-heap (אלקוצים ושחרורים יחסית כרצוננו), חולשה שנותנת לנו דריכה לינארית, ואובייקט שאנחנו רוצים לדרוך עליו כדי להגיע להרצת הקודם. לצורך דוגמא זו נניח שכל האובייקטים נופלים באותו granularity. מצב זה יכול לקרות גם עם סוגים שונים של אובייקטים, אם הם במקרה בגדלים דומים, או אם יש בתוכם דברים שהם variable length כמו מערכים או סטרינגים ואנחנו יכולים לאזן שני סוגי אובייקטים שונים לגודל דומה. למעשה זהו מצב די נפוץ, בו יש לנו אובייקט של מערך שהוא בגודל משתנה ואנחנו רוצים לדרוך על שדה האורך שלו.

הבעיה המרכזית שנצטרך להתמודד איתה היא איך אפשר לדעת בסבירות גבוהה שכשנפעיל את החולשה, יהיה אובייקט מעניין מיד אחרינו בזיכרון?

- אנחנו לא יודעים את סדר האלקוצים בתוך ה-userblocks. יש המון layoutים אפשריים שיכולים להיווצר. אנו לא יכולים ליצור עיצוב יציב שיעבוד בסיכוי גבוה, שכן סדר האלקוצים לא דטרמיניסטי.
- חמור מכך - גם אם יובטח לנו שהמבנה המעניין לדריכה יפול אחרי הבאפר שממנו דורכים - אנו לא יודעים באיזה מרחק הוא. כמה נדרוך?
- גם אם יובטח לנו אורך מקסימלי בין הבאפר לבין המבנה המעניין - אנחנו לא יכולים לדרוך עם כמות גדולה מדי של בתים, כי אם נפלנו בסוף ה-userblocks כבר נדרוך החוצה, וזה מצב שבסיכוי מאד גבוה ייגמר בקריסה (במידה ויש אחרינו Guard Page, נחטוף קריסה מיידית).

מה שנוכל לעשות זה להילחם ברנדום בעזרת רנדום!

0. נרסס עם אובייקט כלשהו שאנחנו רוצים לדרוך עליו.
1. נשחרר חצי מהאובייקטים שאלקצנו בסעיף הקודם (כמובן, לאו דווקא כל אלקוץ שני, כי גם כך הוא מתאלקצים רנדומלית). כך יצרנו חורים בצורה רנדומלית בין האלקוצים והחורים.
2. נאלקץ אובייקט שממנו אפשר לדרוך קדימה, ונדרוך על ה-chunk הסמוך (או כמה סמוכים).

בסיכוי כלשהו הגענו לכתיבה על המבנה שרצינו. אפשר להגדיל את הסיכוי עם כתיבה ארוכה יותר, אך אז נגדיל גם את הסיכוי שנקרוס במידה והתאלקצנו בסוף ה-userblocks. יש פה tradeoff בין הסיכוי שלנו לדרוך באלקוץ מוצלח לבין הסיכוי שלנו לקרוס באלקוץ לא מוצלח (בסוף ה-userblocks).

במידה ויש לנו שליטה מלאה או חלקית על הגודל שאנו עובדים איתו, עדיף לבחור גודל שייצור לנו שארית בסוף ה-userblocks, ולכן נוכל להמשיך לדרוך על זיכרון מיותר שנשאר בין סוף ה-chunk-ים ב-userblocks לסוף ה-page.

Target #4		Free		Target #1	
			Target #5		Target#2
	Free	vuln chunk		Target #7	
Free		Target #9		Free	
	Target #3	Free		Target #8	Target #6

כאן רואים layout לדוגמא של ה-heap לאחר העיצוב. שימו לב שהאלקוצים מכל הסוגים מפוזרים בצורה אקראית, ויצא לנו שהתאלקץ chunk שניתן לדרוך ממנו שניים לפני chunk שמעניין אותנו לדרוך עליו. אם כתבנו קדימה מספיק, הגענו אליו.

בדריכה כזאת חשוב לשים לב לשחרורים של chunk-ים. הכרחי שלא ישתחרר אף chunk שדרכנו עליו בדרך עד למבנה המעניין, כי אם כן נקרוס מיד, מכיוון שדרכנו על ה-header שלו. כדי לתקן את זה נצטרך infoleak רלטיבי כדי לקרוא את ה-header-ים הקודמים, או דרך לשבור את ה-cookie ולהוציא את כתובת ה-heap כדי לחשב את ה-header. חשוב להבין שהנקודה שבה נקרוס היא כאשר בשחרור של ה-chunk נרצה להגיע ל-userblocks שלנו, וזה כמובן יתבצע ע"י החלק המקודד ב-header (כפי שראינו למעלה). אם לא נזייף אותו בצורה טובה, ונפנה אותו לאזור זיכרון שאכן נראה ובנוי כמו userblocks - נקרוס על dereference-ים.

יש כמה יתרונות ב-LFH שמקלים על ניצולים מהבחינה הזאת. גם אם הקריאה הרלטיבית מתבצעת אחרי שיש אלקוצים ושחרורים נוספים באותו userblocks, עדיין ה-header-ים של ה-chunk-ים ישארו כמו שהם. זה מכיוון שה-header נקבע רק ביצירת ה-chunk ולא משתנה באלקוץ או שחרור. כמובן שאם אנחנו קוראים וכותבים מ-chunk-ים שונים לא נוכל לדעת מה ה-offset ל-header הנכון כי הוא שונה לכל chunk. אבל המצב שבו יש לנו אובייקט שמאפשר קריאה וכתיבה בלי לשחרר ולאלקץ אותו בינהם, הוא מספיק טוב, גם אם בעצם הקריאה או הכתיבה אנחנו משפיעים על אלקוצים ושחרורים של chunk-ים אחרים באותו userblocks.

יתרון נוסף הוא שהנקודה שבה נקרוס היא רק בשחרור ולא באלקוץ (השדה שנבדק באלקוץ אינו מוגן מפני דריכה באמצעות xor, ולכן אנו יודעים עם מה אנו צריכים לדרוך, והמצב היחיד שנקרוס הוא בשחרור של chunk שה-header שלו לא נכון (ברוב האלוקטורים נקרוס גם בפעולות על chunk-ים קרובים או בגלל דריכה על freelist). הצורה הפשוטה ביותר לנצל זאת היא אם אנחנו יכולים להחזיק chunk-ים מאולקצים

לאורך זמן. גם אם לא, מספיק שנשמור על זה שבסיכוי סביר chunk-ים משוחררים לא יאולקצו, ומשם אפשר כבר לתקן את ה-userblocks אחרי שהרצנו קוד כדי להחזיק את ה-process חי.

ניצול UAF

בגלל האקראיות של סדר האלקוצים, ניצול UAF עבר להיות מאד לא יציב ובעייתי. כאשר אנו מנצלים UAF, אנחנו רוצים לאלקץ מבנה אחר בין השחרור של אובייקט כלשהו והשימוש בו. אם הצלחנו, נוכל לשלוט בתוכן של האובייקט באופן חלקי לפחות, ומשם נוכל להשפיע על המשך הקוד (ובתקווה להריץ קוד משלנו בסוף). כל התהליך הזה תלוי ביכולת שלנו לחזות מתי, או מהי כמות האלקוצים הדרושה, שבה chunk מסוים הולך להתאלקץ בדיוק באותה כתובת ש-chunk אחר שחרר קודם לכן. במילים אחרות, איך נוכל לגרום לכך שבסיכוי גבוה מאוד, ה-malloc שנמקם בין ה-free לבין השימוש יחזיר את ה-chunk ששוחרר?

האופציה הקלה היא שוב להילחם ברנדום בעזרת רנדום. אם אנחנו יכולים לאלקץ הרבה מהמבנה איתו אנחנו רוצים להחליף, בין ה-free לשימוש בחולשה, אז אפשר פשוט לרסס המון, ובתקווה נקבל אחרי כמות כלשהי של אלקוצים את ה-chunk ששוחרר, ועומדים לבצע בו שימוש.

אנחנו יודעים שכמות ה-chunk-ים ב-userblock חסומה (0x400) ולכן יש לנו חסם תיאורתי לכמות האלקוצים הנדרשת כדי לתפוס chunk משוחרר, אם אנחנו עובדים עדיין באותו userblocks. במציאות, גם עם פחות אלקוצים אפשר להגיע בהסתברות טובה לתפיסת הכתובת הרצויה. הנה תוצאה של ניסוי שבודק אחרי כמה אלקוצים מקבלים את הכתובת האחרונה ששוחררה. זה נעשה על Windows 10 build 17134. דאגתי להכניס ל-heap קצת "לכלוך" קודם לכן - לבצע כמה אלקוצים ושחרורים באופן לא אחיד ובסדר אקראי, על מנת לא לעבוד ב-heap נקי מדי. התוצאות תלויות מקרה ו-bucket, אבל אפשר לראות שגם עם כמות אלקוצים קטנה משמעותית מ-0x400 היינו מצליחים לתפוס את הכתובת הרצויה:

```
C:\projects\LFH>LFH_tester.exe
[*] activate bucket 0x100 in LFH
[*] for fun and "fair" game
[*] we passed 20 allocations until we got the last freed chunk
[*] we passed 38 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 21 allocations until we got the last freed chunk
[*] we passed 27 allocations until we got the last freed chunk
[*] we passed 3 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 8 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 21 allocations until we got the last freed chunk
[*] we passed 30 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 3 allocations until we got the last freed chunk
```

גם כאשר אנחנו מוגבלים במספר האלקוצים, עדיין יש לנו אפשר לרסס את ה-userblocks בתוכן. בניגוד לאלוקטורים אחרים, ה-LFH לא נוגע ב-data של chunk משוחרר. לכן, אם כתבנו ל-chunk ושחררנו אותו, אז שכאותו chunk יתפס באלקוץ חדש, ה-uninitialized data שלו יכיל בדיוק את אותו התוכן שנכתב עליו מהאלקוץ הקודם. אפשר לנצל את זה כדי לרסס data ב-userblocks בלי באמת להחזיק chunk-ים תפוסים. לצורך הזה אפשר להסתפק בפרימיטיב מאוד חלש ועדיין להגיע לתוצאות יפות בניצול UAF, אם אפשר לעשות מספיק כתיבות באופן הזה בין ה-free לשימוש.

שיטות מתקדמות

ראינו שסדר האלקוצים האקראי ב-LFH מקשה משמעתית, ולכן לעתים עדיף להמנע ממנו מראש. האם יש דרך לא להיות ב-LFH?

בסיכוי מאד גבוה גודל אלקוץ סביר יהיה ב-LFH. זה קורה בגלל שהאובייקטים שאנחנו משתמשים בהם עבור הפרימיטיבים לניצול הם בדרך כלל בשימוש נפוץ ב-flow המרכזי, או שהם דומים בגודלם לאובייקטים אחרים שכבר הפעילו את אותו bucket. גם אם אין כאלו בכלל באופן ישיר בקוד, יכול להיות שאובייקטים כאלו נוצרו דרך ה-CRT או ספריות אחרות. הכמות הנמוכה של האלקוצים הדרושים ל- bucket activation, 0x11, מובילה לכך שבסיכוי גבוה נצטרך להתמודד עם LFH. טריק אחד שיכול לעזור הוא לשחק עם השליטה שלנו בגודל של האובייקטים, באמצעות סטרינגים או variable length arrays שנמצאים בתוך האובייקט, כדי "להקפיץ" אותו בין bucketים ולהגיע ל-bucket שעדיין לא activated. גם אם זה המצב, לא יהיו לנו הרבה אלקוצים כאלה כדי לשלוט בצורה טובה ב-NT heap לפני שנעבור ל-LFH - הרי אחרי 0x11 אלקוצים כבר נמצא את עצמנו שוב ב-LFH.

דבר אחד שאפשר לנצל הוא ש-LFH כ-frontend משרת רק אלקוצים עד ל-0x4000. כל אלקוץ גדול יותר יגיע ישירות ל-backend, וינוהל בידי ה-NT heap, שם אין את ה-mitigations שיש ב-LFH. כמובן שלא תמיד נוכל לעבוד עם גדלים כאלה, אבל במידה וכן, זה יאפשר לנצל את ה-heap בצורה קלאסית יותר.

אפשר גם לחשוב על ניצולי data אחרים של ה-metadata של ה-LFH, השיטות הללו הרבה פחות גנריות ממה שתואר כאן, אבל מתאימות למקרים ספציפיים מאוד. נבחן מקרה אחד כזה לדוגמה.

ה-PreviousSize, שהוא שדה ב-header של כל chunk, שבמידה וה-chunk ב-LFH, משמש את השחרור כדי לדעת את ה-index של ה-chunk, שממנו מחשבים את ה-offset של הביט הרלוונטי ב-bitmap של ה-userblocks.

התהליך נראה ככה:

```
int RtlpLowFragHeapFree(_HEAP *Heap, _HEAP_ENTRY *Header)
{
        .
        .
        .
        short BitmapIndex = Header->PreviousSize;

        //Set the chunk as free
        Header->UnusedBytes = 0x80;

        bittestandreset(UserBlocks->BusyBitmap->Buffer, BitmapIndex);

        .
        .
        .
}
```

בניגוד ל-pointer ל-subsegment, ה-PreviousSize הוא חלק ב-header שדווקא לא עובר xor. הוא כן נמצא בזיכרון אחריו, אז בכתיבה רציפה נאלץ להרוס את ה-header ונקרוס, אבל אם יש לנו כתיבה ב-offset (או יכולת לקרוא את ה-header קודם), אז יהיה ניתן לכתוב בו ערך שגדול יותר מהגודל של ה-bitmap. במידה ונעשה זאת, ונשחרר את ה-chunk (בלי לפגוע ב-header שלו, כי דרכו מגיעים ל-userblocks) - נוכל לגרום לכתיבה של הביט 0 ב-offset כלשהו שרלטיבי ל-bitmap שלנו. כמובן שזה פרימיטיב משונה קצת, אבל יכול להועיל במקרים מסויימים. שיטה זו מתוארת בהרחבה כאן. פרקטית, מעולם לא השתמשתי בה במקרה אמיתי.

מעבר למשחקים עם ה-metadata באזור ה-chunk, אפשר גם לתקוף את האלוקטור עצמו כדי לעבור את ה-mitigations שלו. לדוגמה, אם נמצא חולשה ב-random שבשימוש ה-LFH, היא תאפשר לנו לשלוט על ה-layout של ה-heap בצורה דטרמיניסטית, ועל כן נמנע מרוב הבעיות שתוארו. נוכל לנצל פרימיטיבים שלא היה סביר לנצל אותם בלי לחזות את הסדר, וגם אם כן, להעלות משמעותית את היציבות של ניצולים שהיו עובדים גם בלי זה. מסתבר שעד ל-Windows 10 16129, הייתה בדיוק חולשה כזאת, אז שווה לבחון אותה לעומק.

על מנת להבין את החולשה ואת הדרך לנצל אותה, צריך להבין איך מנגנון רנדום סדר האלקוצים עובד. אז כפי שציינו, לכל userblocks יש bitmap, ובו ביט אחד לכל chunk. כשצריך לאלקץ chunk, ה-LFH יחפש ביט אחד שמעיד על chunk פנוי ב-bitmap, יסמן את ה-chunk הזה כמאולקץ ויחזיר לנו אותו. ראינו שחיפוש הביט הפנוי מתחיל מ-offset אקראי ב-bitmap. אז סדר האלקוצים תלוי בתפוסת ה-chunk-ים ב-userblocks וב-offset האקראי. איך הוא מגריל את המספר?

מסתבר שיש מערך באורך 0x100 סטטי ב-ntdll (מוגדר כסימבול, ntdll!RtlpLowFragHeapRandomData), וכש-process נוצר, ממלאים אותו ב-0x100 ערכים אקראיים, עם רנדום קריפטוגרפי חזק:

```
RtlpInitializeLfhRandomDataArray proc near

arg_0= qword ptr  8

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
lea     rbx, RtlpLowFragHeapRandomData
; amarsa: the size of the random
; array is 0x20 * 8 == 0x100
mov     edi, 20h

loop_fill_random_array:
call    RtlpHeapGenerateRandomValue64
mov     rcx, 7F7F7F7F7F7F7F7Fh
and     rax, rcx
mov     [rbx], rax
lea     rbx, [rbx+8]
sub     rdi, 1
jnz     short loop_fill_random_array

mov     rbx, [rsp+28h+arg_0]
add     rsp, 20h
pop     rdi
retn
```

איך המערך הזה עוזר לנו לבחור ביט שמציין chunk משוחרר בצורה אקראית במתוך כל ה-bitmap? יש index שרץ על המערך של הערכים האקראיים בצורה ציקלית, ובכל אלקוץ לוקחים את הערך האקראי שנמצא במערך במקום ה-index, והוא משמש לנו כ-index לתחילת החיפוש ב-bitmap של ה-userblocks הרלוונטי. החל מה-index הזה סורקים קדימה עד שמגיעים ל-bit שמציין chunk משוחרר, ואותו מחזירים.

החולשה הייתה בכך שהערכים הללו נשארו קבועים לאורך כל הריצה. למרות שכל הערכים אקראיים, אנחנו יכולים לנצל פה דטרמיניזם מסויים: אומנם הסדר הוא אקראי, אבל אחרי מחזור של 0x100 אלקוצים מובטח לנו שנחזור לאותו סדר אקראי.

מה אם נבצע אלקוץ אחד, ואז נוכל "לקדם" את ה-index שרץ על מערך ערכי הרנדום שלנו עם אלקוצים ושחרורים, כך שיחזור לבדיוק אותו ה-index שהשתמשנו בו? זה יבטיח לנו שני דברים:

- רציפות באלקוצים - אחרי קידום שכזה האלקוץ הבא יתחיל לסרוק את ה-bitmap בדיוק מאותה נקודה. אם שמרנו על האובייקט הראשון מאולקץ וקיים chunk אחריו, מובטח לנו שקיבלנו אותו, וכך מיקמנו שני אובייקטים באופן רציף בזיכרון.

```
chunk = HeapAlloc(hHeap, 0x0, size);
printf("[*] Chunk 0x%p is freed in the userblocks for bucket size 0x%x\n", chunk, size);

for (size_t i = 0; i < RandomDataArrayLength - 1; ++i) {
    tmp_chunk = HeapAlloc(hHeap, 0x0, size);
    if (!tmp_chunk) {
        return FAIL;
    }
    HeapFree(hHeap, 0x0, tmp_chunk);
}

tmp_chunk = HeapAlloc(hHeap, 0x0, size);
```

- ניצול של UAF יציב - אפשר לשחרר אובייקט, לקדם את האינדקס לאותה הנקודה, ולקבל בדיוק את אותה הכתובת עבור אובייקט אחר שישב מעליו. כמובן שזה עדיין דורש שהם יהיו באותו granularity.

```
chunk = HeapAlloc(hHeap, 0x0, size);
HeapFree(hHeap, 0x0, chunk);
printf("[*] Chunk 0x%p is freed in the userblocks for bucket size 0x%x\n", chunk, size);

for (size_t i = 0; i < RandomDataArrayLength - 1; ++i) {
    tmp_chunk = HeapAlloc(hHeap, 0x0, size);
    if (!tmp_chunk) {
        return FAIL;
    }
    HeapFree(hHeap, 0x0, tmp_chunk);
}

tmp_chunk = HeapAlloc(hHeap, 0x0, size);
```

על הגרסה האחרונה שפגיעה, הקוד המלא יחזיר את התוצאה הצפויה:

```
C:\WINDOWS\system32\cmd.exe
[*] activate LFH bucket for size 0xc0

---------------------Check randomization---------------------------------
[*] Good, different allocations:
        0x011E5058
        0x011E4568
[*] Good, non contiguous allocations:
        0x011E4950
        0x011E4248

------------------------UAF Exploit-------------------------------------
[*] Chunk 0x011E4A18 is freed in the userblocks for bucket size 0xc0
[*] Success! chunk 0x011E4A18 is returned!

--------------------Contiguous Exploit----------------------------------
[*] Chunk 0x011E4310 is freed in the userblocks for bucket size 0xc0
[*] Success! 0x011E43D8 chunk is returned!
Press any key to continue . . .
```

אנו רואים שאכן הצלחנו לעקוף את מנגנון הרנדום, גם למטרה של UAF (לקבל את ה-chunk האחרון ששחררנו), וגם על מנת להשיג רציפות באלקוצים.

ב-MS לקחו זאת לתשומת ליבם, והציגו תיקון. התיקון פשוט מאד - כשה-index על מערך ערכי הרנדום מגיע לסוף, במקום לעשות wrap around ולהתחיל מ-0, קוראים עוד פעם אחת ל-RtlpHeapGenerateRandomValue32(), וקובעים את ה-index הבא להיות רנדומלי (בין 0 ל-0xff כמובן), משם ממשיכים לרוץ לינארית.

```
if ( v19 && *(_QWORD *)v13 == _R13 && (_WORD)v18 )
{
  v_teb = NtCurrentTeb();                // amarsa: this is the logic of
                                         // picking new value from the RandomDataArray
  v21 = RtlpSearchWidth[*(unsigned __int16 *)(_R13 + 0xAC)];
  v_randValue = RtlpLowFragHeapRandomData[v_teb->LowFragHeapDataSlot];
  v_teb->LowFragHeapDataSlot = (v_teb->LowFragHeapDataSlot + 1) & 0xFF;
```

למרות ששיטה זו תוקנה כבר, זו דוגמה טובה ששווה להבין מנגנונים כאלה לעומק כדי למצוא חולשות בתכנון שלהם. במקרה של ה-LFH, יש bounty על חולשות מסוג הזה, שמגיע ל-100,000$

דיבאג

לפני שנסיים, כמה טיפים קצרים להתמודד עם כל זה. כשאתם מדבגים process כלשהו ורוצים לראות איך ה-LFH שלכם נראה, כדאי להשתמש בפקודות builtin של WinDbg:

- !heap -p -a <addr>
- !heap -p -all
- !address <addr>

בואו נבחן מעט מהפקודות האלה בדוגמאות. לצורך העניין, נניח שבוחנים את הקוד הבא:

```
#include <stdio.h>
#include <Windows.h>

int main(void) {

    HANDLE hHeap;
    hHeap = HeapCreate(0, 0, 0);

    for (size_t i = 0; i < 0x12; i++) {
        HeapAlloc(hHeap, 0x0, 0x40);
        HeapAlloc(hHeap, 0x0, 0x100);
        HeapAlloc(hHeap, 0x0, 0x200);
    }

    for (size_t i = 0; i < 0x30; i++) {
        HeapAlloc(hHeap, 0x0, 0x40);
        HeapAlloc(hHeap, 0x0, 0x100);
        HeapAlloc(hHeap, 0x0, 0x200);
    }

    return 0;
}
```

במידה ונרצה לראות את ה-layout של ה-heap, נוכל להריץ את הפקודה הבאה: heap -p –all! היא תציג לנו את כל ה-chunk-ים ב-heap, על פי הסדר שלהם בזיכרון. עבור כל אחד נראה את הכתובת שלו, גודל האלקוץ וה-state שלו (משוחרר/מאולקץ). בפועל הפקודה תציג 2 כתובות: הראשונה היא הכתובת של ה-header, שהיא בדיוק 8 בתים לפני הכתובת של ה-chunk כפי שחזרה ל-caller.

על התוכנית הזאת, התוצאה של הפקודה תראה בערך ככה:

```
  01532d18 0041 0041  [00]   01532d20    00200 - (busy)
  01532f20 0041 0041  [00]   01532f28    00200 - (busy)
  01533128 0041 0041  [00]   01533130    00200 - (busy)
  01533330 0041 0041  [00]   01533338    00200 - (busy)
  01533538 0041 0041  [00]   01533540    00200 - (busy)
  01533740 0041 0041  [00]   01533748    00200 - (busy)
  01533948 0041 0041  [00]   01533950    00200 - (busy)
  01533b50 0041 0041  [00]   01533b58    00200 - (busy)
  01533d58 0041 0041  [00]   01533d60    00200 - (busy)
* 01534048 0201 0041  [00]   01534050    01000 - (busy)
  01534078 0009 0201  [00]   01534080    00040 - (free)
  015340c0 0009 0009  [00]   015340c8    00040 - (free)
  01534108 0009 0009  [00]   01534110    00040 - (free)
  01534150 0009 0009  [00]   01534158    00040 - (free)
  01534198 0009 0009  [00]   015341a0    00040 - (free)
  015341e0 0009 0009  [00]   015341e8    00040 - (free)
  01534228 0009 0009  [00]   01534230    00040 - (free)
  01534270 0009 0009  [00]   01534278    00040 - (free)
  015342b8 0009 0009  [00]   015342c0    00040 - (free)
  01534300 0009 0009  [00]   01534308    00040 - (free)
  01534348 0009 0009  [00]   01534350    00040 - (free)
  01534390 0009 0009  [00]   01534398    00040 - (free)
  015343d8 0009 0009  [00]   015343e0    00040 - (free)
  01534420 0009 0009  [00]   01534428    00040 - (free)
  01534468 0009 0009  [00]   01534470    00040 - (free)
  015344b0 0009 0009  [00]   015344b8    00040 - (free)
  015344f8 0009 0009  [00]   01534500    00040 - (free)
  01534540 0009 0009  [00]   01534548    00040 - (free)
  01534588 0009 0009  [00]   01534590    00040 - (free)
  015345d0 0009 0009  [00]   015345d8    00040 - (free)
  01534618 0009 0009  [00]   01534620    00040 - (free)
  01534660 0009 0009  [00]   01534668    00040 - (free)
  015346a8 0009 0009  [00]   015346b0    00040 - (free)
  015346f0 0009 0009  [00]   015346f8    00040 - (free)
  01534738 0009 0009  [00]   01534740    00040 - (free)
  01534780 0009 0009  [00]   01534788    00040 - (free)
  015347c8 0009 0009  [00]   015347d0    00040 - (free)
  01534810 0009 0009  [00]   01534818    00040 - (free)
  01534858 0009 0009  [00]   01534860    00040 - (free)
  015348a0 0009 0009  [00]   015348a8    00040 - (free)
  015348e8 0009 0009  [00]   015348f0    00040 - (free)
  01534930 0009 0009  [00]   01534938    00040 - (free)
  01534978 0009 0009  [00]   01534980    00040 - (busy)
  015349c0 0009 0009  [00]   015349c8    00040 - (busy)
  01534a08 0009 0009  [00]   01534a10    00040 - (busy)
  01534a50 0009 0009  [00]   01534a58    00040 - (busy)
  01534a98 0009 0009  [00]   01534aa0    00040 - (busy)
  01534ae0 0009 0009  [00]   01534ae8    00040 - (busy)
  01534b28 0009 0009  [00]   01534b30    00040 - (busy)
```

אנחנו רואים כמה אלקוצים בגודל 0x200 מה-bucket המתאים, וכמה אלקוצים בגודל 0x40 ב-bucket שלהם. כמובן שה-output המלא ארוך מזה. שימו לב לחלוקה הברורה ויפה ל-userblocks. למרות שהגדלים השונים מאולקצים לסירוגין, הם יושבים בקבוצות נפרדות.

שימו לב שבסוף של userblocks, ה-debugger מעט מסתבך ונותן ערכים שקריים. נניח הכתובת 0x1534050, שהיא פשוט בתחילת header של userblocks ו-page חדש, מתפרסרת כבעלת גודל של

0x1000 בתים. זה כמובן לא נכון, וניתן גם לראות שהכתובת הבאה היא בסה"כ 0x30 בתים קדימה. פשוט צריך לזהות את הטעויות האלה ולהתעלם מהן, זה קורה לפעמים בפרסור של pageים ו-userblocks חדשים.

בנוסף, ניתן לבחון כתובת ספציפית, ולקבל מהפקודות האחרות נותנים ספציפית עליה, את ה-protection של הזיכרון, גודל אלקוץ וכו':

```
0:004> !heap -p -a 015343d8
    address 015343d8 found in
    _HEAP @ 1520000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        015343d8 0009 0000  [00]   015343e0    00040 - (free)

0:004> !address 015343d8

Usage:                  Heap
Base Address:           01530000
End Address:            0153b000
Region Size:            0000b000 (  44.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        01530000
Allocation Protect:     00000004          PAGE_READWRITE
More info:              heap owning the address: !heap 0x1520000
More info:              heap segment
More info:              heap entry containing the address: !heap -x 0x15343d8
```

סיכום

משמעותית קשה יותר לנצל חולשות heap ב-userspace ב-Windows מאשר בעבר. ראינו מספר שיטות להתמודד גם עם המנגנונים החדשים ביותר, אבל באופן גורף, ההשפעה על ניצול חולשות היא דרסטית. ה-LFH מוריד משמעותית את היציבות של חולשות מסוימות, ואף גורם לפרימיטיבים מסוימים להיות פרקטית לא נצילים. למרות השיפורים באבטחה, הפגיעה בביצועים היא מינימאלית (ואף קיים שיפור בביצועים בחלק מהמקרים).

בהשוואה למצב ב-kernel, מנגנון האלקוץ של ה-Windows Pools הוא צפוי (אין רנדום בכלל), יש coalesceים, ניתן לשבור chunk-ים בקלות ואפילו אין הגנה בסיסית על ה-headerים של chunk-ים (ה-metadata גלוי ולא עובר xor עם cookie). למרות זאת, גם עבור ניצולים קרנליים כדאי מאוד להתחיל להכיר את ה-LFH: החל מ-build 17723 ל-fast ring, ו-18204 ל-skip ahead, קיבלנו את kLFH, שהוא אלוקטור frontend שמבוסס על מקבילו ה-userspace-י. זהו שינוי מהותי שישפיע עמוקות על ניצולים עתידיים, גם בקרנל.

Windows הוא לא מה שהיה פעם.

לקריאה נוספת

- Windows 8 heap internals
- The segment heap
- ישן יותר, מ-windows 7 (חלק לא רלוונטי להיום) - Understanding the LFH
- חלק מהתמונות במאמר זה נלקחו מהמצגת הראשונה